CS 6505: Computability & Algorithms
Lectures for Week 3, January 25-29, 2010

A Turing Machine TM is defined by the following:
$\mathbf{\Sigma}$ is the Input alphabet
$\mathbf{\Gamma}$ is the Tape alphabet. $\Gamma = \Sigma \cup \{B\}$, where B is the blank symbol. $\mathbf{Q}$: set of states
Δ: Transition function
$\Delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
So Δ maps (state, symbol) to (new state, new symbol, head movement left or right)
$\mathbf{q_0}$: start state
$\mathbf{q_A}$: ACCEPT state
$\mathbf{q_R}$: 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. So does a second tape 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 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 around n^2 operations, within a constant.

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 $O(n)$.

Nondeterministic Turing Machines

The following is a list of the finite possible states of a Turing Machine:

$$\Delta(q, a) = \{(q_1, a_1, \text{L/R}), (q_2, a_2, m_2), ...\}$$

While the computation of a Deterministic Turing Machine (DTM) as a sequence of states (i.e., a path), a Nondeterministic Turing Machine (NTM) can be viewed as a computation tree. An NTM can "guess" a path to an accepting state if one exists. (Note that this is an abstract model. We do not actually have an NTM that we can run.)

Suppose that a Turing Machine has input length n. Turing Machines have a *time complexity* $t(n)$ if the Turing Machine takes time at most $t(n)$ on any input of length n. Turing Machines have a *space complexity* $s(n)$ if the Turing Machine uses space at most $s(n)$ on any input of length n.

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

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

.
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, ... 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, ..., 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.)

Space Hierarchy Theorem

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

Big-Oh:

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

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:

$$g = o(f) \rightarrow \nexists c > 0 \text{ such that } \forall x \geq x_0, g(x) \geq cf(x)$$

In other words, no matter what constant c you choose, g will always outstrip f by more than that constant 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 should also assume that $f(n) > \log n$.)

Theorem 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 Any TM M has a finite description $< M >$. Now consider the following table.
Now consider $D = \{< M > |$ If running M on $< M >$ uses space at most $f(n)$ and time $2^{f(n)}$, then M does not accept $< M > \}$
The language D can be recognized by a Turing Machine in space $O(f(n))$.

1. Mark $f(n)$ spaces. If M ever tries to use more space, then REJECTM.
2. Check that the input is valid. If the input is not valid, then REJECT.
3. If the execution time ever exceeds $2^{f(n)}$, then REJECT.
4. Else, if M accepts, then REJECT. If M accepts, then REJECT.

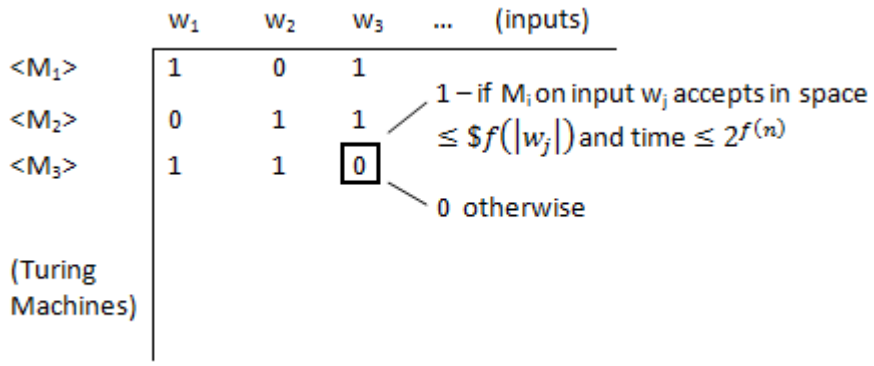

Figure 1: Space Hierarchy Theorem - determine what each Turing Machine accepts and rejects

So D can be recognized by some TM using $O(f(n))$ space.

Claim: No TM using $o(f(n))$ space can recognize D.
Proof: Assume there exists a TM M that recognizes D using $o(f(n))$ space. Running M on input $< M >$, the algorithm (simulator) will finish and use space at most $f(n)$. But M accepts iff M rejects, which is a contradiction. (Add something to clarify the proof here.)

So what do we get out of the Space Hierarchy Theorem? First, we now know that there are algorithms that have minimum requirements for space. So there will be algorithms that require $O(\log(n))$ space, some that require $O(n)$ space, $O(n^2)$ space, $O(n^3)$ space, and so on. We will not be able to get away with just a lower bound on space for all algorithms, either.

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?
Graph reachability: $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges from pairs among V. $ij \in E, i \rightarrow j$ is a directed edge (or arc) from i to j. Let $n = |V|$.

The question is, 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 for a single vertex. For a DTM, if a BFS or depth-first search (DFS) is performed, then the DTM needs up to (n-2) intermediate nodes. So the space complexity is $O(n \log(n))$.

Savitch's Theorem

Definitions Let $f : N \to \mathbb{R}^+$.
$DSPACE(f(n)) = \{L | L$ is a language decided by $O(f(n))$ space DTM $\}$.
$DSPACE$ is also called $PSPACE(f(n))$ in many texts and references.
$NSPACE(f(n)) = \{L | L$ is a language decided by an $O(f(n))$ space NTM $\}$

Theorem $NSPACE(s(n)) \subseteq DSPACE(s^2(n))$
Theorem Graph reachability needs $O(\log^2(n))$ space on a DTM
Proof We want to calculate the path from vertex a to vertex b in Graph G that takes at most k steps (i.e., edges between a and b). Define

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

PATH(a, b, k) algorithm

- 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.

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

For an arbitrary computation, where NSPACE $= s(n)$, consider the graph of configurations using $s(n)$ space. Each configuration needs $O(s(n))$ space to name. Does 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 $DSPACE = O(s(n) \log k) = O(s(n)^2)$.

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)