Deterministic vs. nondeterminism Turing Machines

Deterministic TM's owe their name to the fact that each computation can be viewed as a linear ordered sequence (finite or infinite) of configurations. The first element contains the initial state and each configuration C' follows directly configuration C in the sequence if the TM can change C into C' as a consequence of a legal move of the machine.

Nondeterministic machines are not restricted to such ordered computations. At each moment in time a non-deterministic TM may be allowed to make several moves (or equivalently to choose among a number of possibilities). Hence its computation may "branch" in many possible ways. The resulting computation is best described therefore as a rooted tree of configurations. Each deterministic computation corresponding to a particular sequence of choices is called a *computation path.*

Example: the existence of a valid shift problem

We are given two strings T and P over $\{0, 1\}$ and we want to test whether there exists a valid shift (i.e. P can be matched in T).

On the next slides we define a deterministic and then a non-deterministic TM that solve this problem. Let's assume that the input is presented in the form "$\S T\#P\ B$".

Deterministic Turing Machine

	0	1	Z	U	§	#	B
q_0	(skp0,Z,R)	(skp1,U,R)			(q_0,§,R)	(fail,#,L)	
skp0	(skp0,0,R)	(skp0,1,R)				(tst0,#,R)	
tst0	(bck,Z,L)	(erase,1,L)	(tst0,Z,R)	(tst0,U,R)			(OK,B,L)
skp1	*Similar to skp0, but the TM is storing a* 1 *instead*						
tst1	"						
bck	(bck,0,L)	(bck,1,L)	(bck,Z,L)	(bck,U,L)		(new,#,L)	
new	(new,0,L)	(new,1,L)	(q_0,Z,R)	(q_0,U,R)			
erase	*Erase any* Z *and* U *from* T *and* P*, start over looking for a matching at the next available shift*						
fail	*Go back to "§", restore the input, get into the failing configuration*						
OK	*Go back to "§", restore the input, get into the final configuration*						

Description. This is a TM implementation of the brute-force matching algorithm we saw few weeks ago.

Non-deterministic Turing Machine

	0	1	Z	U	§	#	B
q_0	(skp0,Z,R)	(skp1,U,R)			(guess,§,R)		
guess	(guess,0,R) | (skp0,Z,R)	(guess,1,R) | (skp1,U,R)				(fail,#,L)	
skp0	(skp0,0,R)	(skp0,1,R)				(tst0,#,R)	
tst0	(bck,Z,L)	(fail,1,L)	(tst0,Z,R)	(tst0,U,R)			(OK,B,L)
skp1	*Similar to skp0, but the TM is storing a* 1 *instead*						
tst1	”						
bck	(bck,0,L)	(bck,1,L)	(bck,Z,L)	(bck,U,L)		(new,#,L)	
new	(new,0,L)	(new,1,L)	(q_0,Z,R)	(q_0,U,R)			
fail	*Go back to “§”, restore the input, get into the failing configuration*						
OK	*Go back to “§”, restore the input, get into the final configuration*						

Description. Guess a shift, by positioning the tape head under one of the characters of T, then check whether P occurs in T with that particular shift only.

Comments. There is no need for an “erase” state anymore!

Issue

What can be said about the power of nondeterminism in resource bounded TM’s?

Are non-deterministic Turing machines, in a sense, more powerful than deterministic ones?

Time bounded machines

Any non-deterministic TM N can be simulated by a deterministic TM T. Furthermore there exists a constant c such that, for any input x, if N accepts[a] x in t steps then T accepts x in, at most, c^t steps.

[a] A (Turing) machine T *accepts* x if T (eventually) halts and says “yes” (i.e. enters an accepting state)! A machine *decides* x if T halts on x and says either “yes” or “no”. A language L over an alphabet Σ is *accepted* (resp. *decided*) by a Turing machine T if for every $x \in \Sigma^*$, T halts and says “yes” if (and only if) $x \in L$.

Given N, we define a deterministic TM T which, for any input x systematically visits the computation tree associated with N on x. The only care is in the fact that the computation tree may contain infinite subtrees so we need to use a breadth first search heuristic to visit the tree.

More details

Without getting to the gory details of T’s transition function, let’s try to understand how this would work by looking at T’s tape (assume all TM’s have just a single tape).

To simplify things, assume all non-deterministic moves of N are binary: in a particular state q, looking at a tape symbol s, N may change s to t_1, move to state q_1 and move to some cell c_1 OR it may change s to t_2, move to state q_2 and to a cell c_2.

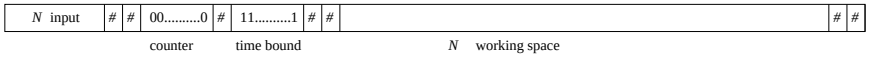
Initializations

Initially, the tape of T contains N's input only. (There is no need to store on the tape the transition function of N as that is hardwired in the transition function of T.)

The TM T knows the time complexity of N, so the first thing that T does is to write on its tape, next to N's input two # symbols followed by $\lceil \log t(n) \rceil$ consecutive zeroes another # and the same number of ones. T will use such space to store a counter that will run from zero to $t(n) - 1$. When the first counter becomes equal to the second one T will know that the simulation can stop.

Also, T knows N cannot use more than $t(n)$ cells, so another mark is placed $t(n)$ cells to the right.

The tape will then look like this:



Deterministic moves

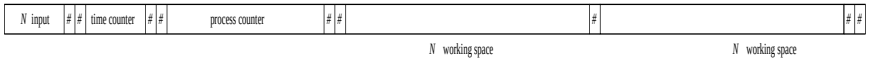
Then (suppose for a while N does not take any non-deterministic move) T simulates N one step at the time, increasing the counter at each step. Anything that N needs to write on the tape is written to the right of the counter and the time bound, within the allocated working space.

Non-deterministic moves

When the first non-deterministic move is to be simulated, T will know that N will make, say, a choice out of two possibilities. This triggers T to duplicate the working space of N and then make each of the possible moves, one in each working space. After this the move counter is updated.

From then on, until the next non-deterministic move, the simulation will proceed in turns in each copy of N working space. A counter (and a bound) of size $t(n)$ is also needed to remember which working space is active at each time instant.

The tape will then look like this (the scale here is much smaller than in the previous picture):



Additional non-deterministic moves with increase the number of copies of N's working tape that need to be handled.

The machine T will accept its input as soon as one computation of N ends in an accepting state.

Complexity analysis. In the worst-case, N may take a non-deterministic move at every step. This will generate a computation tree that could look like a complete binary tree of depth $t(n)$. Such a tree has $2^{t(n)}$ distinct branches, each corresponding to a valid computation of N. To simulate all this T will need time proportional to

$$t(n) \times 2^{t(n)} = 2^{t(n)+\log_2 t(n)} \leq c^{t(n)}$$

for some sufficiently large constant $c > 2$.

End of the story

Nobody knows any significantly better way of simulating a non-deterministic Turing machine!

Space bounded machines

It is quite interesting to discover that the situation changes dramatically if we measure space instead of time.

> (Savitch, 1970) Any one-tape non-deterministic TM which uses at most $s(n)$ tape cells on any input of size n can be simulated by a deterministic TM which uses at most $s(n)^2$ tape cells.

Hence anything we can do in polynomial space with a non-deterministic TM, can be done deterministically too, with a quadratic blow-up in the amount of space required.

Assumptions.

- The function s must be space-constructible function, and it must be $s(n) \geq n$.

Argument

Let N be a nondeterministic Turing machine which works in space $O(s(n))$.

To simplify our description let's assume the following:

1. N is a one-tape TM;
2. each configuration of N can be encoded in exactly $c_1 s(n)$ symbols;
3. immediately before reaching the accepting state N cleans its tape and moves the tape head to cell 0 (so that there is only one initial and final configuration);

For any input x of length n, if N accepts x, then a computation path requiring at most $c_1 \cdot s(n)$ cells must exist.

We also know that the length of such a computation path is at most $2^{c_2 s(n)}$ (with c_2 only depending on N, not on n).

Simulation strategy

As usual we start by giving some details of the simulation strategy. More details will be given next time.

We define a predicate Reach(C_1, C_2, i) which is true if and only if configuration C_2 is reachable from configuration C_1 in at most 2^i steps (i.e. the computation tree of N on x contains a path from C_1 to C_2 of length at most 2^i). Then

$$x \in L \text{ if and only if Reach}(C_0, C_{\text{final}}, c_2 s(n)) \text{ holds.}$$

The simulation will be completely described after we'll have defined a space efficient algorithm that implements the predicate Reach. Such an algorithm will be the main component of the deterministic equivalent to the given NTM N.