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

$$H_\emptyset = S_\emptyset$$

$$H_n = \begin{cases} \underline{\text{if}}\ B_n\ \underline{\text{then}}\ H_{n-1}\ \underline{\text{else}}\ S_n & (n \text{ even}) \\ \underline{\text{repeat}}\ \text{if}\ B_n\ \text{then}\ H_{n-1}\ \underline{\text{else}}\ S_n\ \underline{\text{until}}\ C_n & (n \text{ odd}) \end{cases}$$

(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 ; exit', where P_i is defined recursively by

```
                 if not B_i then L_i          N_i: if not B_i then L_i
                 P_(i-1)                           P_(i-1)
   S_Ø           goto M_i                          goto M_i
             L_i: S_i                         L_i: S_i
             M_i:                             M_i: if not C_i then N_i

 if i = Ø     if i > Ø and i is even          if i > Ø and i is odd
```

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-1)
                                   L_n:  if Bn then L_(n-1)
                                         S_n
              P_n                        goto M_(n-1)
     M_(n+2): exit             M_(n+1):  exit

         n odd                           n even
```

P_i is defined by

```
                                 L_{i-1}: if B_{i-1} then L_{i-2}
L_Ø: S_Ø                                  S_{i-1}
M_1: if C_1 then M_3               M_i:   if C_i then M_{i+2}
L_1: if B_1 then L_Ø               L_i:   if B_i then L_{i-1}
     S_1                                  S_i
     goto M_1                             goto M_i

     i = 1                         i > 1, 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 NP-complete, even for planar graphs with in-degree and out-degree bounded by 2 [2]. Since NP-complete problems are widely conjectured to require exponential time for their solution, we do not try to find optimal translations for <u>arbitrary</u> 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 following four operations (see Figure 3):

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

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

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

<u>REPEAT</u> (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 <u>branching factor</u> of G, denoted e(G). The <u>back arcs</u> of G (denoted B(G)) are the arcs of the form (t,s(P)) introduced by REPEAT; the <u>scope</u> 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 <u>restricted</u> SFG is one constructed without any use of IF. If G be an SFG, the <u>restricted SFG corresponding to G</u> 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 <u>DAG edges</u> of G. A <u>partition</u> 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 <u>DAG path</u>; a <u>DAG partition</u> is one composed of DAG paths. The <u>cost</u> of the partition, c(p), is the number of paths in it. The <u>cost</u> of G, c(G), is the cost of a cheapest partition of G. Partition p is <u>optimal</u> if c(p) = c(G).

A path is a <u>top hook</u> if it starts at s(G) and a <u>bottom hook</u> if it ends at a node in EX(G). A partition is <u>top-open</u> if it contains a top hook, <u>bottom-open</u> if it contains a bottom hook, <u>open</u> if it contains both a top hook and a bottom hook, and <u>nice</u> 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 $PC(p_P,p_Q) = \{h_P\ h_Q\} \uplus (p_P - \{h_P\}) \uplus (p_Q - \{h_Q\})$. Otherwise $PC(p_P,p_Q) = p_P \uplus p_Q$.

(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 $PE(p_P,p_Q,i) = (i\ h_P) \uplus (p_P - \{h_P\}) \uplus p_Q$. Similarly, if p_P is not top-open, but p_Q is, $PE(p_P,p_Q,i) = (i\ h_Q) \uplus (p_Q - \{h_Q\}) \uplus p_P$.

(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 $PR'(p_P,t) = \{h_b\ t\} \uplus (p_P - \{h_b\})$ and $PR(p_P,t) = \{h_b\ t\ h_t\} \uplus (p_P - \{h_b,h_t\})$ 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'_P 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'_P 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 whcih 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.

<u>Case</u> <u>I</u>. 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.

<u>Case</u> <u>II</u>. 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.

<u>Case</u> <u>III</u>. p'_P is optimal, but p'_Q is not. The proof is similar to case II.

<u>Case</u> <u>IV</u>. 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'_P 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'_P is optimal but not nice. In this case, $c(p_G) = c(p'_G) = c(p_P)$. 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'_P 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 conpletes 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

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

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 partions 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, $c(p_P) \geq c(p_P) + c(p_Q) \geq e(P)+1+e(Q)+1 = e(G) + 1$.

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) $c(p_G) \geq \lceil (e(G)+1)/2 \rceil$

(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 yeilds 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, $\lceil (e(G)+1)/2 \rceil + \lfloor (e(G)+1)/2 \rfloor = e+1 \leq c(p_G') = c(p_G) + b(p_G) < \lceil (e(G)+1)/2 \rceil + b(p_G) < \lceil (e(G)+1)/2 \rceil + \lceil (e(G)+1)/2 \rceil$. Cancelling occurences of $\lceil (e(G)+1)/2 \rceil$ yeilds $\lfloor (e(G)+1)/2 \rfloor < b(p_G) < \lceil (e(G)+1)/2 \rceil$, which is impossible.

The proof of part (iii) is the same as part (ii), except all occurences of $<$ should be replaced by $\leq$.

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 $c(G) \geq c(P)+c(Q)-1 \geq$ $(e(G)-b(G)+1) + (e(Q)-b(Q)+1) - 1 = (e(P)+e(Q)) - (b(P)+b(Q)) + 1$ $= e(G)-b(G)+1$.

If $G = E(P,Q,i)$, then $c(G) \geq c(P)+c(Q) \geq$ $(e(G)-b(G)+1) + (e(Q)-b(Q)+1) = (e(P)+e(Q)+1) - (b(P)+b(Q)) + 1 =$ $e(G)-b(G)+1$.

If $G = R(P,t)$, then $c(G) \geq c(P)-1 \geq (e(G)-b(G)+1) - 1 =$ $e(P) - (b(P)+1) + 1 = e(G)-b(G)+1$.

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 $c(G_2) = \lfloor (e+1)/2 \rfloor$. If b is an integer such that $e-b+1 > \lfloor (e+1)/2 \rfloor$, 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 ($b(G_1) = \emptyset$). By Theorem 4.1, $c(G_1) = e+1$.

Let G_2 be the graph H_e defined in the introduction: $H_\emptyset = S_\emptyset$, $H_e = E(H_{e-1}, S_e, B_e)$ 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

$\{B_{2i}S_{2i}C_{2i+1}B_{2i+1}S_{2i+1} \mid 1 \leq i \leq \lfloor (e-1)/2 \rfloor\} \cup \{S_\emptyset C_1 B_1 S_1,\ B_e S_e\}$

(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

[1] F. Baskett, "The best simple code generation technique for WHILE, FOR, and DO loops," Sigplan Notices 13, 4, pp. 31-32 (April 1978).

[2] J. Plesnik, "The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound 2," Information Proc. Letters 8, 4, pp. 199-201 (April 1979).

[3] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," Journal of the ACM 21, 3, pp. 367-375 (1974).

[4] M. S. Hecht, Flow Analysis of Computer Programs, American Elsevier, New York (1977).

[5] F. T. Boesch and J. F. Gimpel, "Covering the points of a digraph with point-disjoint paths and its application to code optimization," Journal of the ACM 24, 2, pp. 192-198 (April 1977).

[6] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.," SIAM Journal on Computing 2, 4, pp. 225-230 (December 1973).

[7] T. G. Szymanski, "Assembling code for machines with span-dependent instructions," CACM 21, 5, pp. 300-308 (April 1978).

[8] R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton, "Can structured programs be efficient?," SIGPLAN Notices, pp. 10-18 (October 1976).

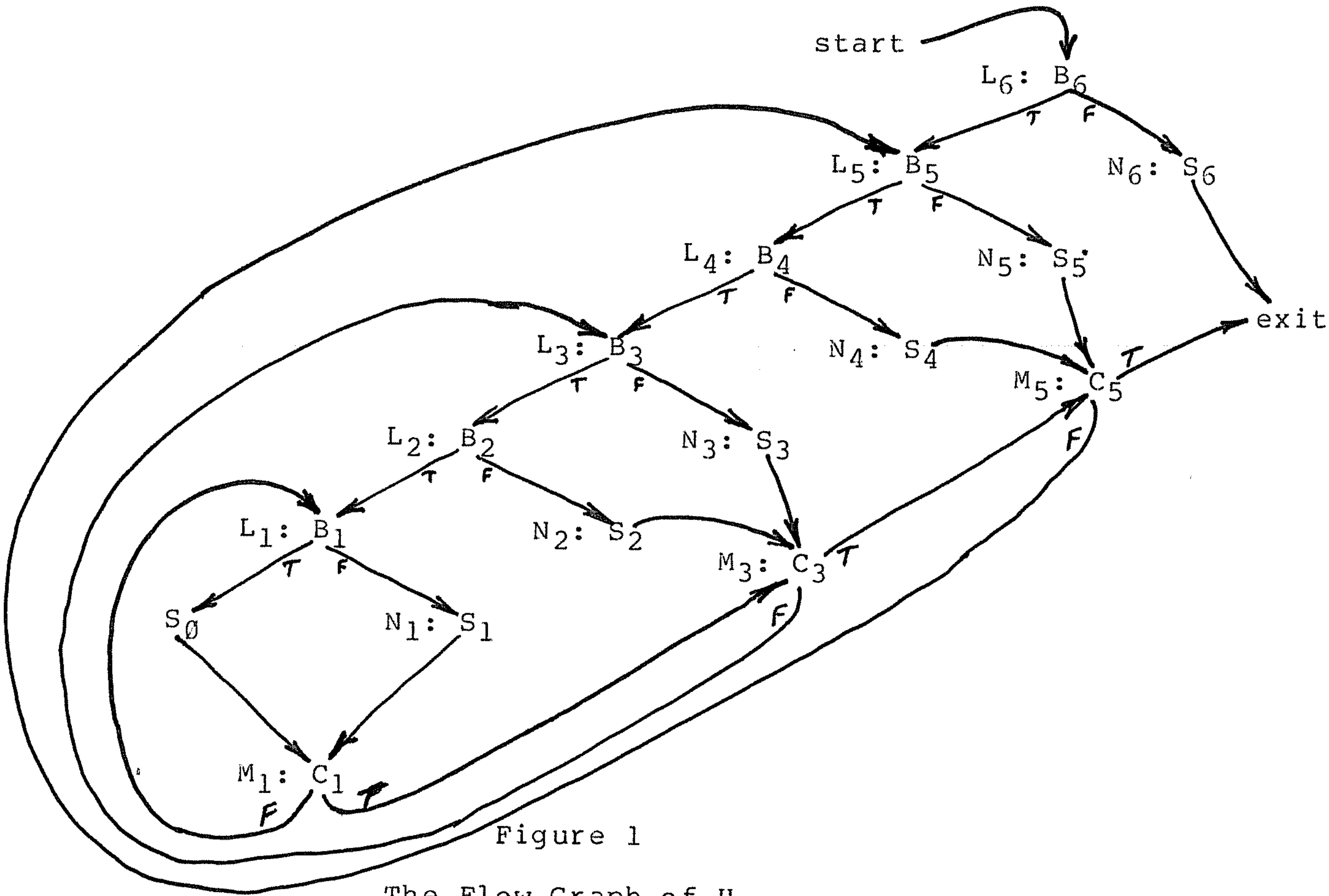

Figure 1

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
    SØ
    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
```

(a)

```
LØ: SØ
M1: if C1 then M3
L1: if B1 then LØ
    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
```

(b)

Figure 2

Two Translations of H_6

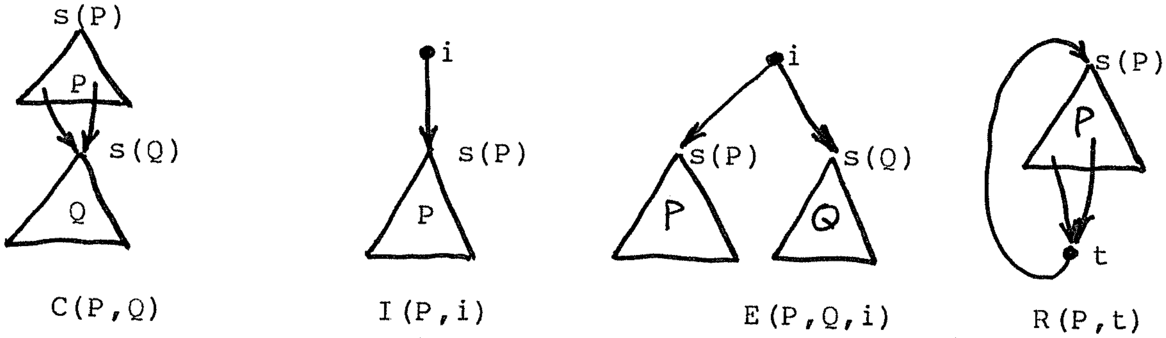

Figure 3

SFG Operations

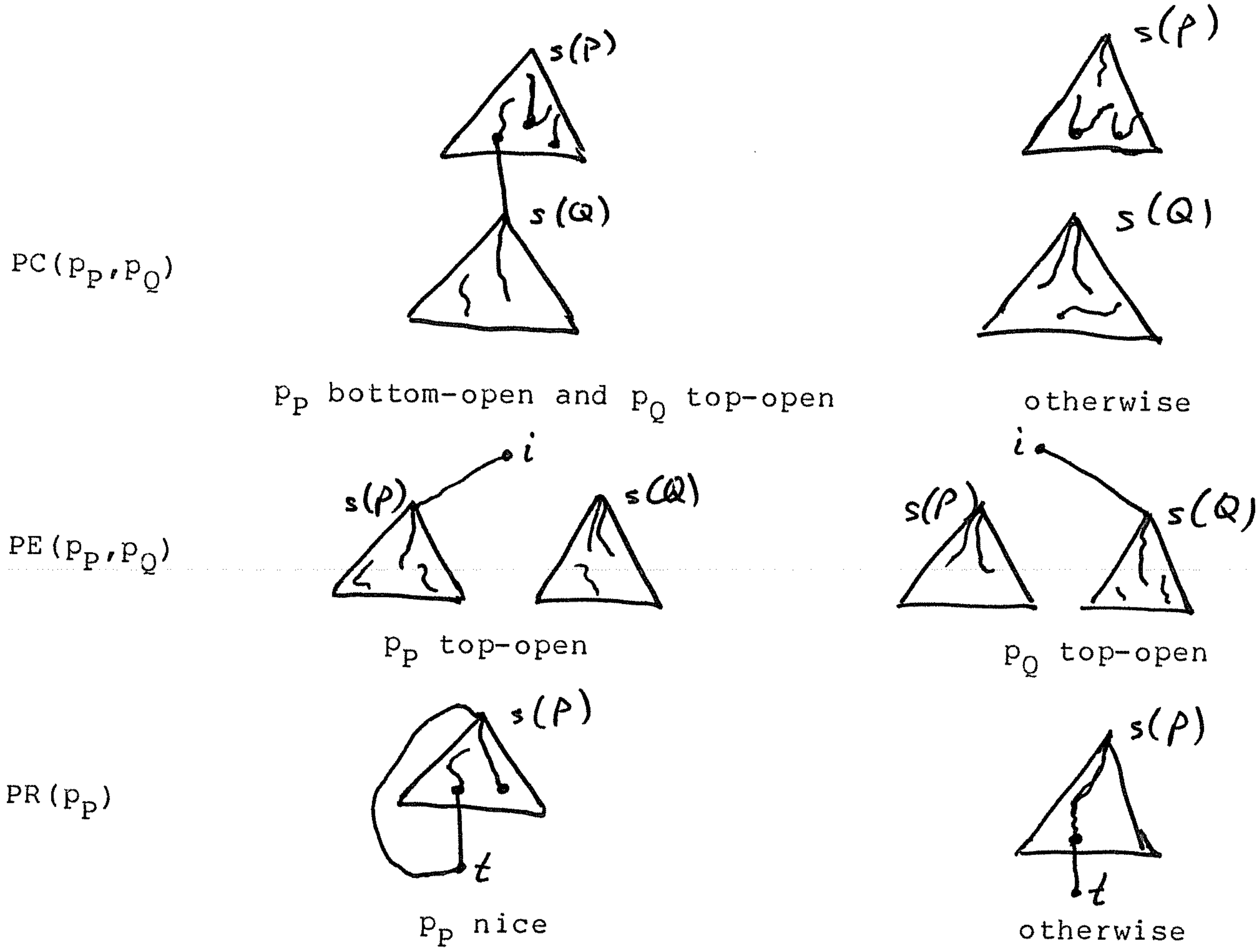

Figure 4

Operations for Combining Partitions

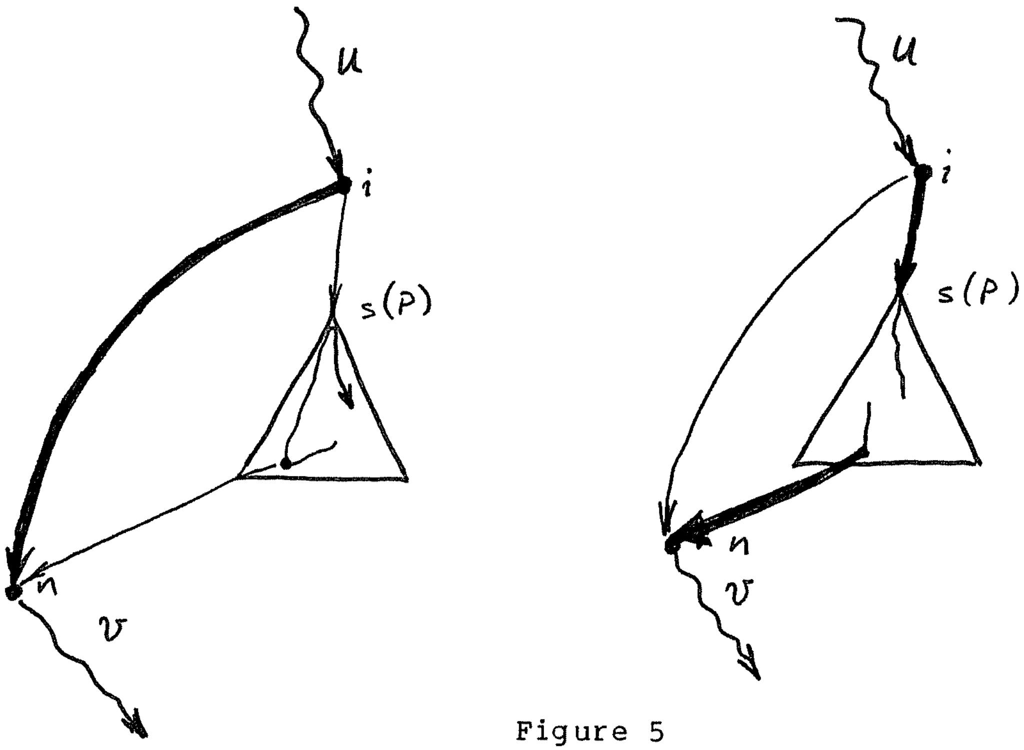

Figure 5

Eliminating Forward Arcs