The Complexity of Control Structures
and Data Structures

R. J. Lipton, S. C. Eisenstat,
and R. A. DeMillo*

Research Report #41

March 1975

* Department of Electrical Engineering
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin 53201

This research was supported in part by the US Army Research Office under grants number DAHC04-75-G-0037 and number DAHC04-74-C-0197 and by the Office of Naval Research under grant number N00014-67-A-0097-0016.

THE COMPLEXITY OF CONTROL STRUCTURES AND DATA STRUCTURES

R. J. Lipton,* S. C. Eisenstat†
Department of Computer Science
Yale University
New Haven, Connecticut 06520

R. A. DeMillo**
Department of Electrical Engineering
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin 53201

1. *Introduction*

The running time or computational complexity of a sequential process is usually determined by summing weights attached to the basic operations from which the process is derived. In practice, however, the complexity is often limited by how efficiently it can access its data structures and how efficiently it can control program flow. Furthermore, it has been extensively argued [4] that certain limitations on the process sequencing mechanisms available to the programmer result in more "efficient" representations for the underlying processes. In this paper we will examine these issues in an attempt to assess the "power" of various data and control structures.

A key observation about sequential processes is that they usually do not reference their data structures randomly. For instance, the many algorithms that organize their data structures as arrays often access the array elements in a "local" manner (e.g. the conventional matrix multiplication algorithm accesses its arrays by rows and by columns). In a paging environment, how one stores an array is especially important (cf. Rosenberg [14]). Therefore, it is natural to investigate how arrays can be stored so that elements "near" one another in the array are stored near one another. The basis for this comparison will be a relation $\leq_{B,1}$ defined so that for data structures G and G*, $G \leq_{B,1} G^*$ if G can be imbedded in G* in 1-1 fashion so that there is at most a B-fold increase in distance between embedded objects.

It is somewhat remarkable that an analogous study for control structures uses the same basic insights. It is well-known that process sequencing disciplines found in programming practice (e.g. "go to", "while") may simulate each other in a functionally equivalent way but that if, in addition, the simulation is constrained to preserve the structure of the original algorithm up to isomorphism, the desired simulations may not exist [1,2,3,9,10]. Obviously, the fundamental issue is neither the construction of functionally equivalent programs nor the inability to preserve structure exactly, but rather is the "naturalness" of the simulation. Control structures are compared by the relation $\leq_{B,M}$ for $M \geq 1$. If G and G* are algorithms with distinct process sequencing mechanisms, then $G \leq_{B,M} G^*$ indicates that G* simulates G by making at most M copies of each operation in G and increasing the cost of sequential access of embedded operations by a factor of at most B. (cf. Lipton [11])

Thus, comparing the power of data structures and control structures involves analyzing the 1-1 and many-one aspects of reduction (or simulation) techniques whose efficiency is bounded by B and M. In a natural way, the relation $\leq_{B,M}$ represents an intertwining of space (i.e., M) and time (i.e., B) complexities. The connection between our relation $\leq_{B,M}$ and the relations used by previous studies of control structure simulation is: (1) $\leq_{B,M}$ is weaker than isomorphism† since it allows both time and space to increase; (2) $\leq_{B,M}$ is stronger than functional or input-output equivalence; (3) $\leq_{B,M}$ makes no assumptions about adding program variables (as is made for example in [2]).

The plan of presentation is as follows. In Section 2, the basic combinatorial definitions used throughout the sequel are presented and the combinatorial models used for representing control structures and data structures are introduced. In Section 3 the relation $\leq_{B,M}$ is defined by means of graph embeddings. This relation is viewed as an embedding in the data structure case and as a simulation relation in the control structure case. Section 4 contains the main result for data structure embeddings: that for certain structures

---

* This research was supported in part by the US Army Research Office under Grant Number DAHC04-75-G-0037.

† This research was supported in part by the Office of Naval Research under Grant Number N00014-67-A-0097-0016.

** This research was supported in part by the US Army Research Office under Grant Number DAHC04-74-C-0179.

†Isomorphism in this sense is taken to mean strict computational equivalence; programs or program schemes are isomorphic if for any interpretation of the computation, we apply the same sequence of operations in the same order.

G, G^*, if $G \le_{B,1} G^*$, then $B \ge O(\log n)$ where n is the number of components of G. This result will also follow from our main theorem in Section 5, but since the $M = 1$ condition can be used extensively to simplify the relevant arguments, it is instructive to compare the two proofs. The main theorem in Section 5 generalizes the result in Section 4 by allowing $M \ge 1$. In this case, if $G \le_{B,M} G^*$ for certain natural choices of G, G^* it must be that $B + \operatorname{Log} M \ge O(\log n)$. A direct result of this theorem is that certain schema constructions, such as Engeler normal [5] form, cannot be achieved "uniformly" with respect to the $\le_{B,M}$ relation. More exactly, for any B and M there is a goto program G such that for no program H in Engeler normal form does $G \le_{B,M} H$ hold. Thus, the construction of Engeler normal forms - while always possible - does not preserve time and space in a bounded way. This result also demonstrates how our results will be asymptotic in their nature. For example, for any goto program G there and B and M such that $G \le_{B,M} H$ where H is in Engeler normal form; however, the values of B and M must grow with the size of the program G. In Section 6, the main simulation results for control structures are developed. The positive and negative results which ensue are responsible for the hierarchy of control structures shown in Figure 1. In figure 1, $\underline{X} \rightarrow \underline{Y}$ means that for no B and M does every G in $\underline{X}$ have an H counterpart in $\underline{Y}$ such that $G \le_{B,M} H$. The key result here is:

goto $\rightarrow$ label exit .

The class of label exit programs includes many of the standard constructs that are often allowed in "structured" programs. Therefore, this result states that there is a time-space speedup between goto programs and "structured" programs: there are goto programs whose only "structured" counterparts explode in either time or space. This result seems to make precise the comments in Knuth [8] on efficiency of goto and "structured" programs.

While the results contained in this paper are motivated by our interest in the power of data and control structures, it has not escaped our attention that they may have interest purely as combinatorial results; they add, for example, to the scarce literature on graph embeddings.

computed goto
↓
goto with d way branching ($d \ge 2$)
↓
label exits
↓
do forever
↓
while

Figure 1: Comparison of Control Structures

2. *The Combinatorial Representations*

We will represent both control structures and data structures by directed graphs. In the control case, the nodes of a graph G represent executable statements and the arcs possible flow of control; in the data case the graph nodes of the graph represent memory locations and the arcs successible elements. Thus, in either case, what is to be modelled is the "difficulty" of accessing nodes: the complexity of a control structure is given by the cost of accessing and sequencing non-control instructions, while the complexity of a data structure is determined by the cost of accessing successible data elements. Some care must be exercised in viewing control structures which are represented in this way; our representations do not always correspond to (temporal) flow-of-control and are not to be looked at as flowcharts. Rather, what is being modelled is the potential control connectivity of an underlying algorithm or process. Each class of control structure or data structure will then be studied in terms of restrictions on what graphs are allowed in that representing class.

A directed graph G is an ordered pair (V,E) of nodes and arcs. The usual graphic notation will be used throughout. If there is an arc from x to y and an arc from y to x, then we will say there is an edge between x and y. Moreover,

$$x \rightleftarrows y$$

will be represented by

$$x \,\text{———}\, y \ .$$

A path from x to y is defined by any sequence of arcs from x_0 to x_1, x_1 to x_2, ..., x_{n-1} to x_n such that $x_0 = x$ and $x_n = y$.

We define a metric $d_G(x,y)$ on G as the number of arcs in the shortest such path. A binary tree is as defined in Knuth [7]. Note carefully that nodes in a binary tree are connected by edges so that the metric is symmetric. The relations son and ancestor defined as usual as is the function depth. If G is a binary tree, then a node x of G is a leaf of G if x has no sons.

The two classes of data structure we will be dealing with are arrays and ancestor trees.

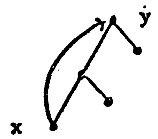
2.1 *Arrays*

G_n will denote the data structure corresponding to an $n \times n$ array, where $n \ge 1$. If the nodes of G_n are indexed by (i,j) where $1 \le i \le n$ and $1 \le j \le n$, there is assumed to be an edge between (i,j) and (i,j+1) and between (i,j) and (i+1,j). Thus, G_n is "rook connected." For instance, G_3 is

$$\begin{array}{ccccc} (1,1) & — & (1,2) & — & (1,3) \\ | & & | & & | \\ (2,1) & — & (2,2) & — & (2,3) \\ | & & | & & | \\ (3,1) & — & (3,2) & — & (3,3) \end{array}$$

2.2 *Ancestor Trees*

Ancestor trees are binary trees with an additional feature: a node x of an ancestor tree may be connected by an arc to any of its ancestors. For example,



is an ancestor tree; y is both an ancestor and a successor of x. Notice, however, that unlike a binary tree, the graph metric d_G is not necessarily symmetric on ancestor trees. Ancestor trees obviously include linear lists, circular lists, binary trees, and threaded lists (cf. Perlis and Thornton [12]) as special cases.

We will consider the following five control structures:

computed go to
go to with d-way branching
labelled exit
do forever
while .

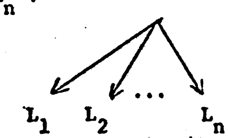
In addition all of the available classes will have access to a sequential flow-of-control and an alternative (e.g., if-then-else) flow of control. However, since the constructions described below do not involve schema manipulation, the details of these features need not be made explicit. We will now present the class of graphs which represent programs formed from each of the five control structures.

2.3 *Computed "go to" Programs*

$GOTO_\omega$ programs are programs which allow arbitrary branching between statements. For instance, we allow for representations of the construct

$$\underline{\text{go to}}\ i\ (L_1, \ldots, L_n)$$

which branches to the jth label depending on the value of i. Thus, this class of programs is represented by the entire class of directed graphs with no restrictions at all. Intuitively, the statement displayed above is represented by a node with n arcs leading to nodes labelled by $L_1, \ldots, L_n$:



2.4 *"go to" Programs with d-way Branching*

$GOTO_d$ programs are programs in which the amount of branching that is possible in one step is bounded by the integer d. For example, the FORTRAN construct

$$\text{IF (E) } L_1, L_2, L_3$$

falls in the class $GOTO_3$. Programs with d-way branching are represented by the class of directed graphs with maximum out degree† d.

†The out degree of a node is $|\{y : \exists \text{ an arc from } x \text{ to } y\}|$

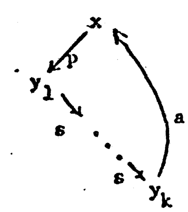
2.5 *While, do forever, label exit programs*

In this section while, do forever, and label exit programs will be defined. Each is defined as a certain class of ancestor trees. In order to define these classes we need the following relations which are defined for any ancestor tree:

1. $x \underset{p}{\rightarrow} y$ if y is the left immediate descendant of x.
2. $x \underset{s}{\rightarrow} y$ if y is the right immediate descendant of x.
3. $x \underset{a}{\rightarrow} y$ if y has an ancestor pointer from x.

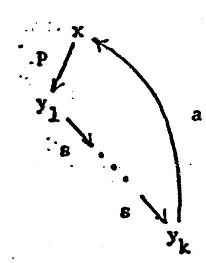
We view $x \underset{p}{\rightarrow} y$ as meaning that statement x can "push" into a substructure with first statement y, while we view $x \underset{s}{\rightarrow} y$ as meaning that statement x is "sequentially" followed by y. Finally, we view $x \underset{a}{\rightarrow} y$ as meaning that statement x can "exit" some structure and return to statement y. By placing restrictions on $\underset{p}{\rightarrow}$, $\underset{s}{\rightarrow}$, and $\underset{a}{\rightarrow}$ we will obtain the classes of programs while, do forever, and label exit.

A program is a while program provided it is an ancestor tree that satisfies: $y \underset{a}{\rightarrow} x$ implies $\exists y_1, \ldots, y_k$ such that



where $y_k = y$ is a leaf and no y_i for $i < k$ has an ancestor pointer. The last restriction of course reflects the fact that in a while loop only the last statement is allowed to exit the loop.

A program is a do forever program provided it is an ancestor tree that satisfies: $y \underset{a}{\rightarrow} x$ implies $\exists y_1, \ldots, y_k = y$ such that



where each y_i can only have ancestor pointers to x. The key distinction between while programs and do forever programs is that in a do forever program all statements in a loop can potentially exit immediately out of the looping structure. Clearly, do forever programs correspond to the BJ_n $(n \geq 1)$ structures of Bohm and Jacopini [2].

Finally, a label exit program is any program that is also an ancestor tree. Essentially label exit programs allow any jumping out of substructures as long as the return is always to ancestors. The class of label exit programs is therefore quite extensive and includes many types of so-called "structured" programs. For example, all label exit programs are reducible in the sense of [6]; moreover, they correspond essentially to programs in Engeler normal form [5].

3. *B, M Bounded Reductions.*

The following definition is fundamental to what follows. Let $G = (V,E)$ and $G^* = (V^*,E^*)$ be directed graphs with associated metrics d_G, d_{G^*}. Then we say that G^* can simulate G (or G can be reduced to G^*) with time constant B and space constant M,

$$G \le_{B,M} G^*$$

if there is a mapping (called an embedding) $\Phi: V^* \to V \cup \{\Lambda\}$ of the nodes of G^* to the nodes of G and a special node 'Λ', so that:

(1) $\forall v^* \in V^*$ with $\Phi(v^*) \neq \Lambda$

- $\forall w \in V$ such that there is an arc from $\Phi(v^*)$ to w

$\exists w^* \in V^*$ such that $\Phi(w^*) = w$ and

$$d_{G^*}(v^*,w^*) \le B\cdot d_G(\Phi(v^*),w);$$

(2) $\forall v \in V$

$$|\Phi^{-1}(v)| = |\{v^* \in V^*: \Phi(v^*) = v\}| \le M.$$

If Φ is an embedding and $\Phi(v^*) = \Lambda$, then we will sometimes refer to v^* as a bookkeeping node. If $\Phi(v^*) = v \neq \Lambda$, then v^* is said to be a copy of v.

Condition (1) states that when G and G^* are control structures (resp. data structures) simulation involves at most a B-fold increase in the cost of statement sequencing (resp. data element accessing); i.e., the embedding induces at most a B-fold increase in path length. Condition (2) states that there are at most M copies of any $v \in V$ in G^*. Note that although $G \le_{B,M} G^*$ may hold between data structures G and G^* when $M > 1$, it is unlikely that such a simulation would be of value (e.g., if an array is being stored as a list structure with multiple copies of array elements, then selective updating of the array may involve multiple updating of list nodes). Instead of $G \le_{B,1} G^*$ we will often write $G \le_B G^*$. For control structures, however, simulations with $M > 1$ are frequently used and are quite natural; this is sometimes called "node splitting."

In the following sections we will investigate uniform simulations, simulations in which B and M can be bounded over a class of graphs.

4. *Data Structure Embeddings*

In this section we will present our main result for data structures, settling negatively the question of whether arrays can be stored as arbitrary lists with linear bounds on proximity. This result generalizes a result of Rosenberg [14] on whether an array can be stored in linear memory without unbounded loss of proximity. However, since the arguments are fundamentally different, it is interesting to compare the two proofs. Recall that Rosenberg's arguments are essentially "volumetric:" a node in an array has at most $O(n^2)$ neighbors within distance n, while a linear list has at most $O(n)$ neighbors within distance n. A volumetric argument then demonstrates that arrays cannot be stored in a system with this neighborhood structure without unbounded loss of proximity. In contrast, these methods do not seem to apply to our problems; e.g., a node in a binary tree can have as many as $O(2^n)$ neighbors within distance n.

To obtain our result we will need a series of lemmas. Let $G = (V,E)$ be a directed graph with associated metric d_G and suppose $A \subseteq V$. We define the "boundary" of A as

$$\delta(A) = \{y \in A: \exists x \notin A \text{ s.t. } d_G(x,y) \le 1\}$$

In other words $\delta(A)$ is the set of vertices in A but reachable from some node not in A by an arc of G.

Lemma 4.1. Let $G_n = (V_n,E)$ be an n-by-n array and suppose that $A \subseteq V_n$ is such that $|A| \le n^2/2$. Then

$$|A| \le 2|\delta(A)|^2 .$$

Proof. Let $A = < A_1 \ldots A_n >$ be the columns of A. Define k to be the number of columns A_i with $|A_i| < n$. Since $|A| \le n^2/2$ it follows that $(n-k)n \le n^2/2$; hence, $k \ge n/2$. There are two cases.

I. No column has zero entries, i.e. for all i, $|A_i| > 0$. In this case $|\delta(A)| \ge k$: each column with at least 1 entry and less than $n-1$ entries contributes 1 to $\delta(A)$. Thus, $2|\delta(A)|^2 \ge n^2/2 \ge |A|$.

II. Some column has zero entries, i.e. say $|A_{i_0}| = 0$. In this case,

$$(1) \quad |\delta(A)| \ge \max_i |A_i| .$$

In order to see this let $|A_j|$ be maximum and assume that $i_0 < j$ (the case $j < i_0$ is similar). Select a row r. Then (r,i_0) has no entry, but (r, i_0+1) or ... or (r,j) has an entry. Thus, each row r contributes at least 1 to the $\delta(A)$. Clearly, we can assume that $|A_i| < n$ for all i; otherwise, the lemma is true. Now let $r_1, \ldots, r_k$ be the columns with $0 < |A_i| < n$. Now as in case I,

$$(2) \quad |\delta(A)| \ge k .$$

Putting (1) and (2) together yields,

(3) $2|\delta(A)| \ge k + \max_i r_i$.

Now $r_1 + \dots + r_k = |A|$ and $\max_i r_i \ge |A|/k$. Thus,

(4) $2|\delta(A)| \ge k + |A|/k$.

It follows that $|\delta(A)|^2 \ge |A|$. □

Lemma 4.2. Let $G_n = (V_n,E)$ and suppose $x,y \in V_n$; then $d_{G_n}(x,y) \le 2n$.

Proof. □

Lemmas 4.1 and 4.2 and the fact that $|V_n| = n^2$ summarize the basic properties of arrays that will be used in the proof of our main result.

Lemma 4.3. Let $T = (V,E)$ be an ancestor tree and let $T_o = (V_o,E_o)$ be a sub-tree of T. If $x \in V_o$ and $y \in V - V_o$, then $d_T(y,x) \ge$ depth of x in T_o.

Proof. Since $y \notin V_o$, any path from y to x must pass through the root of T_o. □

Lemma 4.4. Let $T^* = (V^*_o,E^*)$ be an ancestor tree; let $T^*_o = (V^*_o,E^*_o)$ be a subtree of T^*; and let $A = \phi(V^*_o) - \{\wedge\}$. Then if $G_n \le_B T^*$ and $|A| \le n^2/2$,

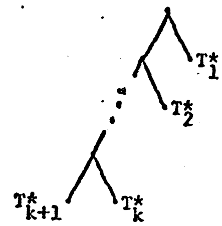
$$|A| \le 2^{2B+1}.$$

Proof. Assume that $|\delta(A)| > 2^B$. Since the root of T^*_o has at most 2^B descendants of depth less than $B + 1$, there is a node $x^* \in V^*_o$ of depth $\ge B + 1$ in T^*_o such that $\phi(x^*) \in \delta(A)$. Since $\phi(x^*) \in \delta(A)$, there is a $y \notin A$ with $d_{G_n}(y,\phi(x^*)) \le 1$. Now there exists a y^* such that $\phi(y^*) = y$ and $d_{T^*}(y^*,x^*) \le B$. Since $y \notin A$ it follows that $y^* \notin V^*_o$. But by Lemma 4.3, $d_{T^*}(y^*,x^*) \ge B+1$, which is a contradiction. Therefore, $|\delta(A)| \le 2^B$ and by Lemma 4.1,

$|A| \le 2|\delta(A)|^2 \le 2^{2B+1}$. □

Theorem 4.5. Let $T^* = (V^*,E^*)$ be an ancestor tree. If $G_n \le_B T^*$, then $B \ge O(\log n)$.

Proof. Assume $G_n \le_B T^*$ and for any subtree $T^*_i = (V^*_i,E^*_i)$ of T^* let $A_i = \phi(V^*_i) - \{\wedge\}$. Let T^*_1 and T^*_2 be subtrees of some node in T^*. Either $|A_1| \le n^2/2$ or $|A_2| \le n^2/2$, since ϕ is 1-1. Using this fact, we may assume that T^* is of the form



where $|A_i| \le n^2/2$ for $1 \le i \le k$ and we have suppressed explicit representation of ancestral links. Without loss of generality we assume always that the "smaller" subtree is on the right. By Lemma 4.4, $|A_i| \le 2^{2B+1}$ for all i.

Let i be the smallest integer such that $A_i \ne 0$ and let j be the largest such integer. Then

$|A| = \sum_{\ell=i}^{j} |A_\ell| \le (j-i+1)2^{2B+1}$.

Since $|A| = n^2$,

$(j-i+1)2^{2B+1} \ge n^2$. (1)

Now, let $x^* \in V^*_j$ and $Y^* \in V^*_i$. Then by Lemma 4.3, $d^*_T(y^*,x^*) \ge j - i$. On the other hand, by Lemma 4.2 $d_{G_n}(\phi(y^*),\phi(x^*)) \le 2n$; hence since $G_n \le_B T^*$,

$d_{T^*}(y^*,x^*) \le 2nB$. Thus,

$j - i \le 2nB$. (2)

Combining (1) and (2) we have that

$2nB + 1 \ge n^2/2^{2B+1}$.

It follows that $B \ge c_1 \log n + c_2$ for constants c_1, c_2. □

There are several ways to extend these results. First notice the extension to generalized ancestor trees with <u>bounded</u> branching is straightforward. Extending the result to more general data structures than arrays may also be interesting. Consider, for instance, the following restriction on the definition of array. Instead of <u>edges</u> between (i,j) and $(i,j+1)$ and (i,j) and $(i+1,j)$, let there be an <u>arc</u> directing (i,j) to $(i,j+1)$ and an <u>arc</u> directing (i,j) to $(i+1,j)$, reflecting, for example, a common accessing machanism. It can be shown that such one-way arrays are $\le_2$ embeddable in ancestor trees (moreover, this embedding exists for any acyclic graph) but at best $\le_{O(\log n)}$ embeddable in binary trees.

5. *Main Theorem*

Observe that the proof of Theorem 4.5 uses the $M = 1$

hypothesis at several key points. Since this restriction would be unrealistic in dealing with control structures, we will now remove it by generalizing the previous result.

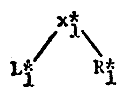
Theorem 5.1. Let $T^* = (V^*, E^*)$ be an ancestor tree and let $G_n \leq_{B,M} T^*$ where G_n is an n-by-n array. Then

$B + \log M \geq \log n + O(1)$.

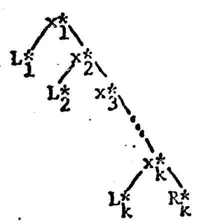
Proof. Let Φ be the embedding function and define another function Ψ with domain subsets of V^* by

$\Psi(A^*) = \Phi(A^*) - \{\Lambda\}$.

As in Theorem 4.5 we will decompose T^* as follows. Let x_1^* be the root of T^* and write T^* as

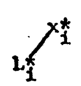


where we may assume $|\Psi(L_1^*)| \leq |\Psi(R_1^*)|$ without loss of generality. Clearly, this process may be iterated, letting x_{i+1}^* denote the root of R_i^* and expanding R_i^* at each stage of the construction. Thus, T^* can be written in the form



where $|\Psi(L_i^*)| \leq |\Psi(R_i^*)|$ for $1 \leq i \leq k$. Notice that we have ignored all ancestral links in this construction. Instead, we shall assume all such links exist but suppress explicit reference to them.

Let $T_i^* = (V_i^*, E_i^*)$ denote the subtree



and define T_i^* to be small if $|\Psi(V_i^*)| \leq n^2/4$; otherwise T_i^* is large. Define

$$B_k = \bigcup_{\substack{1 \leq i \leq k \\ T_i^* \text{ small}}} \Psi(V_i^*) .$$

(This notion of small is motivated by the key to the argument in Theorem 4.5.)

Lemma 5.2. For some k, $n^2/4 \leq |B_k| \leq n^2/2$.

Proof. Clearly by convention $|B_0| = 0$. If $|B_{k-1}| < n^2/4$ and $|B_k| \geq n^2/4$, then

$B_k = B_{k-1} \cup \Psi(V_k^*)$

where T_k^* is small, so that

$|B_k| \leq |B_{k-1}| + |\Psi(V_k^*)| \leq n^2/4 + n^2/4 = n^2/2.$

Thus, we need only show that $|B_k| \geq n^2/4$ for some k.

Assume, to the contrary, that $|B_k| < n^2/4$ for all k. Since $|\Psi(L_1^*)| \leq |\Psi(R_1^*)|$ and

$\Psi(T^*) = \Psi(L_1^*) \cup \{\Psi(x_1^*)\} \cup \Psi(R_1^*)$,

it follows that

$$\begin{aligned} n^2 = |\Psi(T^*)| &\leq |\Psi(L_1^*)| + 1 + |\Psi(R_1^*)| \\ &\leq 2|\Psi(R_1^*)| + 1; \end{aligned}$$

whence, $|\Psi(R_1^*)| \geq (n^2-1)/2 \geq n^2/4$. (Assume $n \geq 2$.) By taking k sufficiently large, we may assume that $|\Psi(R_k^*)| = 0$. Let i be the largest integer such that $|\Psi(R_i^*)| \geq n^2/4$: since $|\Psi(R_1^*)| \geq n^2/4$, i must exist. Then $|\Psi(R_j^*)| < n^2/4$ for $i < j \leq k$ and, since

$\Psi(T_j^*) = \Psi(L_j^*) \cup \{\Psi(x_j^*)\}$,

we have

$|\Psi(T_j^*)| \leq 1 + |\Psi(L_j^*)| \leq 1 + |\Psi(R_j^*)| < 1 + n^2/4.$

Thus, T_j^* is small for $i < j \leq k$. But this implies

$$\Psi(R_i^*) \subseteq \bigcup_{i<j\leq k} \Psi(V_j^*) \subseteq B_k$$

so that $n^2/4 \leq |\Psi(R_i^*)| \leq |B_k|$, which is a contradiction. □

We next need a variant of the concept of boundary. If A is a set of nodes of G_n, then define the adjacent boundary of A, $\tilde{\delta}(A)$, by

$$\tilde{\delta}(A) = \{y \notin A \mid \text{there exists } x \in A: d_{G_n}(x,y) = 1\}.$$

In other words, $\tilde{\delta}(A)$ is the set of nodes not in A reachable from some node in A in 1 step.

Lemma 5.3. Let A be a set of nodes of G_n with $|A| \leq n^2/2$. Then

$|A| \leq 2|\tilde{\delta}(A)|^2$.

Proof. The proof of this result is similar to that of lemma 4.1 and is omitted. □

Let k satisfy Lemma 5.2. By Lemma 5.3,

$|\tilde{\delta}(B_k)| \geq 1/\sqrt{2}\ |B_k|^{1/2} \geq n/2\sqrt{2}.$

Now let $\ell = |\{T_i^* \mid T_i^* \text{ is large}\}|$. Since at most M copies of any node in G_n appear in T^*,

$$\ell \cdot n^2/4 \le \sum_{\substack{1\le i\le k \\ T_i^* \text{ large}}} |\Psi(V_i^*)| \le Mn^2.$$

Hence $\ell \le 4M$.

In order to complete the proof our plan is as follows: We have already shown that $|\tilde{\delta}(B_k)| \ge n/2\sqrt{2}$; we next will show that this implies that there are an unbounded number of paths into the large trees T_i^* from the small trees; however, this is impossible. In order to carry out this plan let

$$S^* = \{V^* \mid \text{there exists large } T_i^*, \text{ there exists small } T_j^*, \text{ there exists } y^* \in V_i, \text{ there exists } x^* \in V_j, \text{ such that } d_{T^*}(x^*,y^*) \le B\}.$$

We will now define a 1-1 mapping g from $\tilde{\delta}(B_k)$ to S^*. Select some $y \in \tilde{\delta}(B_k)$. Then for some $x \in B_k$, $d_{G_n}(x,y) \le 1$ and $y \notin B_k$. Let x^* be a copy of x in some small T_i^*. Such a copy exists because

$$B_k = \bigcup_{\substack{1\le i\le k \\ T_i^* \text{ small}}} \Psi(V_i^*).$$

Since $G_n \le_{B,M} T^*$, there is a copy of y^* of y such that $d_{T^*}(x^*,y^*) \le B$. Now y^* is not in any small T_i^*, for $y \notin B_k$. Thus we can define $g(y) = y^*$ and g is indeed a mapping from $\tilde{\delta}(B_k)$ to S^*. In order to see that g is 1-1 we note that for any $y \in \tilde{\delta}(B_k)$,

$\Phi g(y) = \Phi(y^*) = y$;

hence, g is 1-1.

Finally, since h is 1-1, $|\tilde{\delta}(B_k)| \le |S^*|$. But it follows that $|S^*| \le \ell \cdot 2^B$, and $n/2\sqrt{2} \le 4M2^B$; hence, $B + \log M \ge \log n + O(1)$. □

As an application of Theorem 5.1 we present the following result. Say a goto program G has an B,M Engeler normal form if $G \le_{B,M} H$ for some H as an ancestor tree.

Corollary 5.3. (1) If G_n has a B,M Engeler normal form and B is fixed, then $M \ge O(n)$. (2) If G_n has a B,M Engeler normal form and M is fixed, then $B \ge O(\log n)$. Thus in worst case either time or space must be unbounded in the construction of Engeler normal forms.

An open but interesting question is what other schema constructions can be shown, as in Corollary 5.3, to be unbounded in time and space?

6. *Control Structures*

In this section, we state our main results for control structures using the relation $\le_{B,M}$. In particular, we will establish the hierarchy of figure 1. If X and Y are classes of control structures recall that $X \nrightarrow Y$ provided for no B and M and all $G \in X$ does there exist an $H \in Y$ such that $G \le_{B,M} H$. $Y \nrightarrow X$ we will write $X \geq Y$. It is, of course, the results of the form $X \nrightarrow Y$ that have the greatest novelty; the reader will notice that in contrast to previous results, the negative results here lie intermediate to

1. functional simulation: programs are functionally equivalent when they compute identical outputs for the same input
2. isomorphism or computational simulation: programs are computationally equivalent when the sequences of actions invoked are identical.

Only an indication is given for the proofs. For any directed graph G let

$N_{in}^G(\ell,x) = |\{y \mid y \xrightarrow{*} x \text{ in} \le \ell \text{ steps}\}|$

$N_{out}^G(\ell,x) = |\{y \mid x \xrightarrow{*} y \text{ in} \le \ell \text{ steps}\}|$

The key to the following proofs is:

Lemma 6.1. Suppose that $G \le_{B,M} H$. Then

1) $N_{out}^G(\ell,x) \le N_{out}^H(B\ell,x^*)$ for any x^* copy of x.
2) $N_{in}^G(B,X) \le M \cdot N_{in}^H(1,x^*)$ for some copy x^* of x.

Theorem 6.2. The following are true:

1) $\text{Goto}_w \nrightarrow \text{Goto}_d$ for any d.
2) $\text{Goto}_d \nrightarrow$ label exit.
3) Label exit $\nrightarrow$ do forever.
4) do forever $\nrightarrow$ while.

Proof Outline.

1) This follows from Lemma 6.1 part (1).
2) This is essentially theorem 5.1.
3) This follows by a careful analysis of both in and out degree of do forever's.
4) This follows from Lemma 6.1 part (2) since in degree of whiles is bounded and it is not in do forever. □

References

1. E. Ashcroft and Z. Manna. The translation of "goto" programs to "while" programs. Proc. IFIP

Congress 1971, 250-255.

2. C. Böhm and G. Jacopini. Flow-diagrams, Turing machines, and languages with only two formation rules. CACM 9:366-371, May 1966.

3. J. Bruno and K. Stieglitz. The expression of algorithms by charts. JACM 517-525, 1972.

4. O.-J. Dahl et al. Structured Programming.

5. E. Engeler. Structure and meanings of elementary programs. Symposium on Semantics of Algorithm Languages, 1971.

6. M. S. Hecht and J. D. Ullman. Characterizations of reducible flowgraphs. JACM 21, 1974.

7. D. E. Knuth. Fundamental Algorithms, The Art of Computer Programming Volume I. 1968.

8. D. E. Knuth. Structured programming with "goto" statements. Stanford Report CS-74-416.

9. D. E. Knuth and R. W. Floyd. Notes on avoiding "goto" statements. IPL Volume 1, 1971.

10. S. R. Kosaraju. Analysis of structured programs. Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, 1973.

11. R. J. Lipton. Limitations of synchronization primitives. Yale Computer Science Research Report #31. 1974.

12. A. J. Perlis and C. Thornton. Threaded trees. CACM 3, 1960.

13. W. W. Peterson et al. On the capabilities of <u>while</u>, <u>repeat</u>, and <u>exit</u> statements. CACM 16, 1973.

14. A. L. Rosenberg. Preserving proximity in arrays. IBM Report RC-4875, 1974.