

High level operations of TMs

- Compare if two strings are the same or not.
- Compute the addition, subtraction, multiplication, division, power, log, etc. of numbers in unary form.
- Shift a string right (or left).
- Maintain a base- b counter.

22c:131 Limits of Computation

Hantao Zhang

<http://www.cs.uiowa.edu/~hzhang/c131>

The University of Iowa
Department of Computer Science

Answer

The three possibilities are:

1. *Formal description*: spells out in full all 7 components of a Turing machine. This is the lowest, most detailed level of description.
2. *Implementation description*: use English prose to describe the way Turing machine moves its head and the way it stores data on its tape. No details of state transitions are given
3. *High-level description*: use English prose to describe the algorithm, ignoring the implementation model. No need to mention how machine manages its head and tape.

Standardizing our model

Question: *what is the right level of detail to give when describing a Turing machine algorithm?*

Note: this is a common question asked especially when preparing solutions to various problems such as exercises and problems given in assignments and exams during the process of learning Theory of Computation

Variants of Turing Machine

Transition function of a standard TM in our definition forces the head to move to the left or right after each step. Let us vary the type of transition function permitted.

- Suppose that we allow the head to *stay put*, i.e.;
 $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$
- *S* transition can be represented by two standard transitions: one that move to the left followed by one that moves to the right.
- Since we can convert a TM which stay put into one that doesn't have this facility, the extension does not increase its power.

Equivalence of TMs

To show that two models of TM are equivalent we need to show that we can simulate one by another.

Theorem 3.13

Every multitape Turing machine has an equivalent single tape Turing machine.

Proof: We show how to convert a multitape TM M into a single tape TM S . The key idea is to show how to simulate M with S .

Multitape Turing Machines

A multitape TM is like a standard TM with several tapes

- Each tape has its own head for reading/writing
- Initially the input is on tape 1 and other tapes are blank
- Transition function allow for reading, writing, and moving the heads on all tapes simultaneously, i.e.,
 $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, where k is the number of tapes.
- $\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$ means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k the machine goes to state q_j , writes b_1 through b_k on tapes 1 through k , respectively, and moves each head to the left or right as specified.

Example simulation

Figure 1 shows how to represent a machine with 3 tapes by a machine with one tape.

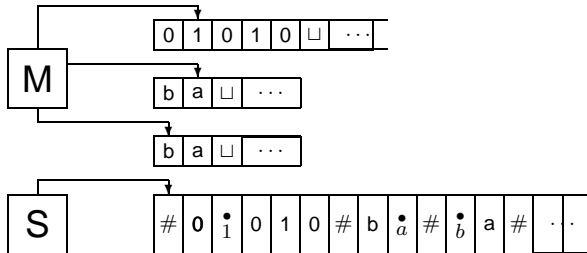


Figure 1: Multitape machine simulation

Simulating M with S

Assume that M has k tapes

- S simulates the effect of k tapes by storing their information on its single tape
- S uses a new symbol $\#$ as a delimiter to separate the contents of different tapes
- S keeps track of the location of the heads by marking with a \bullet the symbols where the heads would be.

Corollary 3.15

A language is Turing recognizable iff some multitape Turing machine recognizes it

Proof:

- **if:** a Turing recognizable language is recognized by an ordinary Turing machine, which is a special case of a multitape Turing machine.
- **only if:** follows from the equivalence of a Turing multitape machine M with the Turing machine S that simulates it. That is, if L is recognized by M then L is also recognized by S

General Construction

$S =$ "On input $w = a_1 a_2 \dots a_n$

1. Put $S(\text{tape})$ in the format that represents $M(\text{tapes})$:

$$S(\text{tape}) = \# \overset{\bullet}{a_1} \dots \overset{\bullet}{a_n} \# \square \dots \# \overset{\bullet}{\square} \#$$
2. Scan the tape from the first $\#$ (which represent the left-hand end) to the $(k + 1)$ -st $\#$ (which represent the right-hand end) to determine the symbols under the virtual heads. Then S makes the second pass over the tape to update it according to the way M 's transition function dictates.
3. If at any time S moves one of the virtual heads to the right of $\#$ it means that M has moved on the corresponding tape onto the unread blank portion of that tape. So, S writes a \square on this tape cell and shifts the tape contents from this cell until the rightmost $\#$, one unit to the right. Then it continues to simulates as before".

Theorem 3.16

Every nondeterministic Turing machine, NTM, has an equivalent deterministic Turing machine, DTM.

Proof idea: show that we can simulate a NTM N with DTM, D .

Note: in this simulation D tries all possible branches of N 's computation. If D ever finds the accept state on one of these branches then it accepts. Otherwise D simulation will not terminate

Nondeterministic TM

- A NTM is defined in the expected way: at any point in a computation the machine may proceed according to several possibilities
- Formally, $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$
- Computation performed by a NTM is a tree whose branches correspond to different possibilities for the machine
- If some branch of the computation tree leads to the *accept* state, the machine accepts the input

A tempting bad idea

Design D to explore $N(w)$ by a *depth-first search*

A depth-first search goes all the way down on one branch before backing up to explore next branch. Hence, D could go forever down on an infinite branch and miss an accepting configuration on an other branch

More on NTM simulation

- N 's computation on an input w is a tree, $N(w)$.
- Each branch of $N(w)$ represents one of the branches of the nondeterminism
- Each node of $N(w)$ is a configuration of N .
- The root of $N(w)$ is the start configuration

Note: D searches $N(w)$ for an accepting configuration

Formal proof

D has three tapes, Figure 2:

1. Tape 1 always contains the input (and the code of N) and is never altered.
2. Tape 2 (called simulation tape) maintains a copy of the N 's tape on some branch of its nondeterministic computation
3. Tape 3 (called address tape) keeps track of D 's location in N 's nondeterministic computation tree

A better idea

Design D to explore the tree by using a *breadth-first search*

This strategy explores all branches at the same depth before going to explore any branch at the next depth. Hence, this method guarantees that D will visit every node of $N(w)$ until it encounters an accepting configuration

Address tape

- Every node in $N(w)$ can have at most b children, where b is the size of the largest set of possible choices given by N 's transition function
- Hence, to every node we assign an address that is a string in the alphabet $\Sigma_b = \{1, 2, \dots, b\}$.
- **Example:** we assign the address 231 to the node reached by starting at the root, going to its second child and then going to that node's third child and then going to that node's first child

Deterministic simulation of NTM

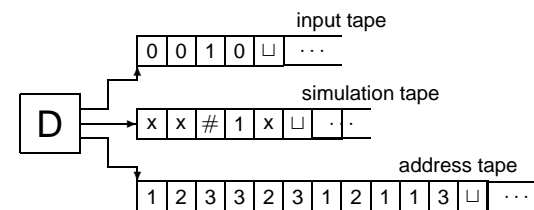


Figure 2: Deterministic TM D simulating N

The description of D

1. Initially tape 1 contains w and tape 2 and 3 are empty
2. Copy tape 1 over tape 2
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation.
 - Before each step of N , consult the next symbol on tape 3 to determine which choice to make among those allowed by N 's transition function
 - If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4.
 - If a rejecting configuration is reached go to stage 4; if an accepting configuration is encountered, *accept* the input
4. Replace the string on tape 3 with the next string as if it is a counter and go to stage 2.

Note

- Each symbol in a node address tells us which choice to make next when simulating a step in one branch in N 's nondeterministic computation
- Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node
- Tape 3 contains a string over Σ_b which represents a branch of N 's computation from the root to the node addressed by that string, unless the address is invalid.
- The empty string ϵ is the address of the root.

Corollary 3.19

A language is decidable iff some NTM decides it.

Corollary 3.18

A language is Turing-recognizable iff some nondeterministic TM recognizes it.

Proof:

- **if:** any deterministic TM is automatically an nondeterministic TM
- **only if:** follow from the fact that any NTM can be simulated by a DTM

Computation of an Enumerator

- An enumerator starts with a blank input tape
- If the enumerator does not halt it may print an infinite list of strings
- The language recognized by the enumerator is the collection of strings that it eventually prints out.

Note: an enumerator may generate the strings of the language it recognizes in any order, possibly with repetitions.

Enumerators

- An enumerator is a variant of a TM with an attached printer (or an output tape)
- The enumerator uses the printer as an output device to print strings
- Every time the TM wants to add a string to the list of recognized strings it sends it to the printer

Note: Some people use the term *recursively enumerable* language for languages recognized by enumerators

Proof, continuation

only if: If M recognizes a language A , we can construct an enumerator E for A . For that consider s_1, s_2, \dots , the list of all possible strings in Σ^* , where Σ is the alphabet of M .

$E =$

1. Let $i = 1$.
2. For $j = 1$ to i , simulate M on s_j at most i steps.
3. If M accepts s_j with i steps or less, prints out s_j .
4. $i = i + 1$, go to 2."

Note: a string may print out multiple times.

Theorem 3.21

A language is Turing-recognizable iff some enumerator enumerates it

Proof:

- **if:** if we have an enumerator E that enumerates a language A then a TM M recognizes A . M works as follows:

$M =$ "On input w :

1. Run E . Every time E outputs a string x , compare it with w .
2. If $w = x$, M *accepts*; else go to 1.

Clearly M accepts those strings that appear on E 's list.

Equivalence with other models

- There are many other models of general purpose computation. **Example:** recursive functions, normal algorithms, semi-Thue systems, λ -calculus, etc.
- Some of these models are very much like Turing machines; other are quite different
- All share the essential feature of a TM: unrestricted access to unlimited memory
- All these models turn out to be equivalent in computation power with TM

Another Proof

only if: If M recognizes a language A , we can construct an enumerator E for A . For that consider s_1, s_2, \dots , the list of all possible strings in Σ^* , where Σ is the alphabet of M . Given a pair of integers $\langle i, j \rangle$, define $\text{Next}(\langle i, j \rangle) = \text{if } (i = 1) \text{ then } \langle j + 1, 1 \rangle \text{ else } \langle i - 1, j + 1 \rangle$.

$E = "$

1. Let $\langle i, j \rangle = \langle 1, 1 \rangle$.
2. Simulate M on s_j at most i steps.
3. If M accepts s_j with exactly i steps, prints out s_j .
4. $\langle i, j \rangle = \text{Next}(\langle i, j \rangle)$, go to 2."

Note: no string prints out more than once.

Algorithm as a Turing Machine

- Alonzo Church and Alan Turing in 1936 came with formal definitions for the concept of algorithm.
- Church used a notational system called λ -calculus to define algorithms.
- Turing used his "Turing Machines" to define algorithms.
- These two definitions were shown to be equivalent.

Algorithms

- Informally speaking an *algorithm* is a collection of simple instructions for carrying out a task.
- In everyday life algorithms are called *procedures* or *recipes*.
- Algorithms abound in contemporary mathematics.

Formal Definition

Definition: an algorithm is a decider TM in the standard representation.

- The input to a Turing machine is always a string.
- If we want an object, other than a string as input, we must first represent that object as a string.
- Strings can easily represent polynomials, graphs, grammars, automata, and any combination of these objects.

Church-Turing Thesis

- Other formal definitions of algorithms have been provided by: Kleene using *recursive functions*, Markov using rewriting (derivation) rules with a grammar called *normal algorithms*.
- Essentially all these formal concepts of algorithm are equivalent among them and are equivalent with Turing Machines.
- **Church-Turing Thesis:** Every computing device can be simulated by a Turing machine.

Example TM

- Let A be the language consisting of all strings representing undirected graphs that are connected.
- **Recall:** a graph is connected if every node can be reached from every other node.
- **Notation:** $A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$

Encoding and Decoding Objects

- Our notation for encoding an object O into its string representation is $\langle O \rangle$.
- If we have several objects O_1, O_2, \dots, O_k we denote their encoding into a string by $\langle O_1, O_2, \dots, O_k \rangle$.
- Encoding itself can be done in many ways. It doesn't matter which encoding we pick because a Turing machine can always translate one encoding into another.
- A (part of) Turing machine may be programmed to decode the input representation so that it can be interpreted the way we intend.

Implementation details

Consider the graph in Figure 3

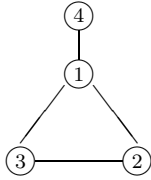


Figure 3: A connected graph

A TM deciding A

$M =$ "On input $\langle G \rangle$, the encoding of G

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked.
3. For each node in G , mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are *accept*, otherwise *reject*."

Checking the encoding

When M receives the input $\langle G \rangle$ it first checks to determine that the input is a proper encoding of some graph:

1. Scan the tape to be sure that there are two lists and that they are in proper form.
2. The first list should be a list of distinct decimal numbers; the second list should be a list of pairs of decimal numbers.
3. The list of decimal numbers should contain no repetitions.
4. Every node on the second list should appear in the first list too.

Note: element distinctness problem can be used to format the lists and to implement the checks above.

Graph encoding, $\langle G \rangle$

- The encoding $\langle G \rangle$ of a graph as a string is a list of nodes followed by a list of edges.
- Each node is a decimal number, and each edge is a pair of decimal numbers that represent the nodes that edge connects
- **Example encoding:** the graph in Figure 3 is encoded by the string:
 $\langle G \rangle = (1, 2, 3)((1, 2), (2, 3), (3, 1), (1, 4))$.