Abstract

The reachability problem for vector addition systems is shown to require at least exponential space. In addition, the boundness problem for vector addition systems is also shown to require exponential space.

Keywords

vector addition system, reachability, boundness, space-complexity, exponential space, parallel programs, intractable problem

The Reachability Problem Requires Exponential Space

Richard J. Lipton

Research Report #~~62~~ 63

January 1976

An earlier version of this paper was presented at the Conference on Petri Nets and Related Methods, July 1975. The research was supported in part by the National Science Foundation under grant DCR-74-12870.

1. Introduction

Since Karp and Miller [4] introduced the concept of a *vector addition system*, considerable interest has been directed toward the analysis and understanding of these systems. Indeed, much of the attention has been directed toward understanding what questions about vector addition systems are decidable. Three basic questions have been studied. Karp and Miller were able to show that the question of whether a vector addition system is bounded [4] is decidable. On the other hand, Rabin [9] showed that the question of whether one vector addition system is included in another is undecidable. This was later refined by Hack [1], who showed that the question of whether one vector addition system is equivalent to another is undecidable. The third question is the so-called reachability problem for vector addition systems [4].

Its importance has been demonstrated by the number of results that show that many other question are equivalent to it (Hack [1], Keller [5]). In contrast to the first two problems, the reachability problem has remained elusive. Currently it is not known whether this problem is decidable. A number of researchers have, however, attempted recently to determine the "space" complexity of this problem. Jones and Lien [2] and Landweber [6] have independently observed that the reachability problem is at least PSPACE-hard [8], i.e. the reachability problem is at least as hard as the parsing of context-sensitive languages. And Meyer has independently observed that the reachability problem restricted to reversible vector addition systems is also PSPACE-hard [7].

While these results supply evidence that the reachability problem is intractable, they fall short of proving it. Here we will present a proof that the reachability problem is indeed intractable. More precisely, we will show that

Theorem 1: The reachability problem for vector addition systems requires at least 2^{cn} space infinitely often for some constant $c > 0$.

The proof of this result is based on the ability to construct vector addition systems that are able assuredly to count to 2^{2^n}. It is interesting to note that some of the folklore surrounding the reachability problem bounted whether this was indeed possible. The main techniques that make the construction of these systems possible are both new. One is essentially a subroutine-calling mechanism. That is, we show how vector addition systems can implement a primitive type of parameter passing and subroutine calling. The other is the introduction of the concept of a "strong computer." The strong computer behaves quite differently from the "weak computer" of Rabin [9]. This difference is reflected in the fact that we can construct vector addition systems that can *guarantee* that they count to 2^{2^n}.

The remainder of this paper is devoted to proving Theorem 1 and its corollary. In section 2 it is shown that the reachability problem for vector addition systems is reducible to the acceptance problem for "parallel programs." Since the proof of Theorem 1 involves a good deal of programming, it is more convenient to work with parallel programs than with vector addition systems. In section 3 the concept of a strong computer is presented. In section 4 the main lemma is proved. This lemma shows how to construct, recursively, parallel programs that can simulate 2^{2^n} counters, i.e. counters capable of counting from 0 to 2^{2^n}. Section 5 completes the proof of Theorem 1 and its corollary.

2. Vector Addition Systems and Parallel Programs

In this section both vector addition systems and parallel programs are presented. It is then shown in Lemma 2 that the reachability problem for vector addition systems is reducible in the sense of Meyer and Stockmeyer [8] to the "acceptance" problem for parallel programs.

Definition: $\langle S,F,\{V_1,\ldots,V_n\}\rangle$ is a *vector addition system* of dimension k provided $S,F \in \mathbf{N}^k$ and $V_1,\ldots,V_n \in \mathbf{Z}^k$ where $\mathbf{N}$ is the set of natural numbers and $\mathbf{Z}$ is the set of integers. The *reachability problem* is the question whether there exist vectors $W_1,\ldots,W_m \in \mathbf{N}^k$ such that

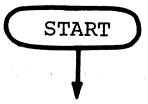
1) $S = W_1$;

2) $F = W_m$;

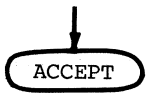
3) for each i, $W_{i+1} = W_i + V_j$ for some j.

We now turn to the definition of parallel programs. A *parallel program* is a collection of *flowcharts*. Each flowchart is a finite directed graph whose nodes are statements of the following form:

1) Start:



2) Accept:



3) Guess:

4) Assignment:

$$\underset{\sim}{x} \leftarrow \underset{\sim}{x}_1 + c_1 \cdots \underset{\sim}{x}_n \leftarrow \underset{\sim}{x}_n + c_n$$

where each c_i is 0, 1, or -1.

Informally, the semantics of these programs is as follows: The variables $\underset{\sim}{x}_1,\ldots,\underset{\sim}{x}_n$ are initially 0; they take on values in $\mathbf{N}$. The start statement is the first to be executed in any flowchart. Statements then execute one at a time until (if ever) some

accept statement is executed. The guess statement causes a nondeterministic branch to occur. The assignment can execute only in the case that $\underset{\sim}{x}_i + c_i \geq 0$ for all i.

The precise semantics of these programs is as follows: The state of a parallel program is of the form

$$\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle.$$

The $x_1,\ldots,x_n$ are the values of the variables $\underset{\sim}{x}_1,\ldots,\underset{\sim}{x}_n$; the $P_1,\ldots,P_m$ are the values of the *program counters*. There is one program counter for each flowchart; the values of the *ith* program counter are the statements of the *ith* flowchart. The initial state of a parallel program has each $x_i = 0$ and each P_i = the start statement of the *ith* flowchart. Now define the yields relation between states by

$$\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle \vdash \langle P_1',\ldots,P_m',x_1',\ldots,x_n'\rangle$$

provided one of the following is true:

1) P_i is a start statement and P_i' = the *unique* successor of this statement and $P_j = P_j'$ for all $j \neq i$ and $x_j = x_j'$ for all j.
2) P_i is a guess statement and P_i' = *one* of the successors of this statement and $P_j = P_j'$ for all $j \neq i$ and $x_j = x_j'$ for all j.
3) P_j is an assignment statement and $x_i + c_i \geq 0$ for all i and P_j' = the *unique* successor of this statement and $P_j = P_j'$ for all $j \neq i$ and $x_j' = x_j + c_j$ for all j.

The acceptance problem for a parallel program is the question whether there is some state

$$S = \langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle$$

with P_i = an accept statement for some i such that $S_0 \vdash^* S$ where S_0 is the initial state and $\vdash^*$ is the transitive closure of $\vdash$. We also say that S is a *reachable state* provided $S_0 \vdash^* S$.

A number of remarks about parallel programs are in order. Essentially, parallel

programs can do only two things, either perform an assignment or execute a nondeterministic branch. They cannot directly test any predicate. But as we will see later in Lemma 3, they can exploit the fact that the assignment statement is "partial" to simulate the testing of certain predicates.

Since we plan next to discuss the space complexity of the reachability problem and the acceptance problem, we need several basic concepts from automata-based complexity (Meyer and Stockmeter [8], Stockmeyer [10]). As usual, a language L is a subset of $\{0,1\}^*$, i.e. it is a set of strings over the alphabet 0,1. Say a language L_1 is *polynomial time reducible* to another language L_2 provided there is some function f computable in polynomial time by a deterministic multitape Turing machine such that, for all strings x, $x \in L_1$ if and only if $f(x) \in L_2$. A language L requires *exponential space* provided any deterministic Turing machine that accepts L requires for some $c > 0$ at least 2^{cn} space infinitely often on inputs of length n. A basic result of Meyer and Stockmeyer's is: If L_1 is polynomial time reducible to L_2 and L_1 requires exponential space, then L_2 requires exponential space.

In order to state the next result, we must agree on some particular encoding for both the acceptance problem and the reachability problem as languages. As in Karp [3], we can use any of a number of encodings; we will assume that some encoding is fixed throughout the rest of the paper.

Lemma 2: The acceptance problem for parallel programs is polynomial time reducible to the reachability problem for vector addition systems.

Proof: We omit a detailed proof of this lemma. It should, however, be clear that parallel programs can be encoded as vector addition systems. □

The consequence of this lemma is that we need only show that the acceptance problem

requires at least exponential space in order to prove Theorem 1.

3. Strong Computers

In this section, the concept of a strong computer is presented. Before doing this we need some additional concepts. A set of statements P of a parallel program is a *block with entry and exits* (entry = S_1; exits = S_2 and S_3) provided

1) only S_1 has a predecessor not in P,

2) only S_2 and S_3 have successors not in P,

3) S_2 and S_3 have unique successors.

If S is a state of some parallel program, then we will say that in S *control is at* statement S provided the program counter P_i = S where S is in the *ith* process. Also say that $S_1 \ |\overline{P}^* \ S_2$ provided there is a computation from state S_1 to state S_2 with only statements from P executing.

We are now ready to define the concept of a strong computer.

Definition: Suppose that P is a block with entry S_1 and exits S_2 and S_3 (S_2 = YES exit, S_3 = NO exit) in some parallel program. Then P is a *strong computer* for the predicate $Q(x_1,\ldots,x_n)$ and the side effect $\underset{\sim}{x}_r \leftrightarrow \underset{\sim}{x}_s$ provided the following are true:

(Let P be in the k*th* process.)

1) If $\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle$ is a reachable state with control at the entry of P, then

   a) If not $Q(x_1,\ldots,x_n)$, then there is a state S such that $\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle \ |\overline{P}^* \ S$ and in S control is at the NO exit of P.

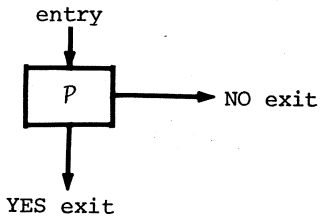
   b) If $Q(x_1,\ldots,x_n)$, then there is a state S such that $\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle \ |\overline{P}^* \ S$ and in S control is at the YES exit of P.

2) If $\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle$ is a reachable state with control at the entry of $\mathcal{P}$ and $\langle P_1,\ldots,P_m,x_1,\ldots,x_n\rangle \vdash^* \langle P'_1,\ldots,P'_m,x'_1,\ldots,x'_n\rangle$ where in $\langle P'_1,\ldots,P'_m,x'_1,\ldots,x'_n\rangle$ control is at one of the exits of $\mathcal{P}$, then the following are true:

   a) If control is at the NO exit, then not $Q(x_1,\ldots,x_n)$ and $x_i = x'_i$ for $i = 1,\ldots,n$ and $P_i = P'_i$ for $i = 1,\ldots,m$ except for $i = k$.

   b) If control is at the YES exit, then $Q(x_1,\ldots,x_n)$ and $x_i = x'_i$ for $i = 1,\ldots,n$ except that $x_r = x'_s$ and $x_s = x'_r$ (i.e. $\underset{\sim}{x}_r$ and $\underset{\sim}{x}_s$ are interchanged) and $P_i = P'_i$ for $i = 1,\ldots,m$ except for $i = k$.

A strong computer $\mathcal{P}$ for the predicate $Q(x_1,\ldots,x_n)$ and the side effect $\underset{\sim}{x}_r \leftrightarrow \underset{\sim}{x}_s$ can be viewed as follows:



The four conditions that $\mathcal{P}$ must satisfy are:

1a) If not $Q(x_1,\ldots,x_n)$, then there is some computation from the entry to the NO exit.

1b) If $Q(x_1,\ldots,x_n)$, then there is some computation from the entry to the YES exit.

2a) If there is a computation from entry to the NO exit, then $Q(x_1,\ldots,x_n)$ was false and no side effects have occurred.

2b) If there is a computation from entry to the YES exit, then $Q(x_1,\ldots,x_n)$ was true and the only side effect is that $\underset{\sim}{x}_r$ and $\underset{\sim}{x}_s$ are interchanged.

4. Basic Lemma

In this section the key lemma to our proof is presented.

Definition: $A_1 = 2$ and $A_{k+1} = A_k^2$.

Clearly $A_i = 2^{2^{i-1}}$.

Lemma 3: Suppose that

$$\underset{\sim}{x}_1\ \underset{\sim}{x}'_1\ \underset{\sim}{y}_1\ \underset{\sim}{y}'_1\ \underset{\sim}{S}_1\ \underset{\sim}{S}'_1\ \underset{\sim}{B}_1\ \underset{\sim}{C}_1\ \cdots\ \underset{\sim}{x}_k\ \underset{\sim}{x}'_k\ \underset{\sim}{y}_k\ \underset{\sim}{y}'_k\ \underset{\sim}{S}_k\ \underset{\sim}{S}'_k\ \underset{\sim}{B}_k\ \underset{\sim}{C}_k\ \underset{\sim}{r}\ \underset{\sim}{r}'$$

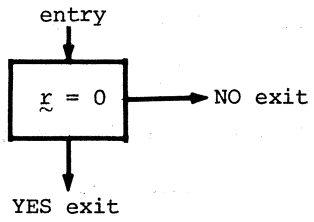
are variables such that

1) $r + r' = A_{k+1}$.

2) $x_i = y_i = S_i = 0$ and $x'_i = y'_i = S'_i = A_i$ for all $i = 1,\ldots,k$.

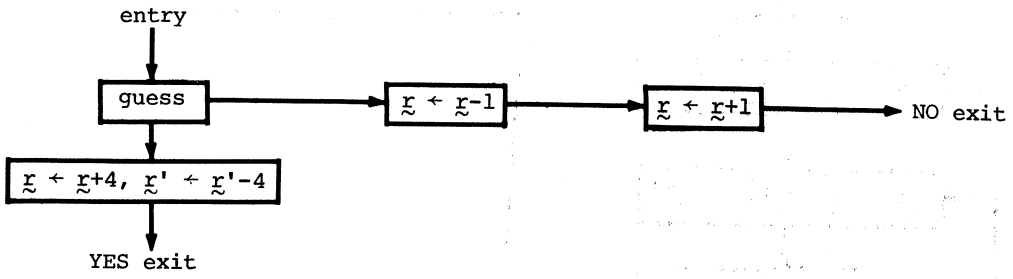
3) $B_i = C_i = 0$ for all $i = 1,\ldots,k$.

Then there is a strong computer P for the predicate $\underset{\sim}{r} = 0$ and with the side effect $\underset{\sim}{r} \leftrightarrow \underset{\sim}{r}'$. Moreover, P contains at most εk statements where ε is a constant independent of k. Thus P informally behaves as a decision statement:



The NO exit has no side effects, while the YES branch has the side effect of swapping the values of $\underset{\sim}{r}$ and $\underset{\sim}{r}'$.

Proof: We proceed by induction on k. For k = 1 the desired program is the following:

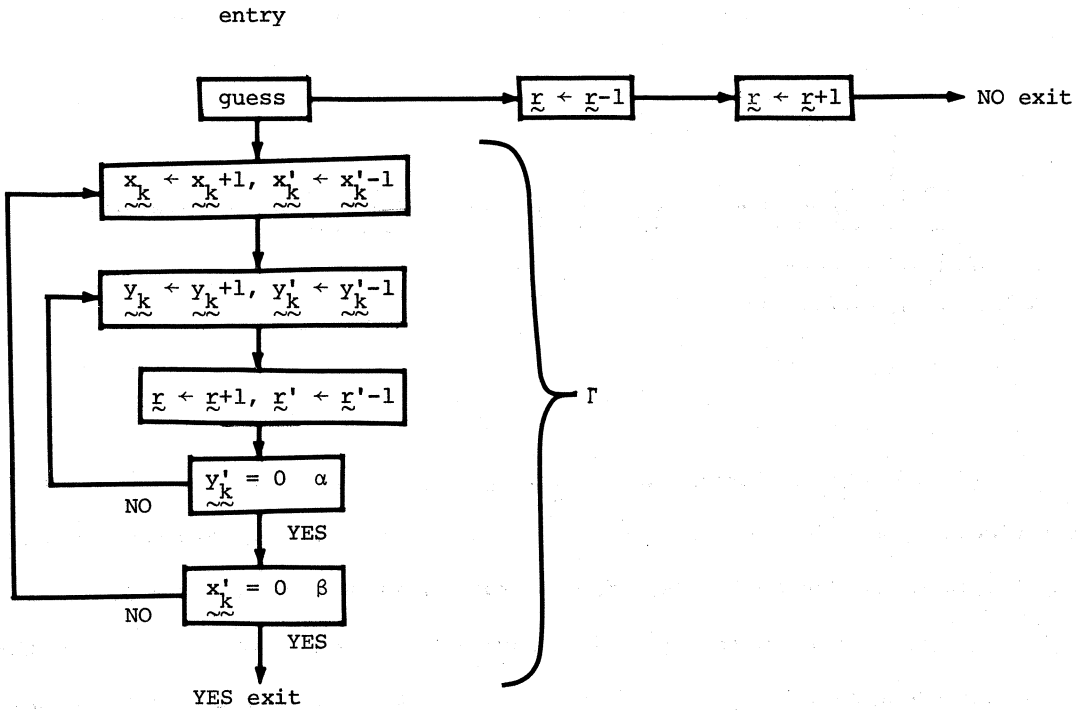


We need only check the four conditions that a strong computer must satisfy. If $\underset{\sim}{r} > 0$ then control can clearly reach the NO exit; moreover, control can reach the NO exit only in the case that $\underset{\sim}{r} \geq 1$ is true. Thus (1a) and (2a) are true. Next suppose that $\underset{\sim}{r} = 0$. Then since $\underset{\sim}{r} + \underset{\sim}{r}' = A_2 = 4$, it follows that $\underset{\sim}{r}' = 4$. Therefore if $\underset{\sim}{r} = 0$, control can reach the YES exit. On the other hand, if control does reach the YES exit, then $\underset{\sim}{r}' \geq 4$ was true. Since $\underset{\sim}{r} + \underset{\sim}{r}' = 4$, this implies that $\underset{\sim}{r} = 0$ was true. Thus (1b) and (2b) are true once one realizes that the effect of control's reaching the YES exit is to map $(\underset{\sim}{r},\underset{\sim}{r}')$ to $(\underset{\sim}{r}+4,\underset{\sim}{r}'-4)$.

Before going on to the case k > 1 it is perhaps important to point out that the use of $\underset{\sim}{r}$ and $\underset{\sim}{r}'$ to complement each other is one of the keys to our proof. We have replaced the test $\underset{\sim}{r} = 0$ by the test $\underset{\sim}{r}' \geq 4$.

Now assume that k > 1. Then the desired program P is given on page 10.

For the moment, let us assume that the statements labelled α and β are strong computers that have been recursively constructed. Then we assert that the program P is indeed a strong computer for the predicate $\underset{\sim}{r} = 0$ and the side effect $\underset{\sim}{r} \leftrightarrow \underset{\sim}{r}'$. As before, if $\underset{\sim}{r} > 0$, then control can reach the NO exit. Also if control does reach the NO exit, then $\underset{\sim}{r} > 0$ was true and no side effect has occurred. Thus (1a) and (2a) are true. Now suppose that $\underset{\sim}{r} = 0$. Then $\underset{\sim}{r}' = A_{k+1}$. The effect of the section of the program labelled is to add 1 to $\underset{\sim}{r}$ A_k^2 times and subtract 1 from $\underset{\sim}{r}'$ A_k^2 times. Thus since $\underset{\sim}{r} = A_k^2$, control can reach the YES exit. Note carefully how the side effects are used to reset the values of

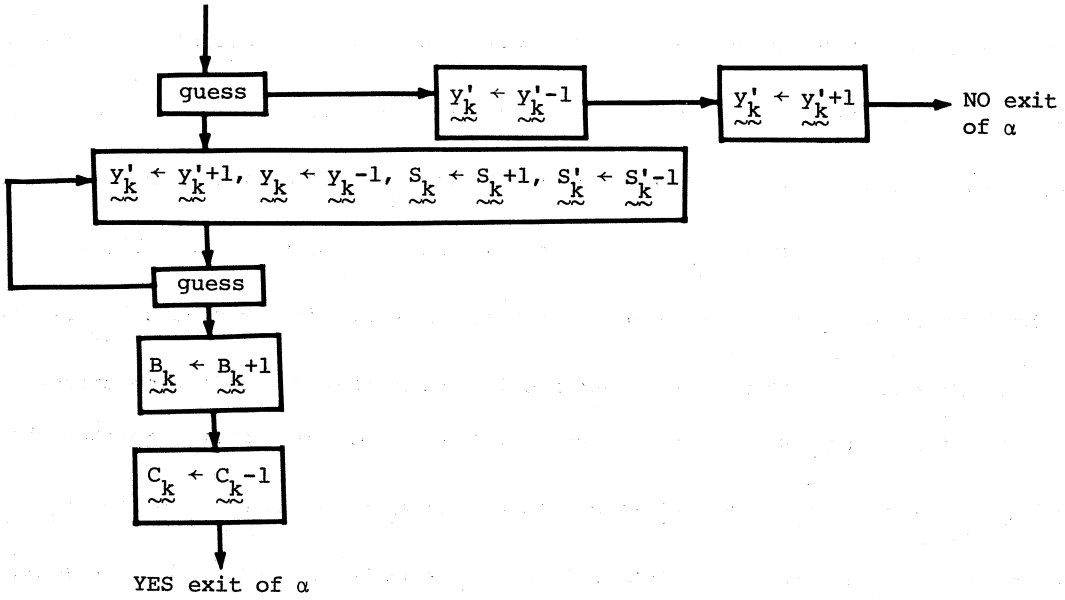


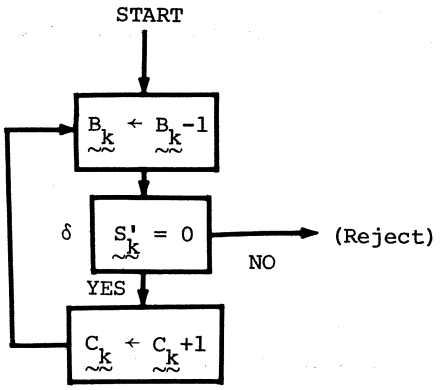
---

of $\underset{\sim\sim}{x}_k$, $\underset{\sim\sim}{x}'_k$, $\underset{\sim\sim}{y}_k$, and $\underset{\sim\sim}{y}'_k$. For example, when the statement α goes to its YES exit the values of $\underset{\sim\sim}{y}_k$ and $\underset{\sim\sim}{y}'_k$ are "reset" to 0 and A_k respectively. On the other hand, if control reaches the YES exit of P, then the reasoning above shows that $\underset{\sim}{r}' \geq A_k^2$. Since $\underset{\sim}{r} + \underset{\sim}{r}' = A_{k+1} = A_k^2$, it follows that $\underset{\sim}{r}$ was 0. Note also that the side effect $\underset{\sim}{r} \leftrightarrow \underset{\sim}{r}'$ is achieved. Thus (1b) and (2b) are true and P is a strong computer.

The problem is that defining P in this way would lead to a strong computer with 2^k statements. Thus statements α and β are *not done recursively*. We use the following "subroutine trick" to avoid this problem. α is shown on page 11. β is the same except that the roles of $\underset{\sim}{x}$ and $\underset{\sim}{y}$ are interchanged. We also include a new flowchart as the subroutine, also shown on page 11. Note that $\underset{\sim\sim}{B}_k$ and $\underset{\sim\sim}{C}_k$ act as a "call" and a "return" signal respectively. Finally, statement δ *is* constructed by recursion.

We now claim that even though α and β are not constructed by recursion they behave





---

just as if they were. We will now prove this in the case of α; the same argument applies to β. We are essentially claiming that α is still a strong computer for the predicate $y'_k = 0$ and the side effect $y_k \leftrightarrow y'_k$. The assertions (1a) and (2a) are as before. Now assume that $y'_k = 0$. Then there is a sequence of guesses that will call the subroutine with

$$y'_k = A_k,\ y_k = 0,\ S_k = A_k,\ S'_k = 0.$$

By induction there is a sequence of guesses that will reach the YES exit of δ and thus C_k can be set to 1. Thus when $y'_k = 0$ control can reach the YES exit of α. On the other hand, assume that control does reach the YES exit of α and yet y'_k was positive. Then no matter what sequences of guesses occurred $S'_k > 0$ was true when the subroutine was called. Therefore, by induction, control will never reach the YES exit of δ, and hence C_k is never set to 1. This is a contradiction. It remains only to see that the sole side effect is the interchange of y_k and y'_k when $y_k = 0$. Suppose, therefore, that control reaches the YES exit of α. Then as we have seen y'_k is set to A_k and y_k to 0 and S_k to A_k and S'_k to 0 on calling the subroutine. But by induction S_k and S'_k will be interchanged. Therefore the side effects are correct.

In order to complete the proof of the lemma we need only observe that the size of the construction for $k \leq$ the size for $k-1$ plus some fixed constant number of statements. □

5. Proof of Theorem 1

In this section we complete the proof of Theorem 1.

Lemma 4: Suppose that

$$x_1\ x'_1\ y_1\ y'_1\ S_1\ S'_1\ B_1\ C_1 \cdots x_k\ x'_k\ y_k\ y'_k\ S_k\ S'_k\ B_k\ C_k\ r\ r'$$

are variables such that

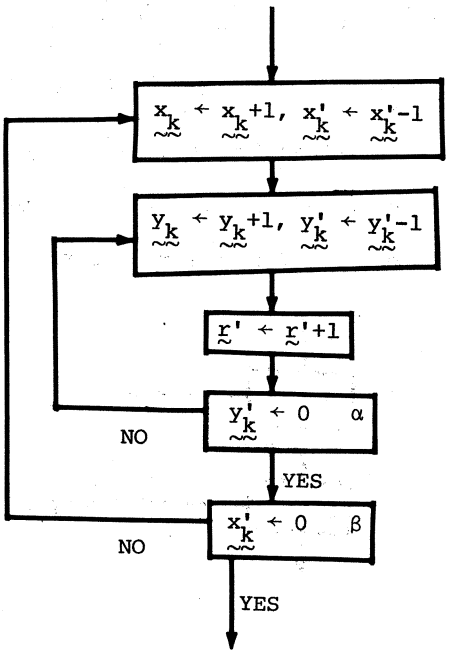
1) $r = r' = 0$.

2) $x_i = y_i = S_i = 0$ and $x'_i = y'_i = S'_i = A_i$ for all i.

3) $\underset{\sim\sim}{B_i} = \underset{\sim\sim}{C_i} = 0$ for all i.

Then there is a program that sets $\underset{\sim}{r}'$ to A_{k+1} and leaves all the values of the other variables unchanged and has at most εk statements for some ε that does not depend on k.

Proof: The program is:



We use Lemma 3 to do the decision statements α and β. □

Lemma 5: Suppose that

$$\underset{\sim\sim}{x_1}\ \underset{\sim\sim}{x_1}'\ \underset{\sim\sim}{y_1}\ \underset{\sim\sim}{y_1}'\ \underset{\sim\sim}{S_1}\ \underset{\sim\sim}{S_1}'\ \underset{\sim\sim}{B_1}\ \underset{\sim\sim}{C_1} \cdots \underset{\sim\sim}{x_k}\ \underset{\sim\sim}{x_k}'\ \underset{\sim\sim}{y_k}\ \underset{\sim\sim}{y_k}'\ \underset{\sim\sim}{S_k}\ \underset{\sim\sim}{S_k}'\ \underset{\sim\sim}{B_k}\ \underset{\sim\sim}{C_k}$$

are variables that are equal to 0. Then there is a program that is a polynomial-in-k size program that sets these variables as follows:

1) $\underset{\sim\sim}{x_i} = \underset{\sim\sim}{y_i} = \underset{\sim\sim}{S_i} = 0$ for all i.

2) $\underset{\sim\sim}{B_i} = \underset{\sim\sim}{C_i} = 0$ for all i

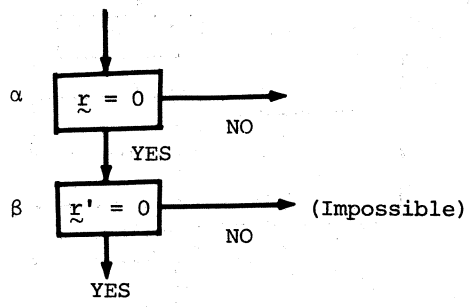
3) $\underset{\sim\sim}{x_i}' = \underset{\sim\sim}{y_i}' = \underset{\sim\sim}{S_i}' = A_i$ for all i.

Proof: Use Lemma 4 repeatedly. □

We now complete the proof of Theorem 1. By Lemmas 4 and 5 we can initialize the variables

$$\underset{\sim}{x}_1\ \underset{\sim}{x}'_1\ \underset{\sim}{y}_1\ \underset{\sim}{y}'_1\ \underset{\sim}{S}_1\ \underset{\sim}{S}'_1\ \underset{\sim}{B}_1\ \underset{\sim}{C}_1 \cdots \underset{\sim}{x}_k\ \underset{\sim}{x}'_k\ \underset{\sim}{y}_k\ \underset{\sim}{y}'_k\ \underset{\sim}{S}_k\ \underset{\sim}{S}'_k\ \underset{\sim}{B}_k\ \underset{\sim}{C}_k\ \underset{\sim}{r}\ \underset{\sim}{r}'$$

so that $\underset{\sim}{x}_i = \underset{\sim}{y}_i = \underset{\sim}{S}_i = \underset{\sim}{B}_i = \underset{\sim}{C}_i$, $\underset{\sim}{x}'_i = \underset{\sim}{y}'_i = \underset{\sim}{S}'_i = A_i$, $\underset{\sim}{r} = 0$, and $\underset{\sim}{r}' = A_k+1$. We will now show how to use $\underset{\sim}{r}$ to simulate a *counter* with values from 0 to 2^{2^k}. By counter here we mean that we can (1) increment and decrement $\underset{\sim}{r}$ and (2) test $\underset{\sim}{r} = 0$. Clearly (1) is trivial. The key of course is that Lemma 3 allows us to test $\underset{\sim}{r} = 0$ as follows:



Here α and β are as in Lemma 3. The repeated test is used to avoid the side effect of interchanging $\underset{\sim}{r}$ and $\underset{\sim}{r}'$. Finally we need only note that we can obtain three counters with values from 0 to 2^{2^2}; hence, we can simulate any Turing machine that uses at most 2^k tape.

This completes the proof of Theorem 1.

Corollary: The boundness of a vector addition system [4], a decidable property, also requires infinitely often exponential space.

Proof: This is a corollary to the proof method of Theorem 1. It is easy to see that the simulation of a counter with values 0 to 2^{2^k} is by a bounded vector addition system. Thus

we can construct a vector addition system that is unbounded if and only if some Turing machine that uses exponential space accepts. □

References

1. M. Hack
   Private communication.
2. N. D. Jones.
   Private communication.
3. R. M. Karp.
   Reducibility among combinatorial problems.
   In Raymond E. Miller and James W. Thatcher, *Complexity of Computer Computations*, 85-104. Plenum, 1972.
4. R. M. Karp and R. E. Miller.
   Parallel program schemata.
   JCSS 3(2):147-195, 1969.
5. R. M. Keller.
   Vector replacement systems: A formalism for modeling asynchronous problems.
   Technical Report 117, Princeton University, 1972.
6. L. Landweber.
   Private communication.
7. A. R. Meyer.
   Private communication.
8. A. R. Meyer and L. J. Stockmeyer.
   The equivalence problem for regular expressions with squaring requires exponential space.
   Proceedings of the 13th Annual SWAT Conference, 1973.
9. M. O. Rabin.
   Cited in Karp and Miller [4].
10. L. J. Stockmeyer.
    The complexity of decision problems in automata theory and logic.
    Project MAC Technical Report 133, MIT, 1974.