CSE 431: Theory of Computation Instructor: Thomas Rothvoß
Lecture 3: Algorithms on Turing Machine

Chapter 3.1: Turing Machines

Turing Machine Example 1:

C = $\{a^i b^j c^k : i * j = k\ ;\ i,j,k \geq 1\}$

Let w be a word where $w_1, w_2, \dots, w_m \in \{a, b, c\}$. We want to determine with a Turing Machine whether w is in language C.

a	a	b	b	b	c	c	c	c	c	c	_	_	_	…

1.) Scan input from left to right testing $w \in a^i b^j c^k$
2.) Cross out the first a (i.e. replace a with ~~a~~ which are both $\in \Gamma$). Move to the first b.
3.) Shuffle between the bs and cs, always crossing out one of each.

~~a~~	a	~~b~~	~~b~~	~~b~~	~~c~~	~~c~~	~~c~~	c	c	c	_	_	_	…
1		2	4	6	3	5	7							

4.) Restore all of the bs, cross out the next a, and repeat
5.) If everything is crossed out at the end, we accept w.

Lemma: Language C is Turing-decidable.

Subtle Note: How can the Turing Machine know if it is at the beginning?
We can write a special mark in the first iteration to recognize the beginning.
$\Gamma' = \Gamma \times \{at\ start, not\ at\ start\}$

Turing Machine Example 2:

$E = \{\#x_1\#x_2\#x_3 \dots \#x_4 : x_n \in \{0,1\}^* \wedge x_i \neq x_j\ \forall\ i \neq j\}$

Where # is a delimiter.

Let the Turing Machine have as input a collection of words separated by hashes and determine if that input would be accepted or not. For sake of this example, we will use words of the same length. If the words are different lengths, they are also detected and accepted with this algorithm, but it is uninteresting to talk about since words with different lengths are always accepted in E.

1.) Place a mark "A" at leftmost #
2.) Scan to the right and place "B" on the next #
3.) Put a dot on first symbols following A and B
4.) By zigzagging, compare both strings that are to the right of A and B

#	1	0	1	#	0	1	0	#	1	1	1	_	_	…
A				B										
	^				^									

5.) If we see no difference, reject.
6.) After both strings are compared, we move B one # to the right and repeat

#	1	0	1	#	0	1	0	#	1	1	1	_	_	…
A								B						
	^								^					

7.) With "B" on last #, we move "A" one # to the right, set "B" to the following # and repeat.

#	1	0	1	#	0	1	0	#	1	1	1	_	_	…
				A				B						
					^				^					

We know that we accept only inputs that have each word different since if we find any two strings that are the same we reject, and we compare every word to every word after it in the sequence by comparing word "A" to each word "B" following it and moving "A" to each word in the input.

Chapter 3.2: Variants of Turing Machines

Multi-Tape Turing Machine:

$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$
- k tapes, where k is an arbitrary constant
- k heads where each can move separately and can independently move/read/write on multiple tapes.

Theorem: Each Multi-Tape Turing Machine has an equivalent single tape Turing Machine.

Proof by picture:

0	1	0	1	_
	^			

a	a	a	-	…
		^		

b	a	_	_	…
^				

K input tapes translated to one input tape, and k many heads remembered as k special markers :

#	0	1	0	1	#	a	a	a	#	b	a	-	-	...
		*						*		*				

1.) Concatenate the k input tapes as one input with a delimiter between the tapes.
2.) For k heads, set up k special marks to remember where the heads are.
3.) To simulate one iteration of the multi-tape, scan the tape to the left, remembering each of the k values where the heads are. Then, make all of the changes the multi-tape TM would by moving back to the right and performing changes.
4.) If and of the k heads needs to move onto the delimiter before the next word of input, shift everything to the right and give that head one extra empty space. This is to simulate the infinite space each of the k input tapes has on the multi-tape TM.

Note: What if k is dependent on input length instead of being an arbitrary constant? Our transition function δ breaks down because the Turing Machine can't depend on input length. Therefore if k is not a constant we can't extend the model we just described.

Non-deterministic Turing Machine:

A non-deterministic Turing machine, in essence, is the same as a deterministic Turing machine, with one key difference: each input gives a set of possible actions.

Formally, this difference is in the transition relation defined by:

$$\delta: Q \times \Gamma \rightarrow \rho(Q \times \Gamma \times \{L, R\}),$$

where Q is the finite set of states, Γ is the tape alphabet, and L and R are moves left and right, respectively.

The machine accepts languages such that there is some path based on the input that will terminate at some accepting state, $q_{accept} \in Q$

Theorem: Each non-deterministic Turing machine has an equivalent deterministic Turing machine.

Basic idea for proof: If we can prove that this is equivalent to a multitape Turing machine, then we can use the previous proof of the equivalence of multitape Turing machines and single tape Turing machines to complete our proof.

Consider a model of δ as a decision tree, in which each node is a state $q \in Q$ and each edge is a decision given by δ based on some input $\sigma \in \Sigma$. In addition, consider the following 3-tape Turing Machine to model the decision tree:

Input: $\boxed{\sigma_0\sigma_1\sigma_2\sigma_3 \ldots}$
Simulation: $\boxed{\alpha_0\alpha_1\alpha_2\alpha_3 \ldots}$

Address: $\boxed{\beta_0\beta_1\beta_2\beta_3 \ldots}$

The input tape is simply the input tape from the original non-deterministic machine. The simulation tape runs a depth-first simulation of the decision tree. The address tape keeps track of the current address of input and simulation, in order to enable backtracking of the tree. The simulation will run as follows:

1. Use this multitape Turing machine to traverse the decision tree of the non-deterministic Turing machine, using the input of the non-deterministic machine as the input of the multitape machine.
2. If at any point during the simulation a leaf is reached (one that must match the appropriate input), accept the input.
3. If the entire tree is traversed without accepting, reject the input.

Note that while this blowup is exponential, the deterministic Turing machine can recognize any language L that can be recognized by its non-deterministic counterpart. It should be mentioned at this point that the Church-Turing machine says nothing of efficiency, but only implies that the Turing machine can recognize anything that any other reasonable model of computation can, and vice versa.

Enumerators: A Turing Machine without an input. Instead of deciding input, enumerators list the set of all words in some language L. In other words, for some language L, it will list all acceptable inputs for a Turing Machine that decides L.

Theorem: A language L is Turing recognizable if and only if there exists an enumerator E for language L.

Proof: Although it may seem inefficient, the proof for this is fairly intuitive:
Suppose that L can be enumerated by enumerator E. Then the following Turing machine can recognize L:

M = "1. Run E. Every time it outputs a word w, compare it with the input w_0 to M.
2. If w is equivalent to w_0, accept w_0"

Now, for the converse: suppose that L can be recognized by Turing Machine M. Further suppose that $s_1, s_2, s_3 \ldots$ is the list of all possible strings in Σ^*. Then we can construct an enumerator E as follows:

E = "1. Repeat the following for i = 1, 2, 3, …
2. Run M for i steps on each input $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding s_j."

This is sufficient to prove the theorem.