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. DuMille
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 running 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. Thus, comparing this view will be a relation $S_{B,M}$ defined so that for data structures $G$ and $G^*$, $G^* \preceq B,M$ if $G$ can be embedded 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 [1,2,3,5,9,10]. Obviously, the fundamental issue is whether 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 $S_{B,M}$ for $M \geq 1$. If $G$ and $G^*$ are algorithms and have distinct sequencing mechanisms, then $G^* \preceq S_{B,M}G$ in 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 [12]).

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 $M$. Thus, we turn our attention to $S_{B,M}$ and the relations used by previous studies of control structure simulation: (1) $S_{B,M}$ is weaker than isomorphism since it allows both time and space to increase; (2) $S_{B,M}$ is stronger than functional or input-output equivalence; (3) $S_{B,M}$ 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 through the sequel are presented and the combinatorial models used for representing control structures and data structures are introduced. In Section 3 the relation $S_{B,M}$ 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-0597.
† 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-G-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.
\( G, G^a, \) if \( G^a \leq B, G^a \), then \( B \geq O(\log n) \) where \( n \)

is the number of components of \( G \). This result will also follow from our main theorem in Section 5, but since the \( \eta = 1 \) 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 \( \eta = 1 \). In this case, if \( G^a \leq B, G^a \) for certain natural choices of \( G, G^a \) it must be that \( B = \log n = O(1) \). 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 \( S_{B, M} \) 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 \( G_{S_{B, M}} \leq H \) 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 \( G_{S_{B, M}} \leq H \) 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 4, 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, \( X + Y \) means that for no \( B \) and \( M \) does every \( G \) in \( X \) have an \( H \) counterpart in \( Y \) such that \( G_{S_{B, M}} \leq H \).

The key result here is:

\[ \text{goto} \rightarrow \text{label exit} \]

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
\[ \downarrow \]
\[ \text{goto with d way branching (d \geq 2)} \]
\[ \downarrow \]
\[ \text{label exits} \]
\[ \downarrow \]
\[ \text{do forever} \]
\[ \downarrow \]
\[ \text{while} \]

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 accessible elements. Thus, in either case, what is to be modeled 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 accessible 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 modeled 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 \textit{edge} between \( x \) and \( y \). Moreover,

\[ x \xrightarrow{\text{edge}} y \]

will be represented by

\[ x \xrightarrow{\text{arc}} y \]

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 
\text{edges} so that the metric is symmetric. The relations \textit{son} and \textit{ancestor} defined as usual as is the function depth. If \( G \) is a binary tree, then a node \( x \) of \( G \) is a \textit{leaf} of \( G \) if \( x \) has no sons.

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

2.1 Arrays

G will denote the data structure corresponding to an \( n \times n \) array, where \( n \geq 1 \). If the nodes of \( G \) are indexed by \((i,j)\) where \( 1 \leq i \leq n \)

\[ 1 \leq j \leq n, \]

there is assumed to be an edge between \((i,j)\) and \((1,j+1)\) and between \((i,j)\) and \((i+1,j)\). Thus, \( G \) is "room connected." For instance, \( G \) is

\[ \begin{align*}
(1,1) & \rightarrow (1,2) \rightarrow (1,3) \\
(2,1) & \rightarrow (2,2) \rightarrow (2,3) \\
(3,1) & \rightarrow (3,2) \rightarrow (3,3)
\end{align*} \]

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 \textit{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 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 al-
ternative (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_d programs are programs which allow
arbitrary branching between statements. For in-
stance, we allow for representations of the constuct

    Go to i (L_1, ..., L_n)

which branches to the i-th 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, ..., 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

    IF (E) L_1, L_2, L_3

falls in the class GOTO_d. Programs with d-way
branching are represented by the class of directed
trees which have maximum out degree d.

The out degree of a node is
|{y: E an arc from x 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 \rightarrow y if y is the left immediate descen-
dant of x.

2. x \leftarrow y if y is the right immediate
descendant of x.

3. x \leftrightarrow y if y has an ancestor pointer
   from x.

We view x \rightarrow y as meaning that statement x can
"push" into a substructure with first statement y,
while we view x \leftarrow y as meaning that statement x
is "sequentially" followed by y. Finally, we view
x \leftrightarrow y as meaning that statement x can "exit" some
structure and return to statement y. By placing
restrictions on \rightarrow, \leftarrow, and \leftrightarrow 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 \leftrightarrow x
implies \exists y_1, ..., y_k such that

where y_k = y is a leaf and no y_i for 1 < 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 \rightarrow x
implies \exists y_1, ..., 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 pro-
gram all statements in a loop can potentially exit
immediately out of the looping structure. Clearly,
do forever programs correspond to the Bj (n ≥ 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 dG and dG*. Then we say that G* can simulate G (or G can be reduced to G*) with time constant B and space constant M,

\[ G \leq_{BM} G^* \]

if there is a mapping (called an embedding)

ϕ: V* → V \cup \{ λ \} of the nodes of G* to the nodes of G and a special node 'λ', so that:

1. \( \forall \psi \in V^* \) with \( ϕ(\psi) \neq \lambda \)
   - \( \forall v \in V \) such that there is an arc from \( ϕ(\psi) \) to \( v \)
   - \( \exists \psi \in V^* \) such that \( ϕ(\psi) = v \) and \( d_{G*}(\psi,\psi) \leq B \cdot d_G(ϕ(\psi),v) \);

2. \( \forall v \in V \)
   - \( |ϕ^{-1}(v)| = |\{ ϕ(\psi) : ϕ(\psi) = v \}| \leq M. \)

If ϕ is an embedding and \( ϕ(v) = λ \), then we will sometimes refer to v as a bookkeeping node. If \( ϕ(v) = v \neq λ \), 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 ∈ V in G*.

Note that although \( G \leq_{BM} G^* \) 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 \( G \leq_{BM} G^* \) we will often write \( G \leq_{B,M} G^* \).

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 \( 0(\eta^2) \) neighbors within distance η, while a linear list has at most \( 0(\eta) \) neighbors within distance η. 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 \( 0(\log n) \) neighbors within distance η.

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

\[ \delta(A) = \{ y ∈ A : ∃ x / A s.t. d_G(x,y) ≤ 1 \} \]

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 \( G_n = (V_n,E_n) \) be an n-by-n array and suppose that \( A \subseteq V_n \) is such that \( |A| ≤ n^2/2. \) Then

\[ |A| ≥ 2|\delta(A)|^2. \]

Proof. Let \( A = \langle A_1^{n} \rangle \) be the columns of A. Define k to be the number of columns \( A_i \) with \( |A_i| < n. \) Since \( |A| ≤ n^2/2 \) it follows that \( (n-k)n ≤ n^2/2; \) hence, \( k ≥ n/2. \) There are two cases.

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

II. Some column has zero entries, i.e. say \( |A_0| = 0. \) In this case,

\[ (1) \quad |\delta(A)| ≥ \max |A_i|. \]

In order to see this let \( |A_j| \) be maximum and assume that \( i_0 < j \) (the case \( i_0 < j \) 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 contributes at least 1 to the \( \delta(A). \) Clearly, we can assume that \( |A_j| < n \) 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,

\[ (2) \quad |\delta(A)| ≥ k. \]

Putting (1) and (2) together yields,
(2) $2|\delta(A)| \geq k + \max_i r_i.$

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

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

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

Lemma 4.2. Let $G_n = (V_n, E)$ and suppose $x, y \in V_n$; then $d_G(x, y) \leq 2n.$

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

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

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

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

$|A| \leq 2^{2n}.$

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 $\geq B + 1$ in $T^*_o$ such that $\phi(x^*) \in \delta(A).$ Since $\phi(x^*) \in \phi(A),$ there is a $y \notin A$ with $d_{G_n}(y, \phi(x^*)) \leq 1.$ Now there exists a $y^*$ such that $\phi(y^*) = y$ and $d_{T^*}(y^*, x^*) \leq B.$ Since $y^* \notin A$ it follows that $y^* \in V^*_o.$ But by Lemma 4.3, $d_{T^*}(y^*, x^*) \geq B + 1,$ which is a contradiction.

Therefore, $|\delta(A)| \leq 2^B$ and by Lemma 4.1,

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

Theorem 4.5. Let $T^* = (V^*, E^*)$ be an ancestor tree.

If $G_n \leq_B T^*$, then $B \geq O(\log n).$

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

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

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

$|A| = \frac{j}{\Delta} \leq (j-i+1)2^{2n}.$

Since $|A| = n^2,$

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

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

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

$j - i \geq 2nB.$

Combining (1) and (2) we have that

$2nB + 1 \geq n^2/2.$

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

There are several ways to extend these results. First notice the extension to generalized ancestor trees with bounded 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 edges between $(i, j)$ and $(i, i+1)$ and $(j, j)$ and $(i+1, j)$, let there be an arc directing $(i, j)$ to $(i, i+1)$ and an arc directing $(j, j)$ to $(i+1, j)$, reflecting, for example, a common accessing mechanism. It can be shown that such one-way arrays are $\epsilon_2$ embeddable in ancestor trees (moreover, this embedding exists for any acyclic graph) but at best $\epsilon_0(\log n)$ 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 could 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 \( C_n \subseteq B_n \) \( T^* \) where \( C_n \) is an \( n \)-by-\( n \) array.
Then
\[ B + \log n \geq \log n + O(1). \]

Proof. Let \( \psi \) be the embedding function and define
another function \( \psi \) with domain subsets of \( V^* \) by
\[ \psi(A) = \psi(\Lambda) - \{A\}. \]
As in Theorem 4.5 we will decompose \( T^* \) as follows.
Let \( x_1^* \) be the root of \( T^* \) and write \( T^* \) as

\[ L_1^* \hskip 1cm x_1^* \hskip 1cm R_1^* \]

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

\[ L_1 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_k \rightarrow L_k \rightarrow x_k \rightarrow \cdots \rightarrow x_{k-1} \rightarrow L_{k-1} \rightarrow x_{k-1} \rightarrow \cdots \rightarrow x_1 \rightarrow x_1 \]

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 \( T_1^* = (V_1^*, E_1^*) \) denote the subtree

\[ L_1 \rightarrow x_1 \rightarrow \]

and define \( T_1^* \) to be small if \( |\psi(V_1^*)| \leq n^{2/4} \);
otherwise \( T_1^* \) is large. Define
\[ B_k = \psi(V_1^*) \cup \cdots \cup \psi(V_k^*) \]
(Cf. Lemma 5.2.)

Lemma 5.2. For some \( k \), \( n^{2/4} \leq |B_k| \leq n^{2/2} \).
Proof. Clearly by convention \( |B_0| = 0 \). If

\[ |B_{k-1}| < n^{2/4} \text{ and } |B_k| \geq n^{2/4}, \text{ then} \]
\[ B_k = B_{k-1} \cup \psi(V_k^*) \]
where \( T_k^* \) is small, so that
\[ |B_k| < |B_{k-1}| + |\psi(V_k^*)| \leq n^{2/4} + n^{2/4} = n^2/2. \]
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 \( |\psi(L_k^*)| \leq |\psi(R_1^*)| \) and
\[ \psi(T_k^*) = \psi(L_k^*) \cup \psi(x_k^*), \]
it follows that
\[ n^2 = |\psi(T_1^*)| \leq |\psi(L_1^*)| + 1 + |\psi(R_1^*)| \]
\[ \leq 2|\psi(R_1^*)| + 1; \]
whence, \( |\psi(R_1^*)| > (n^2-1)/2 \geq n^2/4 \). (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 \( |\psi(R_i^*)| > n^2/4 \); since \( |\psi(R_i^*)| > n^2/4 \), \( i \) must
exist. Then \( |\psi(R_i^*)| < n^2/4 \) for \( i < j \leq k \) and, since
\[ \psi(T_j^*) = \psi(L_j^*) \cup \psi(x_j^*), \]
we have
\[ |\psi(T_j^*)| \leq 1 + |\psi(L_j^*)| \leq 1 + |\psi(R_j^*)| \leq 1 + n^2/4. \]
Thus, \( T_j^* \) is small for \( i < j \leq k \). But this implies
\[ \psi(R_k^*) \subseteq \cup_{i < j \leq k} \psi(V_j^*), \]
so that \( n^2/4 \leq |\psi(R_k^*)| \leq |B_k| \), which is a
contradiction. \( \square \)

We next need a variant of the concept of
boundary. If \( A \) is a set of nodes of \( C_n \), then
define the adjacent boundary of \( A \), \( \delta(A) \), by
\[ \delta(A) = \{ y \in A \mid \text{there exists } x \in A; \delta_{C_n}(x, y) = 1 \}. \]
In other words, \( \delta(A) \) 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 \( C_n \) with
\( |A| \leq n^2/2 \). Then
\[ |A| = \leq 2|\delta(A)|^2. \]
Proof. The proof of this result is similar to that
of Lemma 4.1 and is omitted. \( \square \)

Let \( k \) satisfy Lemma 5.2. By Lemma 4.3,
\[ |\delta(B_k)| \geq 1/\sqrt{2} |B_k|^{1/2} \geq n^{1/2}. \]
Now let \( i = \{ T_1^* \mid T_1^* \text{ is large} \} \). Since at most \( M \)
copies of any node in \( C_n \) appear in \( T^* \),
5. $\psi$ to be unbounded in time and space?
6. Control Structures

In this section, we state our main results for control structures using the relation $\leq_{b,n}$. In particular, we will establish the hierarchy of figure 1. If $X$ and $Y$ are classes of control structures recall that $X \leq Y$ provided for no $B$ and $M$ and all $C \in X$ does there exist an $H \in Y$ such that $C \leq_{b,n} H$. $Y \not\leq X$ we will write $X > Y$. It is, of course, the results of the form $X > Y$ 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

\[ N_{in}^G(t,x) = \{ y \mid y \not\geq x \text{ in } t \text{ steps}\} \]
\[ N_{out}^G(t,x) = \{ y \mid x \not\geq y \text{ in } t \text{ steps}\} \]

The key to the following proofs is:

**Lemma 6.1.** Suppose that $C \leq_{b,n} H$. Then

1) $N_{out}^G(t,x) \subseteq N_{out}^H(1,x)$ for any $x \not\geq y$ of $x$. 
2) $N_{in}^G(t,x) \subseteq N_{in}^H(1,x)$ for some copy of $x$.

**Theorem 6.2.** The following are true:

1) $\text{Goto}_e \rightarrow \text{Goto}_d$ for any $d$.
2) $\text{Goto}_d \rightarrow \text{label exit}$.
3) Label exit $\rightarrow$ do forever.
4) do forever $\rightarrow$ 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 while is bounded and it is not in do forever. 

**References**

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


