University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Glenn Sun
Date: October 8, 2020

LECTURE

3

The Universal Turing Machine and Introduction to Nondeterminism

This lecture is divided into two parts, which are largely separate topics. For the first part, recall that a k-tape Turing machine can be simulated by a single-tape Turing machine with a quadratic blowup in time, and this bound is tight. A natural question to ask now is whether one can achieve a smaller blowup with a two-tape Turing machine. The answer is yes, and in fact we can achieve a logarithmic factor blowup, which is all but negligible.

As this simulation wraps up our discussion on deterministic computation, in the second part of this lecture, we turn our attention to the class of languages called NP. Although we will see in future lectures how NP uses an important concept called nondeterminism, we opt for a more usable certificate-verifier definition in this lecture's introduction. We end with some examples of problems in the class NP.

3.1 Simulation on a two-tape Turing machine

Recall the port problem, in which you are trying to unload n pieces of cargo onto an infinitely long pier, but the value of n is not revealed to you beforehand. The pier is divided into cells, each of which can hold one piece of cargo. Every time you return to the end of the pier, you are given a new piece of cargo unless there are none left. You can carry an unlimited amount of cargo at any given time. The goal is to minimize the number of steps you need to take in the worst case.

A trivial solution is to put the first piece of cargo in the first cell, the second in the second, and so on. This takes $O(n^2)$ time, and this solution does not use the fact that you can carry an unlimited amount of cargo. To improve on this solution, we periodically pick up the cargo near the pier's end and move all of it far away in one trip, and the larger the pile being moved, the further away it goes. This uses far less time because expensive movements are only made once in a long time, and indeed one can perform the analysis and conclude that this solution takes $O(n \log n)$ time. For more precise details, please see the previous lecture.

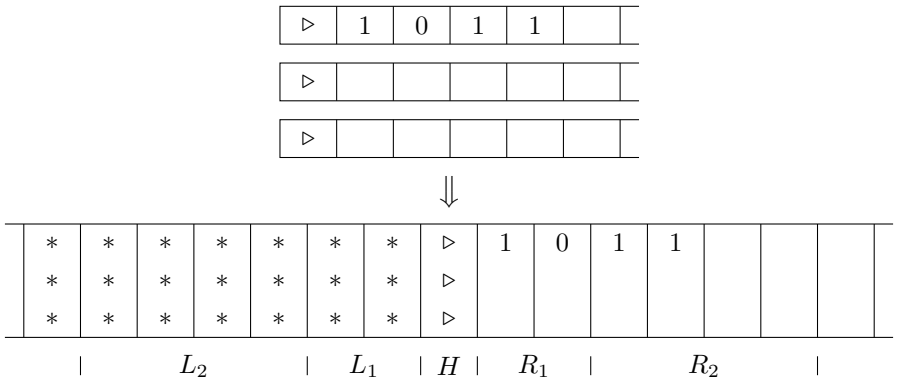
This problem is surprisingly analogous to simulating k-tape Turing machines on smaller Turing machines. If the k-tape Turing machine takes $T(n)$ time, then it was not difficult to show that a single-tape Turing machine can simulate it in $O(T(n)^2)$ time. The addition of a

second tape allows us to have something similar to carrying an unbounded amount of cargo, and will allow us to achieve simulation in $O(T(n) \log T(n))$ time. The pier is the first tape, and the cargo is the bits that need to be written as governed by the transition function.

To avoid the proof from becoming too long, motivations for the choices being made are deferred to a list of "footnotes" that appear after the proof.

THEOREM 3.1 ([2]). *Let $f : \{0,1\}^* \to \{0,1\}^*$ be computable in time $T(n)$ on a k-tape Turing machine. Then $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $O(T(n) \log T(n))$ on a two-tape Turing machine.*

Proof. Without loss of generality, let us use bidirectional tapes[1]. Divide the first tape into zones $\ldots, L_3, L_2, L_1, H, R_1, R_2, R_3, \ldots$, where the number of cells in each zone is $|L_k| = |R_k| = 2^k$ and $|H| = 1$. This tape will simply contain the entire contents of the k-tape Turing machine stacked vertically, possibly with additional whitespace, so if the original alphabet was Γ, the new alphabet is $(\Gamma \cup \{*\})^k$, where $*$ is the new whitespace character. Naturally, zones $L_1, L_2, \ldots$ are all initialized to $(*, \ldots, *)$ and zones $H, R_1, R_2, \ldots$ are initialized to tuples in Γ^k, as shown in the diagram below.



Then for every step of the original computation, the two-tape Turing machine does the following:

1. Read the cell in zone H and store the intended transition in state.
2. For each subtape, if the read/write head was supposed to move left, move the cells around zone H to the right instead, so that the read/write head ends up staying in zone H. Similarly for the other direction.[2]
3. Write to the cell in zone H.

It just remains to show how to do step 2 in, on average, $O(\log T(n))$ steps for each subtape. Fix a subtape, and call any of its cells *empty* if it contains the new whitespace character $*$, and *full* if it contains any character from the original alphabet. Hence, zones $L_1, L_2, \ldots$ are initially all empty and zones $R_1, R_2, \ldots$ are initially all full.

Let us prove the following: Suppose for each pair (L_i, R_i) that either L_i is empty and R_i is full, or vice versa, or both are half full[3]. Then there exists a process that preserves the order of cells, moves the nearest full cell to zone H into zone H (for either direction), and preserves the related emptiness property above.

Suppose that we are moving cells right (left is similar), and the cell that we want to move into H is currently in L_i. Then from our assumption, L_i is either half full or full and the remaining $L_1, \ldots, L_{i-1}$ are all empty, and we can further deduce the status of $R_1, \ldots, R_i$, so we may define the shift as follows:

	L_i	L_{i-1}	$\cdots$	L_1	H	R_1	$\cdots$	R_{i-1}	R_i
old	half/full	empty	$\cdots$	empty	full	full	$\cdots$	full	empty/half
new	empty/half	half	$\cdots$	half	full	half	$\cdots$	half	half/full

The shift is performed using the second tape (that we haven't used so far) as a queue, and furthermore this indeed preserves the order of cells. Note that this also correctly moves the desired cell into zone H, using the fact that $1 + \sum_{k=1}^{n} 2^k = 2^{n+1}$. Finally, it is clear that this preserves the way that L_i and R_i's emptiness are related.

The above gives an algorithm and proves it correctness, so it remains to analyze the running time. Each shift from L_i (or R_i) uses $O(2^i)$ time because the amount of cells being modified is approximately $4 \cdot 2^i$, and we run through it a constant number of times to load and unload the queue. However, because such a shift required $L_1, \ldots, L_{i-1}$ to all be empty, there must be at least $|L_1| + \cdots + |L_{i-1}| = O(2^i)$ other shifts performed before this shift is performed again, so at most an $O(\frac{1}{2^i})$-fraction of shifts are of this type. Hence, shifts from each L_i or R_i contribute only a constant to the average running time. There are a total of $2\lfloor \log T(n) \rfloor$ zones that a shift can start at, hence on average each shift takes $O(\log T(n))$ running time as desired. □

This is a long proof, so we will run through it again with an example. Before that, as promised, below are motivations and comments for the various choices being made:

1. We chose to use a bidirectional tape because read/write heads are allowed to move both left and right. We want to move tapes instead of heads, so we need a bidirectional tape to move these tapes.

2. One might be curious why we choose to move tapes instead of moving and tracking read/write heads, as we did in the single-tape simulation. This change allows us to use our movement more efficiently, as you saw by leaving empty space. Furthermore, this approach would not work without a second tape, because we would need to run back and forth in order to perform the shifts, costing much more time.

3. In the port problem, we never needed a zone to be half full. In fact, recall that all of the zones were always either completely full or completely empty. But this all-or-nothing approach can be difficult to use when we are in general not sure which direction the next move is. If we ever moved all boxes far away, it would take a long time to move one back. Hence, for symmetry, we ask that zones on both sides be half full so that there is ample space for shifting on both sides.

Example 3.2. Recall the palindrome Turing machine, and let us test the palindrome 1001. The first tape of the two-tape simulation starts as follows:

*	*	*	*	*	*	*	▷	1	0	0	1				
*	*	*	*	*	*	*	▷								
*	*	*	*	*	*	*	▷								
	L_2				L_1		H	R_1			R_2				

The first transition for every subtape is to move the read/write head to the right and copy the input onto the work tape, so instead we move the entire tape to the left before copying. The cell that we want to move is in R_1, so we only affect L_1, H, R_1. For half filled zones, it does not matter which half is filled, we will always just fill the right half for a consistent picture.

*	*	*	*	*	*	▷	1	*	0	0	1			
*	*	*	*	*	*	▷	1	*						
*	*	*	*	*	*	▷		*						
L_2				L_1		H	R_1		R_2					

The next transition is similar, but the output tape does not move its read/write head.

*	*	*	*	*	▷	1	0	*	*	0	1			
*	*	*	*	*	▷	1	0	*	*					
*	*	*	*	*	*	▷		*						
L_2				L_1		H	R_1		R_2					

For the next transition, the cell we want to move is now in R_2, so we perform a much bigger shift. Since R_2 was full to begin with, now every zone depicted is half full.

*	*	*	▷	1	*	0	0	*	1	*	*			*
*	*	*	▷	1	*	0	0	*		*	*			*
*	*	*	*	*	*	▷		*						
L_2				L_1		H	R_1		R_2					

The next transition is again a small transition only involving L_1, H, R_1.

*	*	*	▷	1	0	0	1	*	*	*	*			*
*	*	*	▷	1	0	0	1	*	*	*	*			*
*	*	*	*	*	*	▷		*						
L_2				L_1		H	R_1		R_2					

The input tape now begins to move in the opposite direction, and since the desired cell is currently in L_1, we again only involve L_1, H, R_1.

*	*	*	▷	1	*	0	0	*	1	*	*			*
*	*	*	▷	1	0	0	1	*	*	*	*			*
*	*	*	*	*	*	▷		*						
L_2				L_1		H	R_1		R_2					

It should now be clear how the remaining computation proceeds, and it should also give an intuitive feel for why this procedure spends so much time performing small transitions that there is only a logarithmic-size time blowup.

This concludes our discussion of deterministic Turing machine computation. One main takeaway from the past few lectures is that the definition of a Turing machine is quite resilient to changes. Regardless whether we change the alphabet, give bidirectionality, or give additional or fewer tapes, important classes of problems (languages) such as P remain the same. As we move forward, we'll be free to choose whatever model of computation is most convenient.

3.2 Introduction to nondeterminism

Nondeterminism is one modification to Turing machines that does significantly change the problems that can be solved in polynomial time. In this short introduction to nondeterminism and the class NP, we will defer the definition of nondeterministic Turing machines for a later time, in order to have a more intuitive feel of what the class NP is. (Contrary to popular misbelief, the N in NP stands for "nondeterministic", not "not".)

The intuitive definition that we will use this lecture is that a language is in NP if a correct answer can be *checked* in polynomial time. We are not really concerned with how long it takes to solve problems, but if for example

Definition 3.3 (NP). A language L is in the set NP if there exists a polynomial time computable function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ such that $x \in L$ if and only if there exists a string $u \in \{0,1\}^*$, called the certificate, such that $f(x,u) = 1$. A Turing machine that can compute f is called a verifier.

One fine detail is that from here on, we may not necessarily write languages as subsets of $\{0,1\}^*$. Since almost any mathematical object can be encoded into $\{0,1\}^*$ in a fairly natural way, we will opt for defining languages as generic (at most countably infinite) sets. Furthermore, when we talk about graphs or other objects that may be "too large to form a set" under ZFC and hence don't have encodings into $\{0,1\}^*$, we may always assume that graphs are taken with vertex set $V \subseteq \mathbb{N}$, so that the set of all graphs considered is at most countably infinite.

As we discuss more complicated languages, one more key thing to note is that an integer $n \in \mathbb{Z}$ by itself is an input of size $O(\log n)$, considering binary representations of numbers. On the other hand, a graph on n vertices is indeed an input of size $O(n)$ (or more precisely $O(n \log n)$, but this difference is negligible to us).

Example 3.4. Consider the following languages:

1. Independent Set $= \{(G,k)$: there exists $S \in V(G)$ such that for all $u, v \in S$, $\{u,v\} \notin E(G)$, and $|S| = k\}$. Such a subset is called an independent set.
2. Composite $= \{m \in \mathbb{Z} : m$ is composite$\}$.
3. Subset Sum $= \{a_1, \ldots, a_n, t \in \mathbb{Z} :$ there exists $S \subseteq \{a_1, \ldots, a_n\}$ such that $\sum_{a \in S} = t\}$.
4. Factoring $= \{m, a, b \in \mathbb{Z} : m$ has a factor between a and $b\}$.
5. Traveling Salesman Problem $= \{(G,k)$: the weighted graph G has a tour of length $k\}$.
6. Graph Isomorphism $= \{(G, G')$: there exists a bijection between $V(G)$ and $V(G')$ that preserves the edge relation $\}$. Such a bijection is a called an isomorphism.

7. LINEAR PROGRAMMING = { system of linear inequalities over $\mathbb{Q}$: there exists a solution $x_1, \ldots, x_n \in \mathbb{Q}$ satisfying all of them }.

All of these languages belong to the class NP. One can see that the following suffice as certificates:

1. A list of the vertices in the independent set. The existence of edges between them can be checked in $O(n^2)$ time.
2. Two factors of m that multiply to give m. Schoolbook multiplication runs in polynomial time.
3. The subset S. Schoolbook addition runs in polynomial time.
4. A factor in the interval. Schoolbook division runs in polynomial time.
5. The list of edges used in the tour. One simply needs to follow it, check that edges are valid, and that all vertices are used.
6. The bijection between vertex sets. One simply needs $O(n^2)$ time to check that the edge set is exactly preserved.
7. The solution vector $(x_1, \ldots, x_n) \in \mathbb{Q}$. One can simply plug in the numbers to and compare.

To be more specific for one of these examples, take the independent set problem. Let $f(G, k, S) = 1$ if S is an independent set of size k in G, and 0 otherwise. The function f clearly satisfies the condition that $(G, k) \in$ INDEPENDENT SET if and only if there exists S such that $f(G, k, S) = 1$. Then as outlined previously, there exists a Turing machine that computes f is $O(n^2)$ time, so INDEPENDENT SET $\in$ NP.

Although these languages are easy to check, it is probably already clear that these are not as easy to compute. In fact, there are no known polynomial time algorithms for any of them except composite and linear programming. The polynomial time algorithm for solving composite was shown recently, in 2004 [1]. In the next lecture, we'll explore more about these languages and how a modification on Turing machines allows us an equivalent definition of this class of problems.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of Mathematics*, 160, 09 2002.

[2] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape turing machines. *J. ACM*, 13(4):533–546, Oct. 1966.