Lecture 3

Time and Space Complexity

January 13, 1998
Lecturer: Richard Ladner
Notes: Dan Fasulo

3.1. Savitch's Theorem

Recall that in the previous lecture, we defined a reachability function REACH such that, for a given Turing machine T and m-configurations C and D of T, $\mathrm{REACH}(m, C, D, k)$ is true if and only if state D can be reached from state C in at most 2^k steps and using a set of at most m work tape cells. Define $\mathrm{Init}_m(x)$ to be the initial m-configuration for T in which T is at a start state, x is the input string, and m is the number of blank spaces available on the work tape. Thus, for a given Turing maching T and input x,

$$x \in L(T) \Leftrightarrow \mathrm{REACH}(s(|x|), \mathrm{Init}_{s(|x|)}(x), A, c \cdot s(|x|)),$$

where A is an accept state of T and c is a positive integer. We now show that the REACH function can be computed in an efficient amount of space by a deterministic algorithm.

Theorem 3.1 (Savitch):

$$\mathrm{NSPACE}(s(n)) \subseteq \mathrm{SPACE}(s^2(n)).$$

Proof: By our observation above, we only need to show that we can deterministically compute the REACH function in $O(s^2(n))$ space; then we will have a space-efficient algorithm which accepts x if T accepts x. Let $n = |x|$ and assume that we can use m work tape cells for computation. Also assume that we have a representation of T as a directed graph in which the vertices correspond to m-configurations and there is an edge from configuration A to configuration B if and only if B can be reached from A in one step. We now give the algorithm for REACH.

```
/* Decides if D can be reached from C in 2^k time with out exceeding m
   work tape cells. We assume T and x are fixed. */

algorithm reach(m : integer, C, D : m-configuration, k : integer)
{
   if k = 0 then
```

```
      if D reachable from C in 1 step or C = D then
         return true;
      else
         return false;
   else (k > 0)
      for each m-configuration E
         if reach(m, C, E, k-1) and reach(m, E, D, k-1) then
            return true;
      return false;
}
```

We now show that algorithm `reach` requires less than $s^2(n)$ units of storage if both m and k are $O(s(n))$. Observe that each m-configuration contains m units of tape storage, a number of size $\log n$ representing the index into the input string, and a number of size $\log m$ representing the index into the work area. The storage of each instance of the `reach` algorithm uses two such configurations plus $\log k$ storage to represent k. Thus, it takes a total of $O(m + \log n + \log k)$ storage. If we imagine a machine running the `reach` algorithm, there would be at most k instances of the reach function on the machine's stack at any given time. Hence, the total storage required would be $O(k(m + \log n + \log k))$. If we m and k are $O(s(n))$ then because $s(n) \geq \log n$ then the total storage is

$$O(s(n)(s(n) + \log n + \log\log s(n))) = O(s^2(n)).$$

This analysis, however, requires that we know $s(n)$ so that we can set m and k to be the appropriate size. If $s(n)$ is not known but the input x is guaranteed to be in the set, then we can simply let $m = 1, 2, \ldots,$ until a value which works is found. However, if x is not necessarily in the set, then a more sophisticated approach is required. We define a new algorithm `safe` which determines if a given amount of storage is sufficient for the computation, regardless of whether the input is accepted or not. It can be seen that `safe` requires approximately the same amount of storage as `reach`.

```
/* Determines if T, starting in configuration C, does not
   exceed m tape cells in <= 2^k moves. */

algorithm safe(m : integer, C : m-configuration, k : integer)
{
   if k = 0 then
      if in one move of T from C, T exceeds the workspace then
         return false;
      else
         return true;
   else
      return (safe(m, C, k-1) and (for all m-configurations D)
              (reach(m, C, D, k-1) implies safe(m, D, k-1)));
}
```

We can then use the `safe` and `reach` algorithms to give a deterministic procedure for determining if $x \in L(T)$ in a space-efficient manner:

```
/* Deterministically simulates NTM T on input x in a space-
   efficient manner. */

algorithm deterministic_T(x : string)
{
   m : integer;
   enough_space : boolean;

   m = 0;
   enough_space = false;

   repeat
      m = m + 1;
      enough_space = enough_space or safe(m, Init(m,x), c*m);
   until enough_space;

   for some accepting m-configurations A
      if reach(Init(m,x), A, c * m) then
         return true;

   return false;
}
```

□

3.2. Linear Speed-up

We begin by presenting two results that show that one can always construct a faster or more space efficient Turing machine for a given problem, at the cost of creating a significantly larger machine. The first two results are known as the *Linear Speed-up Theorems*.

Theorem 3.2 (Linear Speed-up): For $\epsilon > 0$

$$\begin{aligned}
\text{[N]SPACE}(s(n)) &= \text{[N]SPACE}(\epsilon \cdot s(n)), \\
\text{[N]TIME}(t(n)) &= \text{[N]TIME}(n + 2 + \epsilon \cdot t(n)).
\end{aligned}$$

The idea behind the proof is that given T, we can create a new machine T' such that each tape cell of T' encodes m tape cells in T. This obviously reduces the number of cells required to store the information. It also leads to a speedup in the machine. We create a transition function for T' that simulate T for at least m move in just 8 moves of T'. In four moves of T', it accumulate

$3m$ characters of T where the head of T is in the middle third of the characters. In 0 moves of T' it computes what would happen to these characters when the head of T would move off of them. This would simulate at least 8 moves of T. Finally, T' restores the new $3m$ characters to the 3 cells of T' and move the head of T' to be under where the head of T is. This restoration takes another 4 moves of T'.

3.3. Tape Reduction

Let $[\mathrm{N}]\mathrm{SPACE}_k(s(n))$ be the class of problems which can be solved in [non]deterministic space $s(n)$ by a machine with k work tapes, and, analogously, $[\mathrm{N}]\mathrm{TIME}_k(t(n))$ be the class of problems which can be solved in [non]deterministic time $t(n)$ by a k-tape machine.

Theorem 3.3 (Tape Reduction to One Tape): If $t(n) > n$ then

$$\begin{aligned} [\mathrm{N}]\mathrm{SPACE}_k(s(n)) &= [\mathrm{N}]\mathrm{SPACE}_1(s(n)), \\ [\mathrm{N}]\mathrm{TIME}_k(t(n)) &= [\mathrm{N}]\mathrm{TIME}_1(t^2(n)). \end{aligned}$$

The idea behind of the proof of this theorem is that the characters on each tape at index i in the multi-tape machine can be grouped into a single character on a 1-tape machine. The head position for each tape is also encoded in the characters of the 1-tape machine. In this way the storage stays the same, but the time will grow quadratically. Each may take as many at $t(n)$ steps of 1-tape machine to simulate one step of the multi-tape machine. It takes that long to find out what is under the heads and update their contents.

There is a very clever construction that yields a better time simulation if the simulating machine is allowed two tapes instead of one.

Theorem 3.4 (Tape Reduction to Two Tapes): If $t(n) > n$ then

$$[\mathrm{N}]\mathrm{TIME}_k(t(n)) = [\mathrm{N}]\mathrm{TIME}_2(t(n)\log t(n)).$$

3.4. Hierarchy

Finally, we begin looking at hierarchy theorems that demonstrate that complexity classes get truly larger if more and more space or time is allowed for computing. We say that $s(n)$ is *tapeconstructible* if and only if there exists T such that T uses exactly $s(n)$ tape cells on an input a^n and then halts.

Theorem 3.5 (Space-Hierarchy): If $s_2(n)$ is tape constructible, $\lim_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$, and $s_1(n)$ and $s_2(n)$ are both greater than $\log_2 n$, then

$$\mathrm{SPACE}(s_1(n)) \subset \mathrm{SPACE}(s_2(n))^1.$$

[1]We reserve the symbol "$\subset$" to mean proper subset and the symbol "$\subseteq$" for plain subset.

Interestingly there is a gap theorem that demonstrates that some pathological behavior can happen if space bounds are not tape constructible.

Theorem 3.6 (Gap): For some computable function $s(n)$,

$$\mathrm{SPACE}(s(n)) = \mathrm{SPACE}(2^{s(n)}).$$