A General NP-Completeness Theorem

Nimrod Megiddo*

(Revised)

Abstract. A detailed model of a random access computation over an abstract domain is presented, and the existence of an NP-complete problem is proven under broad conditions which unify Cook's theorem and recent results in the real number model by Blum, Shub and Smale.

1. Introduction

Blum, Shub and Smale [2] formalized a model of computation over a general ring. They proved an analogue of Cook's theorem [3] over the reals. Smale [4] has recently raised the question of existence of NP-complete problems relative to "linear" machines, i.e., machines which add, subtract, multiply by constants, and branch, depending on the sign of a number.

My motivation for writing up this note is the encouragement I received from Steve Smale. In a number of conversations we had on the subject at IMPA, I said I could prove a general theorem on the existence of NP-complete problems, but I was not particularly excited about dealing with all the details. Steve convinced me to a certain extent that there were some subtleties involved, and that I should indeed work under a precise model.

In this note I indeed state a theorem about the existence of NP-complete problems in general. The model of computation unifies the results of Cook and of Blum, Shub and Smale. In particular, it gives an affirmative answer to Smale's question. I prefer to present the model in terms more familiar to computer scientists, namely, using RAMs instead of flowcharts. In Section 2 I define an abstract domain of computation. The model is described in Section 3. The problem of satisfiability over the abstract domain is discussed in Section 4 and its NP-completeness is proven.

*IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099, and School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel. Parts of this work were done during a visit to IMPA, Rio de Janeiro.

2. An abstract domain

Let D be any set such that $|D| \geq 2$, and consider any *finite* set of binary operations over D. A typical operation will be denoted by $\circ$, so that for any $\xi, \eta \in D$, $\zeta = \xi \circ \eta \in D$ is well-defined. The finiteness assumption is crucial. However, a single binary operation induces a possibly infinite set of unary operations corresponding to operating on constants. For example, even if the only operation is that of multiplying two reals, we may still consider the infinite set of multiplications of a variable by some real constant. Denote by $\mathcal{O}_1$ the finite set of operations which may be applied to any two variable values, and let $\mathcal{O}_2$ denote the set of operations which induce the unary operations as explained above.

Let T be a nonempty proper subset of D. Members of D are assigned with truth-values, so that the members of T are "true" and the others are "false." For $\xi \in T$ we write $\tau(\xi) = 1$, and otherwise $\tau(\xi) = 0$. For the convenience of presentation, let us fix two elements: **true** $\in T$ and **false** $\in D \setminus T$. We denote by $\mathcal{D} = \langle D, \mathcal{O}_1, \mathcal{O}_2, T \rangle$ the underlying domain of computation. If $D = \{\textbf{true}, \textbf{false}\}$ and $\mathcal{O}_1 = \mathcal{O}_2 = \emptyset$, our model gives Cook's original theorem. A linear machine over the reals can be handled using this formalism by choosing D to be the set of real numbers, $\mathcal{O}_1 = \{+, -\}$, $\mathcal{O}_2 = \{\times\}$ (where $\times$ is multiplication over the reals) and T is the set of positive reals. The "real number" model corresponds to $\mathcal{O}_1 = \mathcal{O}_2 = \{+, -, \times, \div\}$.

3. Random Access Machines over abstract domains

To distinguish the model of the present paper from that of Blum, Shub and Smale, I talk about *programs*, rather than *machines*.

Two models of computation are considered equivalent if they induce the same set of computable functions, and the time complexities of the computable functions are related polynomially. There are many equivalent models of computation. I have chosen to generalize the model of a RAM as described by Aho, Hopcroft and Ullman [1], so I try to stay as close as possible to the notation used by those authors.

A RAM has a memory and two tapes: a read-only input tape and a write-only output tape. The input consists of a finite sequence of members of D and a finite sequence of positive integers. The combined length of these sequences is called the *size* of the input. The output consists of a sequence of members of D. I would like to deal with computation over an abstract domain, and also use indirect addressing. Indirect addressing is important since one finite program can explicitly use only a fixed number of addresses, whereas the input problems may be of arbitrary size and need more addresses than the ones that appear in the program explicitly. The addresses are given by integers. Thus, I have to distinguish two types of memory registers. The registers of the first type are denoted $R_0, R_1, \ldots$ and each of them is capable of holding one member of D or a

blank. The registers of the second type are denoted $I_0, I_1, \ldots$ and each of them is capable of holding an integer of arbitrary magnitude. These integers are used for addresseing only and as we shall see later, the computational capability associated with them is very limited.

Operands

All computation over $\mathcal{D}$ takes place in the register R_0 whereas all computation over the integers takes place in I_0. Each *instruction* of the program consists of an *operation code* and an *address*. The possible operands are:

(i) $=i$, indicating the integer i.
(ii) A nonnegative integer i, indicating the contents of either R_i or I_i, depending on the instruction.
(iii) $*i$, the contents of either R_j or I_j, where $j \geq 0$ is the integer currently stored in register I_i.
(iv) $=\xi$ (where $\xi \in D$), indicating the element ξ.

We denote the integer held in I_i by $j(i)$ and the element (or the blank) held in R_i by $c(i)$. The output tape is always assumed to be blank initially. By *deterministic computation* we mean that initially $j(i) = 0$ and $c(i)$ is a blank for all $i \geq 0$. By *nondeterministic computation* we mean that the memory may initially hold a finite number of members of D and integers in the respective registers.[1]

The *value* $v(a)$ *of an operand* a depends on the instruction and is defined as follows:

(i) $v(=i) = i$ and $v(=\xi) = \xi$.
(ii) $v(i) = j(i)$ if the instruction operates on integers, and $v(i) = c(i)$ if it operates on members of D.
(iii) $v(*i) = j(j(i))$ if the instruction operates on integers, and $v(*i) = c(j(i))$ if it operates on members of D.

Instructions

The RAM instructions are:

(i) HALT: Stop the execution.
(ii) ILOAD a: This instruction operates on integers. It assigns $v(a)$ to $j(0)$.
(iii) DLOAD a: This instruction operates on members of D (or blanks). If $v(a)$ is not a blank then it assigns it to $c(0)$; otherwise, HALT.

[1]The class NP will be defined more precisely later.

(iv) ISTORE i: Assign $j(0)$ to $j(i)$.
(v) ISTORE $*i$: If $j(i) \geq 0$, then assign $j(0)$ to $j(j(i))$; otherwise, HALT.
(vi) DSTORE i: Assign $c(0)$ to $c(i)$.
(vii) DSTORE $*i$: If $j(i) \geq 0$, then assign $c(0)$ to $c(j(i))$; otherwise, HALT.
(viii) $\circ\ a$ (for any $\circ \in \mathcal{O}_1$): This instruction operates on members of D (or blanks). If $v(a)$ is not a blank then assign $c(0) \circ v(a)$ to $c(0)$; otherwise, HALT.
(ix) $\circ_\xi$ (for any $\circ \in \mathcal{O}_2$ and any $\xi \in D$). Assign $\xi \circ c(0)$ to $c(0)$.
(x) INC: Assign $j(0) + 1$ to $j(0)$.
(xi) DEC: Assign $j(0) - 1$ to $j(0)$.
(xii) READ i: Assign the current input symbol to $c(i)$ (and move the input head one step ahead).
(xiii) READ $*i$: If $j(i) \geq 0$, then assign the current input symbol to $c(j(i))$; otherwise, HALT.
(xiv) WRITE a: This instruction operates on members of D (or blanks). If $v(a)$ is not a blank, then print it on the output tape (and move the output head one step ahead); otherwise, HALT.
(xv) JUMP ℓ: Go to the instruction labeled ℓ.
(xvi) JUMPT ℓ: If $c(0) \in T$, then go to the instruction labeled ℓ; otherwise, go to the next instruction.
(xvii) JUMP0 ℓ: If $j(0) = 0$, then go to the instruction labeled ℓ; otherwise, go to the next instruction.

The classes P and NP

We assume the uniform cost model, i.e., the running time of the program equals the number of instructions executed. A program is said to run in polynomial time if its running time is bounded by some polynomial in terms of the input size. Denote by D^* the set of all finite sequences of members of D. Similarly, denote by Z^* the set of all finite sequences of integers.

A subset of $D^* \times Z^*$ is called a *language*. A language L is said to be in the class P if there exists a (deterministic) RAM which runs in polynomial time and outputs **true** if the input $(\sigma, \nu) \in D^* \times Z^*$ is in L and **false** otherwise.

A language $L \subseteq D^* \times Z^*$ is in the class NP if there exists a RAM with the properties as follows. There exists a polynomial n^k such that if $(\sigma, \nu) \in L$, then with some initialization of the registers $R_0, R_1, \ldots$ with either members of D or blanks, and with some initialization of the registers $I_0, I_1, \ldots$ with integers, the RAM runs in time n^k and outputs **true**; if $(\sigma, \nu) \in D^* \setminus L$, then there is no initialization of the memory with which the machine outputs **true** and halts.

NP**-completeness**

As usual, a language L_0 is said to be NP-*complete* if for every L in NP, there exists a polynomial-time RAM which converts members of L into members of L_0 and nonmembers of L into nonmembers of L_0. We prove below the existence of an NP-complete language.

4. Satisfiability over $\mathcal{D}$

In this section we introduce a certain computational problem and prove its completeness for NP. More specifically, we show that if L is in NP, then the problem of recognizing whether or not a given input is in L can be reduced in polynomial time to the problem of recognizing the existence of members of D and integers, which together satisfy a certain system of constraints. A precise definition of this satisfiability problem is given in Subsection 4.1.

4.1 Definition of Satisfiability over $\mathcal{D}$

The problem of satisfiability over D is stated as a conjunction of constraints as follows. We use three types of *variables:*

(i) x_j – attains any value in D,
(ii) y_j – attains a value in $\{\mathbf{true}, \mathbf{false}\} \subseteq D$,
(iii) z_j – attains any integer value.

Using the above variables, we build *elementary propositions* of the following types:

(i) $x_j = \xi$ (for any $\xi \in D$)
(ii) $x_j = x_k$
(iii) $x_j = x_k \circ x_\ell$ (for any $\circ \in \mathcal{O}_1$)
(iv) $x_j = x_k \circ \xi$ (for any $\circ \in \mathcal{O}_2$ and $\xi \in D$)
(v) $x_j \in T$
(vi) $x_j \notin T$
(vii) $y_j = \mathbf{true}$
(viii) $y_j = \mathbf{false}$
(ix) $z_j = i$ (for any integer i)
(x) $z_j \neq i$ (for any integer i)
(xi) $z_j = z_k$
(xii) $z_j = z_k + 1$

A *constraint* is either an elementary proposition or an implication of the form:

$$\phi_1 \wedge \phi_2 \wedge \phi_3 \Rightarrow \phi_4 \ ,$$

where ϕ_1, ϕ_2, ϕ_3 and ϕ_4 are elementary propositions.

4.2 Satisfiability over $\mathcal{D}$ is in NP

In this subsection we prove that there exists a nondeterministic RAM which solves the satisfiability problem over $\mathcal{D}$ in polynomial time.

Encoding

We first discuss how an instance of the satisfiability problem over $\mathcal{D}$ is encoded as an element of $D^* \times Z^*$, i.e, a pair consisting of a finite string of elements of D and a finite sequence of integers. As explained above, an instance is a conjunction of constraints. We assume we have an alphabet where each element $\xi \in D$ is represented by a symbol $\underline{\xi}$ and each integer n is represented by a symbol $\underline{n}$. The integers used here are either subscripts of names of variables or integer constants that appear in the constraints. For simplicity, let us omit the underbars. Besides elements of D and integer constants, an instance of satisfiability consists of objects from a finite set for which we have to design an encoding scheme: (i) the start of a new conjunct, (ii) members of $\mathcal{O}_1$ and $\mathcal{O}_2$, and (iii) the symbols x, y, z, $=$, $\neq$, $*$, $\in$, $\notin$, T and $\Rightarrow$. Obviously, one can easily design a binary encoding scheme using the symbols 0 and 1, say, to encode all the necessary information.

Testing a solution

We claim that the problem of satisfiability over $\mathcal{D}$ can be solved in nondeterministic polynomial time. To prove this claim we have to exhibit a RAM which receives a code of a set of constraints as input. If there are values of the variables that satisfy the constraints, the RAM confirms this fact. We have to confirm a conjunction of constraints, each of which is either an elementary proposition or an implication involving at most four elementary propositions. Thus, it suffices to show how each type of an elementary proposition can be confirmed. Let us interpret the initial contents of the registers R_i $(i = 1, \ldots, n)$ as the "guesses" of values for the variables x_i, respectively. Similarly, the contents of R_{n+i} will be the guess for the y_i's and that of I_i will the guess for z_i. For each type of a constraint there will a segment in the program where that constraint is tested. If a violated constraint is detected, the program halts with no output. If the validity of all the constraints has been confirmed, the program outputs **true** and halts. It is straightforward to see how each type of elementary proposition is confirmed.

4.3 Reducing problems in NP to Satisfiability over $\mathcal{D}$

Suppose L is in NP and let n^k be a polynomial time bound for a corresponding nondeterministic RAM which we denote by Π. Without loss of generality, we also assume that n^k is a bound on the index i of any register R_i or I_i which is used throughout the execution

of the algorithm. Moreover, we assume without loss of generality that the integers which are held in any register I_i are nonnegative and not greater than n^k.

Given an input (σ, ν) of length n, let $m = n^k$. We construct a system of constraints on the values (in D) of variables $x_{00}, x_{01}, \ldots, x_{0m}$, representing the initial contents of registers $R_0, R_1, \ldots, R_m$, respectively, and a system of constraints on the (integer) values of the variables $z_{00}, z_{01}, \ldots, z_{0m}$, representing the initial contents of registers $I_0, I_1, \ldots, I_m$. These variables may be viewed as the independent variables, while the values of the auxiliary variables introduced below are determined by the values of the x_{0j}'s and the z_{0j}'s. In particular, let x_{tj} denote the contents of R_j after t time units. Analogously, let z_{tj} denote the contents of I_j after t time units.

We assume the instructions of Π are labeled by consecutive integers $\ell = 1, \ldots, s$ ($s \geq 2$). Without loss of generality, assume the instruction labeled s is the only HALT in the program.

For each instruction labeled ℓ, and for $t = 1, \ldots, m$, let $y_{t\ell}$ be a variable such that $y_{t\ell} = \textbf{true} \in D$ if the instruction that is executed during the t'th time unit is the one labeled ℓ, and $y_{t\ell} = \textbf{false} \in D$ otherwise. Similarly, for every i ($i = 1, \ldots, n$) let u_{ti} be a variable such that $u_{ti} = \textbf{true}$ if during the t'th time unit the input head is positioned over the i'th square of the input tape, and $u_{ti} = \textbf{false}$ otherwise. Also, for every i ($i = 1, \ldots, m$) let v_{ti} be a variable such that $v_{ti} = \textbf{true}$ if during the t'th time unit the output head is positioned over the i'th square of the output tape, and $v_{ti} = \textbf{false}$ otherwise. Finally, denote by τ_j ($j = 1, \ldots, m$) the contents of the j'th square of the output tape at the end of the run.

Below we state constraints ensuring the correct interpretation of the variables defined above.

4.4 Constraints

We first introduce constraints to ensure that at most one instruction is executed during each time unit, and the input head and the output head are each positioned over at most one square.

General constraints

For $t = 0, 1, \ldots, m$, we impose the following constraints:

(i) For $\ell = 1, \ldots, s$ and $q \neq \ell$, $q = 1, \ldots, s$,

$$(y_{t\ell} = \textbf{true}) \Rightarrow (y_{tq} = \textbf{false}) \ ,$$

and also

$$y_{01} = \textbf{true} \ .$$

(ii) For $i = 1, \ldots, n$ and $j \neq i$, $j = 1, \ldots, n$,

$$(u_{ti} = \textbf{true}) \Rightarrow (u_{tj} = \textbf{false}) \ ,$$

and also

$$u_{01} = \textbf{true} \ .$$

(iii) For $i = 1, \ldots, m$ and $j \neq i$, $j = 1, \ldots, m$,

$$(v_{ti} = \textbf{true}) \Rightarrow (v_{tj} = \textbf{false}) \ ,$$

and also

$$v_{01} = \textbf{true} \ .$$

The RAM recognizes an instance of L by printing **true** in the first square of the output tape. Thus, we impose the constraint:

$$\tau_1 = \textbf{true} \ .$$

The rest of the constraints are derived directly from the definitions of the various instructions.

Consider any instruction whose label is ℓ. We impose a set of constraints, as a function of t, for $t = 1, \ldots, m$, depending on the nature of the instruction.

(i) If the instruction is one of the following: ILOAD, DLOAD, ISTORE, DSTORE, $\circ$, $\circ_\xi$, INC, DEC, READ and WRITE, then we impose the following constraint:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (y_{t+1,\ell+1} = \textbf{true}) \ .$$

(ii) If the instruction is one of the following: ILOAD, ISTORE, WRITE, JUMP, JUMPT and JUMP0, then we impose the following constraint:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (x_{ti} = x_{t-1,i}) \quad (i = 0, \ldots, m) \ .$$

(iii) If the instruction is one of the following: DLOAD, DSTORE, $\circ$, $\circ_\xi$, READ, WRITE JUMP, JUMPT, JUMP0, then we impose the following constraint:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (z_{ti} = z_{t-1,i}) \quad (i = 0, \ldots, m) \ .$$

(iv) For any instruction other than READ,

$$(y_{t\ell} = \textbf{true}) \Rightarrow (u_{t+1,i} = u_{t,i}) \quad (i = 1, \ldots, n) \ .$$

(v) For any instruction other than WRITE,

$$(y_{t\ell} = \textbf{true}) \Rightarrow (v_{t+1,i} = v_{t,i}) \quad (i = 1, \ldots, m) \ .$$

More instruction related constraints are as follows.

HALT:

Recall that we have assumed the only HALT in the program is the last instruction. We impose the following constraint:

$$(y_{ts} = \textbf{true}) \Rightarrow (y_{t+1,s} = \textbf{true}) \ .$$

ILOAD a

Here a may be either $=i$, i or $*i$. First, we impose

$$(y_{t\ell} = \textbf{true}) \Rightarrow (z_{tj} = z_{t-1,j}) \quad (j = 1, \ldots, m) \ .$$

Next, if a is $=i$, we write:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (z_{t0} = i) \ ,$$

and if a is i, we write:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (z_{t0} = z_{t-1,i}) \ .$$

If a is $*i$, we write for every j, $j = 0, \ldots, m$,

$$((y_{t\ell} = \textbf{true}) \wedge (z_{t-1,i} = j)) \Rightarrow (z_{t0} = z_{t-1,j}) \ .$$

DLOAD a**:**

Here a is either $=\xi$, i or $*i$. First,

$$(y_{t\ell} = \textbf{true}) \Rightarrow (x_{tj} = x_{t-1,j}) \quad (j = 1, \ldots, m) \ .$$

Without loss of generality we assume that $v(a)$ is a member of D (rather than a blank). Next, if a is $=\xi$, we write:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (x_{t0} = \xi) \ ,$$

and if a is i, we write:

$$(y_{t\ell} = \textbf{true}) \Rightarrow (x_{t0} = x_{t-1,i}) \ .$$

If a is $*i$, we write for every j, $j = 0, \ldots, m$,

$$((y_{t\ell} = \textbf{true}) \wedge (z_{t-1,i} = j)) \Rightarrow (x_{t0} = x_{t-1,j}) \ .$$

ISTORE i**:**

(i) $(y_{t\ell} = \textbf{true}) \Rightarrow (z_{tj} = z_{t-1,j})$ $(j = 0, \ldots, m, \ j \neq i)$.
(ii) $(y_{t\ell} = \textbf{true}) \Rightarrow (z_{ti} = z_{t-1,0})$.

ISTORE $*i$:

(i) $((y_{t\ell} = \mathbf{true}) \wedge (z_{t-1,i} \neq j)) \Rightarrow (z_{tj} = z_{t-1,j})$ $(j = 0, \ldots, m)$.
(ii) $((y_{t\ell} = \mathbf{true}) \wedge (z_{t-1,i} = j)) \Rightarrow (z_{tj} = z_{t-1,0})$ $(j = 0, \ldots, m)$.

DSTORE i:

(i) $(y_{t\ell} = \mathbf{true}) \Rightarrow (x_{tj} = x_{t-1,j})$ $(j = 0, \ldots, m, j \neq i)$.
(ii) $(y_{t\ell} = \mathbf{true}) \Rightarrow (x_{ti} = x_{t-1,0})$.

DSTORE $*i$:

(i) $((y_{t\ell} = \mathbf{true}) \wedge (z_{t-1,i} \neq j)) \Rightarrow (x_{tj} = x_{t-1,j})$ $(j = 0, \ldots, m)$.
(ii) $((y_{t\ell} = \mathbf{true}) \wedge (z_{t-1,i} = j)) \Rightarrow (x_{tj} = x_{t-1,0})$ $(j = 0, \ldots, m)$.

$\circ\, a$ and $\circ_\xi$:

First,

$$(y_{t\ell} = \mathbf{true}) \Rightarrow (x_{tj} = x_{t-1,j}) \quad (j = 1, \ldots, m) \ .$$

In the case of $\circ_\xi$,

$$(y_{t\ell} = \mathbf{true}) \Rightarrow (x_{t0} = x_{t-1,0} \circ \xi) \ .$$

In the case of $\circ\, a$, if a is i then

$$(y_{t\ell} = \mathbf{true}) \Rightarrow (x_{t0} = x_{t-1,0} \circ x_{t-1,i}) \ ,$$

and if a is $*i$, then we write for every j, $(j = 0, \ldots, m)$

$$((y_{t\ell} = \mathbf{true}) \wedge (z_{t-1,i} = j)) \Rightarrow (x_{t0} = x_{t-1,0} \circ x_{t-1,j}) \ .$$

INC and DEC:

(i) $(y_{t\ell} = \mathbf{true}) \Rightarrow (z_{tj} = z_{t-1,j})$ $(j = 1, \ldots, m)$.
(ii) $(y_{t\ell} = \mathbf{true}) \Rightarrow (z_{t0} = z_{t-1,0} \pm 1)$ ($+$ in the case of INC, $-$ in the case of DEC).

READ i:

(i) $((y_{t\ell} = \mathbf{true}) \wedge (u_{tj} = \mathbf{true})) \Rightarrow (x_{ti} = \sigma_j) \quad (j = 1, \ldots, n)$.
(ii) $((y_{t\ell} = \mathbf{true}) \wedge (u_{tj} = \mathbf{true})) \Rightarrow (u_{t+1,j+1} = \mathbf{true}) \quad (j = 1, \ldots, n)$.

READ $*i$**:**

(i) $((y_{t\ell} = \textbf{true}) \wedge (u_{tj} = \textbf{true}) \wedge (z_{t-1,i} = q)) \Rightarrow (x_{tq} = \sigma_j)$ $(j = 1, \ldots, n,\ q = 1, \ldots, m)$.
(ii) $((y_{t\ell} = \textbf{true}) \wedge (u_{tj} = \textbf{true})) \Rightarrow (u_{t+1,j+1} = \textbf{true})$ $(j = 1, \ldots, n)$.

WRITE i**:**

(i) $((y_{t\ell} = \textbf{true}) \wedge (v_{tj} = \textbf{true})) \Rightarrow (\tau_j = x_{ti})$ (for $j = 1, \ldots, m$).
(ii) $((y_{t\ell} = \textbf{true}) \wedge (v_{tj} = \textbf{true})) \Rightarrow (v_{t+1,j+1} = \textbf{true})$ (for $j = 1, \ldots, m$).

WRITE $*i$**:**

(i) $((y_{t\ell} = \textbf{true}) \wedge (v_{tj} = \textbf{true}) \wedge (z_{t-1,i} = q)) \Rightarrow (\tau_j = x_{tq})$ (for $j = 1, \ldots, m$ and $q = 1, \ldots, m$).
(ii) $((y_{t\ell} = \textbf{true}) \wedge (v_{tj} = \textbf{true}) \Rightarrow (v_{t+1,j+1} = \textbf{true})$ (for $j = 1, \ldots, m$).

JUMP ℓ'**:**

$(y_{t\ell} = \textbf{true}) \Rightarrow (y_{t+1,\ell'} = \textbf{true})$.

JUMPT ℓ'**:**

(i) $((y_{t\ell} = \textbf{true}) \wedge (x_{t-1,0} \in T)) \Rightarrow (y_{t+1,\ell'} = \textbf{true})$.
(ii) $((y_{t\ell} = \textbf{true}) \wedge (x_{t-1,0} \notin T)) \Rightarrow (y_{t+1,\ell+1} = \textbf{true})$.

JUMP0 ℓ'**:**

(i) $((y_{t\ell} = \textbf{true}) \wedge (z_{t-1,0} = 0)) \Rightarrow (y_{t+1,\ell'} = \textbf{true})$.
(ii) $((y_{t\ell} = \textbf{true}) \wedge (x_{t-1,0} \neq 0)) \Rightarrow (y_{t+1,\ell+1} = \textbf{true})$.

Proposition 4.1. *For any language L in* NP *and any nondeterministic polynomial-time RAM for recognizing instances in L, given an input (σ, ν), the system of constraints defined in Subsection 4.4 can be constructed in polynomial time. Moreover, the string (σ, ν) is in L if and only if there exists an assignment of appropriate values to the variables x_{ti}, y_{ti}, z_{ti}, u_{ti}, v_{ti}, and τ_j, so that all the constraints are satisfied.*

Proof: The proof follows from the fact if the interpretation of the variables is correct (and this is guaranteed if the constraints are satisfied), then the state of the machine at every stage of the execution is described precisely by these variables. ∎

To summarize the reduction, suppose a description of a RAM Π in NP is given. Recall that the constraints are imposed for each time unit t. It is easy to see from the above that we can write a program Π^* (i.e., a RAM) that does the following. It receives as input any pair (σ, ν) which is presented to Π. Recognizing the length the input, Π^* develops the set of constraints defined above and creates an instance of the satisfiability problem.

Acknowledgement. Helpful conversations with Steve Smale are gratefully acknowledged.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1976.

[2] L. Blum, M. Shub and S. Smale, "On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines," *Bulletin of the American Mathematical Society* **21** (1989) 1–46.

[3] S. A. Cook, "The complexity of theorem proving procedures," *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing* (1971), pp. 151–158.

[4] S. Smale, Talk at the Workshop on Computational Complexity, IMPA, Rio de Janeiro, January, 1990.