Turing Machines

A Turing Machine (TM) is defined by the tuple $(\Sigma, \Gamma, Q, \Delta, q_0, q_A, q_R)$, where
Σ is the input alphabet
Γ is the tape alphabet. $\Gamma = \Sigma \cup \{B\}$, where B is the blank symbol.
Q is a finite set of states
Δ is a transition function

$$\Delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

So Δ maps the tuple (state, symbol) to the tuple (new state, new symbol, head movement left or right).
$\mathbf{q_0}$ is the start state
$\mathbf{q_A}$ is the ACCEPT state
$\mathbf{q_R}$ is the REJECT state

So what can a Turing Machine do?
"All computable functions can be computed on Turing Machines." - Church-Turing Thesis
Example 1: Everything that you can do on your laptop can be done (eventually) on a Turing Machine.
Example 2: We could compute $f(n) = 2n$ or $f(n) = n^2$.

It can be helpful to think of an input tape that is read only and a working tape that acts as memory. There are two resources associated with a Turing Machine: time and space. The time complexity is the number of operations that the turing machine performs before reaching a terminating state (ACCEPT or REJECT state). The space complexity is the number of memory cells that the turing machine uses during the course of its execution.

So does a second tape in the turing machine help? That is, does adding a second tape allow us to compute something that cannot be done on a single tape?

For example, suppose that we have a Turing Machine that will accept a string x iff x is a palindrome. Suppose that we operate on a single tape to determine if x is a palindrome. The head starts at the beginning of the tape and then moves to the end of the tape. The symbol at the end of the tape must match the symbol at the beginning of the tape. We then scan from the next-to-last entry to the second entry, repeating the process. If the length of the input is n, then we require about n^2 operations.

Now suppose that we use two tapes. We could copy the input tape to the second tape and then compare each element in order. This would require time complexity $O(n)$.

In fact, anything that can be computed on a Turing Machine with two tapes can also be computed on a Turing Machine with a single tape.

Nondeterministic Turing Machines

Nondeterministic Turing Machines (NTM) are defined very similar to Deterministic Turing Machines (DTM). In the case of DTMs, the transition function maps each (state, symbol) tuple to a tuple (new state, new symbol, head movement left or right); in the case of NTMs, the transition function maps each (state, symbol) tuple to a set of tuples as follows.

$$\Delta(q, a) = \{(q_1, a_1, \text{L/R}), (q_2, a_2, L/R), ...\}$$

While the computation of a DTM is a sequence of states (i.e., a path), the computation of a NTM can be viewed as a computation <u>tree</u>. An NTM can "guess" a path to an accepting state if one exists and perform the computations along that path. (Note that this is an abstract model. We do not actually have an NTM that we can run). The language associated with an NTM is defined to be the set of strings such that for each such string there exists at least one path that leads to an accepting state.

A Turing Machine M has *time complexity* $t(n)$ if M takes time at most $t(n)$ on any input of length n. A Turing Machine M has *space complexity* $s(n)$ if M uses space at most $s(n)$ on any input of length n.

NTMs can be simulated by DTMs. To simulate a given NTM, apply breadth-first search (BFS) to the NTM's computation tree. Then

$$\text{DTime}(|T|) \text{ captures NTime}(\mathit{depth}(T)).$$

For example, we can simulate an NTM that solves the Factoring problem. Given an integer n, we want to find prime numbers $p_1, \ldots, p_k$ such that $\prod_{j=1}^{k} p_j = n$. We could create a DTM to solve this by having a DTM first divide n by prime numbers $2, 3, \ldots$. Then we would go to the next level of the computation: we would find whether n was divisible by each of the primes and then continue to divide each n/p_i by $2, 3, \ldots$, and so on.

Another example is to find a path from s to t in $LOGSPACE$ (i.e., find a path from s to t using space that is logarithmic in the size of the graph). We will cover this algorithm later in these notes. (Note that LOGSPACE denotes all of the algorithms that can be solved with space $O(\log n)$, where n is the length of the input).

Hierarchy Theorems

So now we ask: does more *time* allow for more powerful Turing Machines? Does more *space* allow for more powerful Turing Machines? Does *nondeterminism* allow for powerful Turing Machines?

Notation

Big-Oh:

$$g = O(f) \text{ if } \exists C > 0 \text{ such that } \forall x \geq x_0, g(x) <= Cf(x).$$

In other words, g is an asymptotic upper bound for f. So when x becomes sufficiently large, we can ignore constant factors between f and g.

Little-Oh:

$$g = o(f) \text{ if } \forall c > 0, \exists x_0 \text{ such that } g(x) < cf(x) \forall x \geq x_0.$$

In other words, no matter what small constant c you choose, g will always be less than f for sufficiently large x.

Space Hierarchy Theorem

A function f is space constructible if there exists a TM M that, on unary input n outputs $f(n)$ (in, say, binary) and uses $O(f(n))$ space. (We also assume that $f(n) > \log n$).

Theorem 1. *For any space constructible function f, there exists a language L accepted by a Turing Machine using $f(n)$ space but not any Turing Machine using $o(f(n))$ space.*

Proof. We will come up with a language L and a turing machine M that decides L using $O(f(n))$ space and show that no turing machine can decide L in $o(f(n))$ space. We describe the language L to be the set of strings accepted by the following turing machine A. (Recall that any TM M has a finite description $< M >$).

1. Check that the input is a valid Turing Machine $< M >$. If the input is not valid, then REJECT.
2. Let n be the length of $< M >$.
3. Mark $f(n)$ spaces. If M ever tries to use more space, then REJECT.
4. Simulate M on $< M >$. If the execution time ever exceeds $2^{f(n)}$, then REJECT.
5. Else, if simulator M accepts, then REJECT. If simulator M rejects, then ACCEPT.

The language L can be recognized by a Turing Machine in space $O(f(n))$. In fact, A is a turing machine that decides L in space $O(f(n))$. (Exercise: Show that each step of H takes $O(f(n))$ space).

Now we show that L is not decidable in $o(f(n))$ space. Assume for contradiction that some Turing Machine H decides L in space $g(n) = o(f(n))$. Consider what happens when the input to H is $< H >$. By assumption, H uses space at most $g(n) = o(f(n))$. Further, since H decides L using space at most $o(f(n))$, it can use at most $2^{o(f(n))}$ time. Thus, H uses space at most $o(f(n))$ and time at most $2^{o(f(n))}$. Therefore, H has to either accept or reject $< H >$.

If simulator H accepts $< H >$, then by the description of L, we have that $< H >$ is not in L. If $< H > \notin L$, then the Turing Machine that recognizes L should reject $< H >$. But since the turing machine H accepted $< H >$, it does not decide L. This is a contradiction.(Exercise: Similarly show a contradiction if H rejects $< H >$). $\square$

So what is the message of the Space Hierarchy Theorem? First, more space allows us to decide <u>strictly</u> more languages. Secondly, we now know that there are languages that have minimum requirements on space to be decidable. So there will be languages that require $O(\log(n))$ space, some that require $O(n)$ space, $O(n^2)$ space, $O(n^3)$ space, etc.

Space Complexity

Suppose an NTM uses $s(n)$ space and $t(n)$ time. We saw that a DTM can do the same computation in $C^{t(n)}$ time. How much space will it need?

<u>Graph reachability:</u> A graph $G = (V, E)$ is a collection of vertices V and edges E, where V is a set of vertices (or nodes) and E is a set of edges from pairs among V. Directed graphs contains directed edges $ij \in E, i \to j$ is a directed edge (or arc) from i to j. Let $n = |V|$. Given a graph G, is there a path from s to t?

For an NTM, we could "guess" the next vertex on a path. The NTM could effectively be performing a breadth-first search on the graph. We would require $O(\log n)$ space to describe a <u>single vertex</u>. For a DTM, if a BFS or depth-first search (DFS) is performed, then the DTM needs to store at most $(n-2)$ intermediate nodes. So the space complexity is $O(n \log(n))$.

Savitch's Theorem

Definition 2. *Let* $f : N \to \mathbb{R}^+$.

$$DSPACE(f(n)) = \{L | L \textit{ is a language decided by } O(f(n)) \textit{ space } DTM\}.$$
$$NSPACE(f(n)) = \{L | L \textit{ is a language decided by an } O(f(n)) \textit{ space } NTM\}.$$

The set $DSPACE(f(n))$ is also called $PSPACE(f(n))$ in many texts and references.

Theorem 3 (Savitch's Theorem)**.**

$$NSPACE(s(n)) \subseteq DSPACE(s^2(n)).$$

Theorem 4. *Graph reachability can be solved in $O(\log^2(n))$ space on a DTM.*

Proof. We want to verify if there exists a path of length k from vertex a to vertex b in Graph G. Define

$$PATH(a,b,k) = \begin{cases} 1 & \text{if there exists a path in } G \text{ of at most } k \text{ steps} \\ 0 & \text{otherwise.} \end{cases}$$

<u>PATH(a, b, k) algorithm</u>

- If $k = 0$, then
  - if $a = b$, ACCEPT
  - else REJECT
- else if $k = 1$, then
  - if $a = b$ or $(a, b) \in t$, ACCEPT
  - else REJECT
- else
  - for every $c \in V \backslash \{a, b\}$: if PATH($a, c, \lceil k/2 \rceil$) accepts AND PATH($c, b, \lceil k/2 \rceil$) accepts, then ACCEPT
  - If we have not accepted, then REJECT

We have to store one node's "name" or label for every level of the recursion. This requires $\log n$ space. The depth of recursion $= \log_2 k \leq \log_2 n$. Therefore the DTM algorithm uses space $O(\log^2 n)$ (i.e., it is in $DSPACE(O(\log^2 n))$). □

For an arbitrary computation, where the turing machine computes a language $L \in$ NSPACE($s(n)$), consider the graph of configurations using $s(n)$ space. Each configuration needs $O(s(n))$ space to name. To test membership of a string in L, we need to verify if there exists a path from the START configuration to the ACCEPT configuration. Applying PATH(START, ACCEPT, k), we find that k only needs to be $C^{s(n)}$ as that is the maximum number of configurations. So $L \in DSPACE(O(s(n) \log k)) = DSPACE(O(s(n)^2))$.

Reading

Sipser, Chapter 3 (Turing Machine Basics),
Section 7.1 (Time Complexity and Big-Oh, Little-Oh notation),
Section 8.1 (Savitch's Theorem, Space Complexity),
Section 8.2 (PSPACE/DSPACE),
Section 9.1 (Space Hierarchy Theorem)