ECS 120 Lesson 23 – The Class $\mathcal{P}$

Oliver Kreylos

Wednesday, May 23th, 2001

We saw last time how to analyze the time complexity of Turing Machines, and how to classify languages into complexity classes. We also saw that the complexity class of a language depends on the underlying model of computation. Today we are going to investigate the relationship between different models, and how this relationship leads to the definition of the class of *tractable problems*.

1 Time Complexity Relationship Between Computation Models

In general, the time complexities of languages depend on the underlying computation model. But as it turns out, they do not differ too much: As long as only deterministic models are considered, the running times are related by polynomials:

Theorem 1 (Time Complexity Equivalence) *Let $f: \mathbf{N} \to \mathbf{N}$ be a function, with $f(n) \geq n$. Then every $f(n)$ time multitape Turing Machine has an equivalent single-tape Turing Machine that runs in time $O\big(t^2(n)\big)$.*

This theorem can be proved easily by analyzing the algorithm that simulates a multitape Turing Machine using a single-tape machine: For each step the multi-tape machine M performs, the single-tape simulator S has to perform up to $t_M(n)$ steps. In total, this means that the running-time of S is $t_S(n) = t_M(n) \cdot O\big(t_M(n)\big) = O\big(t_M^2(n)\big)$. In the case that the running-time of M is bounded by a polynomial, say $O(n^k)$, the running-time of S is also bounded by a polynomial: $O\big((n^k)^2\big) = O(n^{2k})$. This *polynomial relationship* between multitape and single-tape Turing Machines also holds for

other (deterministic) machine classes. Without proof, we state the following theorem:

Theorem 2 (Polynomial Equivalence of Deterministic Models) *If a problem P is decided in polynomial time by some reasonable model of deterministic computation M_1, it is also decided in polynomial time by any other reasonable model of deterministic computation M_2. In other words, if $t_{M_1}(n) = O(n^{k_1})$, then $t_{M_2}(n) = O(n^{k_2})$ for constants k_1, k_2.*

This theorem shows that deterministic models of computation are roughly equivalent to each other in terms of running-time. What about non-deterministic models? Are they also related by polynomial factors? Before we can answer this question, we have to define running-time for a non-deterministic model, for example non-deterministic Turing Machines. Going back to complexity analysis for deterministic Turing Machines, we defined running-time as the maximum number of steps a machine performs on an input of given size. In other words, we defined running-time as the maximum length of all computation chains for a word of length n. In a deterministic machine, there is exactly one computation chain for every word, as shown in the upper part of Figure 1. In non-deterministic computation, however, there are multiple chains, because computation can branch at each step, see the lower part of Figure 1. In order to define running-time for non-deterministic machines in a way that it is a generalization of deterministic running-time, we define it as the length of the longest branch in all computation trees for words of length n. If a deterministic machine is interpreted as a non-deterministic one that just happens to never branch, the two definitions match exactly, as intended.

1.1 Complexity of Simulation Algorithm for NTMs

In order to relate the complexity of a non-deterministic Turing Machine to a deterministic one, we have to analyze the time complexity of the algorithm that simulates an NTM using a DTM given in the textbook. The basic idea of this algorithm is to explore the computation tree of an NTM using a breadth-first strategy. Here is a high-level outline of the simulation:

Algorithm NTM $\rightarrow$ DTM On input $\langle M, w \rangle$, where M is an NTM and w is an input word,

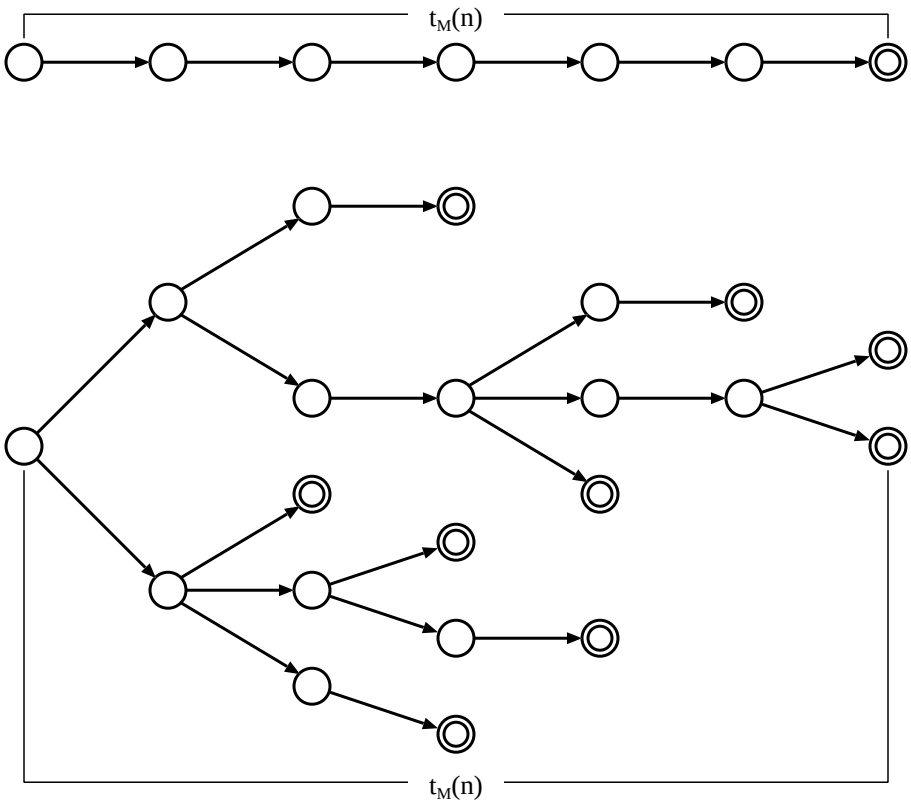

Figure 1: Definition of running-time $t_M(n)$ for deterministic (upper part) and non-deterministic (lower part) machines. The definition for non-deterministic computation is compatible to the deterministic one.

1. Write the initial configuration of M onto the tape.
2. Scan the first configuration on the tape, and determine whether it is an accepting configuration. If it is, accept.
3. Otherwise, if it is not a rejecting configuration, determine which other configurations M can be reached from it. Write all those configurations to the end of the tape.
4. Cross out the first configuration on the tape.
5. If there are configurations left on the tape, repeat from step 2. Otherwise, reject.

In algorithm lingo, this machine puts all possible configurations M could go through into a queue, and performs all possible computation steps on the head of the queue and puts the resulting configurations back into the queue. Altogether, this algorithm will explore all possible configurations in the order of their depth in the computation tree: First the initial one (depth 0), then all the ones reachable in one step (depth 1), then all of depth 2, and so on.

Each of the steps in the algorithm can be performed in a "reasonable" amount of time: If the running-time of machine M is $t_M(n)$, this means that any branch of its computation is at most $t_M(n)$ steps long. This, in turn, means that the maximal length of any configuration machine S has to deal with is $O\big(n+t_M(n)\big)$ – even though a Turing Machine can write an arbitrary long word onto its tape, it can write at most n characters in n steps. Thus, the steps of checking a configuration, crossing out a configuration and writing a configuration are all bounded by $O\big(t_M(n)\big)$, and the total running-time of S is bounded by the number of these steps it has to perform. To find a bound for that number, we have to consider the structure of computation trees for NTMs.

For any NTM M, the number of configurations it can branch to from any configuration is limited by a constant b: Since both the set of states and the tape alphabet are finite, there is only a finite number of possible transition arrays going out of each state. This means that any computation tree for M is a b-ary tree (a tree where each internal node has at most b children), where b is a constant depending on M. The simulating Turing Machine S must visit all nodes in the computation tree, so the question is how many nodes a computation tree can have if the length of its longest branch is $t_M(n)$, i. e.,

if the tree's depth is $t_M(n)$. Theory of b-ary trees tells us that the number of nodes in a b-ary tree of depth $t_M(n)$ is at most $b^{t_M(n)+1}-1$, which is then also an upper bound of the number of simulation steps S has to perform. Together with the fact that each step is bounded by $O\big(t_M(n)\big)$, the total running-time of S is $t_S = \big(b^{t_M(n)+1}-1\big) \cdot O\big(t_M(n)\big) = O\big(b^{t_M(n)+1}\big) = 2^{O(t_M(n))}$.

This analysis shows that non-deterministic and deterministic models of computation are *not* related by polynomial factors – changing from an NTM to a DTM increases the running-time exponentially. It is interesting to compare this result with the one we have seen for NFAs and DFAs, see Table 1. When converting an NFA to a DFA, the number of states increases exponentially, but the running-time stays the same: For a FSM, the number of steps is always the number of input characters. For a TM, on the other hand, the number of states stays roughly the same, but the running-time increases exponentially.

		non-deterministic	deterministic
FSM	# states	k	2^k
	time	n	n
TM	# states	k	$O(k)$
	time	$t_M(n)$	$2^{O(t_M(n))}$

Table 1: Growth in number of states and running-time when converting from a non-deterministic machine to its deterministic equivalent.

2 The Class $\mathcal{P}$

As we have seen, the running-time of an algorithm depends on the underlying model of computation. But as long as only deterministic models are considered, the running-times only differ by polynomials: If a problem can be decided in polynomial time by one model, it can be decided in polynomial time by any other (reasonable) model. This property makes the class of all problems that can be decided in polynomial time special: The membership of a problem in this class does not depend on the particular machine model used. They are also special in another respect: They roughly coincide with

the class of problems that can be solved in a "reasonable" amount of time. We make the following definition:

Definition 1 (The Class $\mathcal{P}$) *The class $\mathcal{P}$ is the set of all problems that can be decided by some deterministic model of computation in polynomial time. In other words,*

$$\mathcal{P} = \bigcup_{k \geq 0} \text{TIME}(n^k)$$

We also call $\mathcal{P}$ the set of tractable problems.

From now on, we will focus on the question whether a problem is in $\mathcal{P}$ or not, instead of determining its exact time complexity. This might seem like an outrageous simplification, and in fact most programmers spend most of their time on squeezing just a little linear-factor improvement out of their programs, but the question "in $\mathcal{P}$ or not in $\mathcal{P}$" is nonetheless interesting. We will just leave the at least as interesting question of exact complexity to an algorithms course. To see that the difference between problems in $\mathcal{P}$ and problems not in $\mathcal{P}$ is in fact bigger than the difference between problems in $\mathcal{P}$, let us consider the following example: Say that we have a program solving a certain problem P that runs for one second on an input of size n. We now want to know how long the program would run on an input of length $2n$. The following Table 2 lists the estimated running-times for different complexities of P:

Complexity	time for input size $2n$
$O(n)$	$O(2n) = 2 \cdot O(n) \approx 2\,\text{s}$
$O(n^2)$	$O\big((2n)^2\big) = O(4n^2) \approx 4\,\text{s}$
$O(n^{10})$	$O\big((2n)^{10}\big) = O(1024n^{10}) \approx 1024\,\text{s}$
$2^{O(n)}$	$2^{O(2n)} = 2^{2 \cdot O(n)} = \big(2^{O(n)}\big)^2 = 30$ years

Table 2: Growth in running-time for a problem P that takes one second for input size n, depending on the time complexity of P.

To understand the "weird" result in the last table row, we have to go back to our definition of running-time. From the arithmetics, we know we must square the running-time, but when squaring one second, we end up with one

second as result, which is intuitively wrong. To get the correct result, we must square the number of steps the program performed, not the number of seconds it ran (we defined running-time in terms of steps). The number of steps a program performs in a second obviously depends on the type of computer used, so let us assume we used a modern desktop PC, with a CPU running at 1.33 Mhz. Also assuming that the program runs completely in first-level cache, and that there are few pipeline stalls, we can estimate that the program performed about one billion steps (10^9) in one second. Thus, on input size $2n$, the program will run for one billion billion steps (10^{18}), which, at a rate of one billion steps per second, would take a billion seconds or roughly thirty years. This little calculation shows that ignoring polynomial differences in running-time, though being overly sloppy, still yields interesting results.

3 Polynomial Complexity Analysis

If we only want to determine whether a given problem can be decided in polynomial time or not, we can be even more sloppy than in our original time complexity analysis. Typically, a problem is solved by stating a high-level algorithm, as we have done before. These algorithms are a list of steps, including loops, recursion, and other forms of repetition. In order to prove that an algorithm runs in polynomial time, we have to show two things:

1. The total number of steps performed by the algorithm is bounded by a polynomial $O(n^{k_1})$, and

2. the running-time for each single step, using some reasonable deterministic model of computation, is bounded by a polynomial $O(n^{k_2})$.

If both properties are satisfied, the total running-time is bounded by $O(n^{k_1}) \cdot O(n^{k_2}) = O(n^{k_1+k_2})$, another polynomial. Let us consider an example problem in $\mathcal{P}$, and how we prove its tractability using polynomial complexity analysis.

3.1 Finding Paths in a Graph

An important problem in graph theory is whether two nodes are *connected*. More precisely, if given a (directed) graph $G = (V, E)$ as a set V of nodes

and a set $E \subset V \times V$ of edges, and two nodes $s, t \in V$, the question is whether there exists a sequence of nodes $s = v_0, v_1, \ldots, v_n = t$, such that for all $i = 1, 2, \ldots, n : (v_{i-1}, v_i) \in E$. In other words, the question asks about the existence of a path leading from s to t, see Figure 2. This problem arises quite frequently; for example, it can be used to decide the emptiness of a finite automaton's language: If M is a finite state machine, M's language is empty if and only if there exists no path from the start state to any final state in M's transition diagram.

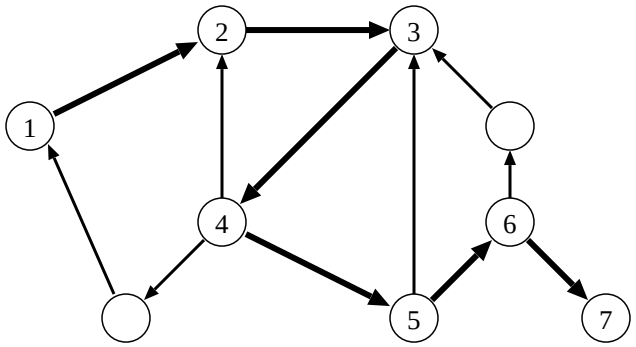

Figure 2: A directed graph G and a path from one node to another (bold arrows). The numbers denote the order of nodes in the path.

Our first algorithm to decide the path finding problem is based on *brute force*, by trying all possible sequences of nodes leading from s to t, and checking if the sequence forms a valid path in G. We first observe that, if there is a path from s to t at all, there is one using at most $n := |V|$ nodes. If there is a longer path, by the Pigeonhole Principle there must be a node occuring twice, implying a loop. Since we are only looking for paths, loops need not be taken, so the path can be made shorter by removing the loop. With this restriction, it becomes possible to exhaustively search for paths: Generate all possible sequences $(v_1, \ldots, v_k)$ of k states, for $k = 1, 2, \ldots, n$, and check if any of them is a path. The checking can be done in $O(k)$ time – by checking that $v_1 = s$, $v_k = t$, and that for each $i = 2, 3, \ldots, k : (v_{i-1}, v_i) \in E$. The total number N of sequences of length up to and including n is given by $N = n^{n+1} - 2$; multypling by the checking complexity yields a total running-time of $t_M(n) = O(k) \cdot (n^{n+1} - 2) = O(n^{n+2})$. This function is definitely not polynomial, so we have to go back to work and find a better algorithm. That second approach works as follows:

Algorithm Path in Graph On input $\langle V, E, s, t\rangle$, where V is a set of nodes, $E \subset V \times V$ is a set of edges and $s, t \in V$ are two nodes,

1. Put s into a queue of nodes, and mark it.
2. Retrieve and remove the first element v from the queue.
3. Find all nodes w directly reachable from v by finding all pairs $(v, w) \in E$. If a node w is not yet marked, mark it and add it to the end of the node queue.
4. As long as there are nodes in the queue, repeat from step 2.
5. If node t is marked, accept; otherwise, reject.

This algorithm works by searching all possible paths starting at node s in a breadth-first manner: It first puts s into a queue, then visits all nodes reachable from s in a single step, then visits all nodes reachable in two steps, and so on. If node t is visited at some point, the algorithm will accept. Now let us analyze the algorithm's complexity. To show it is in $\mathcal{P}$, we only have to show that both the total number of steps and the complexity of each step are polynomial in input size. Let us consider the individual steps first:

- Step 1 can be executed in time $O(n)$. It involves finding node s in the graph's encoding, marking it, and writing it to an unused portion of the tape thereby forming the node queue.
- Step 2 can be done in polynomial time as well, by moving to the start of the node queue, retrieving the node and removing it from the queue.
- Step 3 involves scanning the encoding of E to find any node reachable from v. Each of these nodes must be marked, and must possibly be added to the node queue. Altogether, this can be performed in polynomial time.
- Step 5 only has to check node t for a mark, and accept/reject based on the case. This is polynomial time as well.

The remaining question is to bound the total number of steps executed, in other words, to bound the number of executions of the main loop based on step 4. At first glance, it seems the number of loop executions can not be

bounded by a polynomial: Every node can have up to n direct neighbours, so whenever a node is removed, up to n nodes can be added, growing the queue exponentially. Upon closer inspection, it turns out that a node is only added to the queue when it is not marked, and it is marked when added to the queue. In other word, every node can be added at most once. Since there are n nodes, the loop can be executed at most n times. This means that the total number of steps executed is also polynomial, showing that the problem is in $\mathcal{P}$.