1 Introduction

To try and justify the claim that TMs may represent a correct, or perhaps better, appropriate, model of computation it makes sense to consider what happens when their capacity is enhanced in various ways. We will be able to show that none of the enhancements actually change the languages we can recognise, or the functions that we can compute and this is further evidence of the appropriateness of the model.

This is actually helpful in at least two ways:

- To show that a problem is mechanically solvable we can use a more powerful and convenient extended machine; and
- To show that a problem is not mechanically solvable we can restrict ourselves to ordinary TMs.

2 Multi-track machines

In a multi-track Turing machine, instead of one tape we have several. There is still a single read-write head which simultaneously reads aligned symbols on the various tapes (and can write on them all, before moving in the same direction on all of them).

But, thinking of the column of letters that the read-write head sees as a single letter we see that this is really just an ordinary TM with an expanded alphabet (Γ^k where k is the number of tracks instead of Γ). So, adding multiple tracks does not change the power of a TM.

3 Two-way tapes

In a two-way tape machine we assume that the tape is infinite in both directions (i.e. the tape cells can be indexed by integers rather than by natural numbers).

Given a two-way tape machine, imagine wrapping the tape around (duplicating the 0 cell) so that cell -1 lies above cell 1, cell -2 above cell 2 and so on. The extra copy of the 0

cell (in the upper tape) has a special symbol # written on it. Now simulate the original machine by a new two track machine here by duplicating each state q of the original into a pair q^{upper} and q^{lower}. Then each transition of the original machine can be transformed into transitions of the new one. For instance if we have a transition between q and s that reads a, writes b and moves left, it becomes:

State	Read	Write	Move	New state
q^{lower}	ax	bx	L	s^{lower}
q^{lower}	$a\#$	$b\#$	R	s^{upper}
q^{upper}	xa	xb	R	s^{upper}

For each possible symbol x on the other tape.

This transformation shows the general strategy we need to pursue – take an instance of the more powerful machine and show how to build one of a simpler type whose computations simulate it.

4 Multi-tape machines

In a multi-tape Turing machine we again have a number of separate tapes. Each tape has its own read-write head and these can move independently of one another (or indeed stay in place). Transitions are based on the complete sequence of symbols read at any one time, as well as the current state – that is, there is one central control unit which takes charge of all the different heads simultaneously.

Simulating this using a multi-track machine is a little bit complicated. Suppose that there are k tapes. We will use $2k + 1$ tracks. One of these is pretty much for decoration. It has a # in the leftmost cell and nothing else. Its only purpose is to allow us to reliably wind the head back to the leftmost cell.

For each of the k tapes of the multi-tape machine we dedicate two tracks. One of these contains the actual contents of the tape. The other contains a single symbol M which marks where the 'virtual' version of the read-write head for that tape is currently located.

Now the simulation of a single transition in the multi-tape machine requires a long sequence of operations in the multi-track machine. We always assume that the read-write head begins at the leftmost cell. One by one we advance to the various marked cells and

determine the symbols on each 'virtual' tape, storing these by means of state (this is possible because there are only finitely many possibilities for the symbols we see). Then we rewind back to the left again. Now we determine what should be written on each tape and which way its head should be moved (by means of the original transition table in the multi tape machine). Again, we advance to each of the marks, change the symbol on the virtual copy, and move the mark as necessary. When all this is done, we go back to the left (remembering the new state), and start again.

5 Non-determinism

Non-determinism in TMs comes in two apparently different flavours.

The first, non-determinism by transition (NDT for local reference) is just like non-determinism in finite state or push down automata. That is, given a current state, and a current symbol, there may be more than one possibility for the read-write action, movement direction, or resulting state. Basically our machine is 'loose' and might take one of several actions in certain configurations. An input word w is accepted by such a machine if *some* choice of those actions results in an accepting computation.

The second, non-determinism by oracle (NDO) introduces a new idea. We imagine a standard two tape deterministic TM. One tape is called the input tape and on it we write the input word w that we wish to process. The other is called the oracle tape. Just before setting the computation in motion we summon a genie and say 'Oh most clever and puissant genie, it would be our dearest wish if you would be so kind as to write upon this second tape some helpful guidance for our computation'. Since we were so polite[1], the genie then writes some information on the oracle tape. We then run the machine and accept w if it halts in an accepting state. Since we don't *entirely* trust the genie it is our responsibility to have designed the machine in such a way that the set of words we want to accept is precisely the set of words that will be accepted for *some* string on the oracle tape. That is, while the genie can ensure that we accept the words we want, we can never be tricked into accepting a word that we don't want.

Despite their apparent differences, these two models accept the same languages – which

[1] For reasons that should be obvious, it is always advisable to be polite to powerful supernatural entities.

is useful as the second one is *much* easier to work with in a theoretical context (as well as allowing for all sorts of clever variations where we might restrict access to the oracle tape in some way, or the types of sequences that can be written on it, ...)

Theorem 5.1. *If a language, L, is accepted by an NDT Turing machine, T, then it is accepted by an NDO Turing machine O and vice versa.*

Proof. Suppose first that L is accepted by an NDT machine T. Create a two track machine O where on the oracle track the head always moves one place to the right with each transition. Let m be the maximum degree of non-determinism in T (i.e. the maximum number of transitions associated with a single state-symbol pair) and take the alphabet of the oracle track to be $\{1, 2, \ldots, m\}$. Now simply make each of the non deterministic transitions deterministic by indexing them from 1 to (at most) m, using the contents of the oracle tape to justify the choice. Given an acceptable word w, the genie simply chooses an accepting computation in T and writes the appropriate indices on the oracle tape meaning that O accepts w. However, whatever the genie writes on the oracle tape leads to some computation path in T, so we can never be forced to accept a word that T does not accept.

Now suppose that L is accepted by an NDO machine O. We may assume that O works as in the previous paragraph (i.e. each step advances the read-write head on the oracle tape one step to the right). This is because the contents of the oracle tape are completely under the genie's control. While it might be more convenient to allow any sort of movement on that tape, it can't hurt to simply write out in order all the symbols that will ever be seen there instead (and for a powerful genie, questions of convenience don't really enter into it). Now we produce our NDT T simply by the reverse of the preceding construction – just erase all references to the second tape from the transitions. If T accepts a word w then we can choose an accepting computation, 'remember' which transitions involving the oracle tape were used, and construct an input sequence for the oracle tape that shows that O accepts w. On the other hand if O accepts w then we can just ignore the oracle tape and 'see' an accepting computation of T on w. So, T and O accept the same language. □

6 Non-determinism doesn't help!

Despite the power that non-determinism seems to give us, it doesn't actually change anything!

Theorem 6.1. *If a language, L, is accepted by a non-deterministic Turing machine then it is also accepted by a deterministic Turing machine.*

Proof. Suppose that L is accepted by an NDO, O. To show that L is accepted by a deterministic TM it's enough to show that it's accepted by a deterministic multi-tape TM since we know we can simulate a multi-tape TM in a single-tape TM. Our multi-tape TM will have the following four tapes (and possibly some additional working tapes for convenience):

- The input tape (i.e., the main tape of O)
- An "input copy" tape
- A "simulated oracle" tape
- A "timer" tape, initialised to 1.

The operation is as follows: if the timer is set to k then we simulate the operation of O under every possible string on the oracle tape of length $\leqslant k$, running for k steps. After each simulation, use the input copy to restore the input tape to its original form (you may need to include a marker on the main tape that records the rightmost position ever reached – or just agree that you're not allowed to write "real" blanks on the tape). If any simulation ever accepts we halt and accept. If we finish all simulations with the timer at k without accepting, we increment the timer by 1 and start again.

Obviously, if there's no accepting computation then we won't accept here either. But if there is an accepting computation then we'll wind up accepting when we reach round k with k being the maximum of the size of the oracle tape required and the number of computational steps required (in fact, since we examine at most one cell on the oracle tape per step, only the number of computational steps matters).

There's plenty of book-keeping to do and it's wildly inefficient but eventually, if there is some accepting computation in O which requires say s steps then we will get around to simulating it and accept. If there's no accepting computation for O then, since all we're doing is simulating computations in O, we won't accept either. □

7 Exercises

Use multiple tracks, tapes, or other variations on the Turing theme to find machines that accept the following languages (or compute certain functions). Again, worry more about the high level description, and understanding how, while these variations may improve efficiency, their computations could all be accomplished by a standard TM

1. Compute the n^{th} Fibonacci number f_n (i.e. on input a^n, arrange that a^{f_n} is written on a working tape). Take $f_0 = f_1 = 1$ and for $n > 1$, $f_n = f_{n-1} + f_{n-2}$.
2. Convert binary numbers to unary. That is, on input $w \in \{0,1\}^*$ arrange output on a working tape of a^n where n is the value of w interpreted as a binary number.
3. Convert unary numbers to binary ones.
4. Accept the language $L = \{a^p \mid p \text{ is prime}\}$.
   - Think about the standard loop version "for each $2 \leq i < p$ check whether the remainder when p divided by i is 0. If so, reject. If all these tests succeed, accept.
   - There is an improvement to this method where you only look at $i \leq \sqrt{p}$ since if a number is composite it has a proper factor less than or equal to its square root – how might that be implemented?
   - Does using non-determinism seem to help for this problem? What about for the complement of L, i.e. the set of strings a^n where n is composite?