ICS 260—Fall, 2000–Class Notes 11: Turing Machines, Non-determinism, P and NP

Notes Originally Developed by Sandy Irani
Modified by Michael B. Dillencourt

This material is taken from [P94]; also see [GJ79] for more explanation.

1 Turing Machines

So far our notion of an algorithm has been intuitive, but when reasoning about complexity classes, it useful to have a more formal model of computation. The most common such model is the *Turing machine*. A Turing machine is a very simple model, yet it is capable of computing anything which can be computed by any physically realized computing device.

A Turing machine M is a quadruple (K, Σ, δ, s). K is a set of *states*. These can be thought of as the instructions of the machine. Σ is a finite set of symbols which we will call the *alphabet* of M. Σ always contains the blank symbol $\sqcup$. We assume that K and Σ are disjoint sets. The start state s is an element of K. Finally, δ is the transition function which maps $K \times \Sigma$ to $(K \cup \{\text{"'yes'}, \text{'no'}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$.

A Turing machine has a tape which is divided into an infinite number of discrete locations each of which can hold one input character. When the computation begins, we start with the input (coded in the alphabet Σ) written on the tape, on character per location. To the right and left of the input is an infinite sequence of blank characters written on the tape. There is a cursor (sometimes called the *head*) which points to a single location on the tape. Initially, the cursor points to the first character of the input. At each step, the Turing machine is in one of the states of K. The start state is s. In each step of the computation, based on the current state and the character in the location pointed to by the cursor, the Turing machine overwrites the current character and changes state. The cursor then moves left, right or stays in the same place. This action is determined by the function δ. If the machine transitions into the 'yes' or 'no' state, then it can not continue. If the machine halts by entering the 'yes' state, then we say that the Turing machine *accepts* the input. If the machine halts by entering the 'no' state, then we say that the Turing machine *rejects* the input.

For now, we will only consider *decision* problems, where the goal is to answer a yes/no question. We can expand our definition of a Turing machine to compute more complex output by adding a halting state h to the terminating states. If the Turing machine enters the halt state, then the computation is complete and the contents of the tape is the output of the TM.

Since we are restricting ourselves to decision problems, we can think of a problem (usually referred to as a *language*) as just a subset of all finite strings over the alphabet Σ. In mathematical notation, a problem L is a subset of Σ^*. The subset L denotes all the inputs (or *instances*) for which the correct answer is 'yes' and $\bar{L} = \Sigma^* - L$ denotes all those inputs for which the correct answer is 'no'. We say that a TM M decides a language L if for every $x \in \Sigma^*$, if $x \in L$, then M halts in state 'yes' on input x and if $x \notin L$, then M halts in state 'no' on input x.

Obviously, the Turing machine is a fairly cumbersome computing device. (Try devising a Turing machine which solves a very simple problem, like determining whether a string is a palindrome to convince yourself of this). The single data structure consists of only a single string. What can one hope to accomplish with such a primitive computing device? The answer to this question is that a TM can compute anything which can be computed by any physically realized computing device. The problems that a Turing machines can solve correspond exactly to what any 'algorithm' can solve. It may do so much less efficiently than the workstation on your desk, but anything that your computer can solve can also be solved by a Turing machine. This idea is known as *Church's Thesis* which says that any algorithm can be rendered as a Turing machine.

It is impossible to prove Church's Thesis, since the notion of an 'algorithm' is not formally defined. However, it may help convince you to see that a simple Random Access Machine with a simple instruction set can be simulated by a Turing machine. We will not go through the details here, but the reader is referred to [P94] for more details. There are many variations of the Turing machine which have been studied in the literature. These involve multi-tape Turing machines, multi-headed Turing machines, Turing machines with only a one-way infinite head, etc. The details of how you define a Turing machine usually don't matter since the set of problems which a Turing machine can compute remain the same. Thus, the Turing machine gives us a very simple, well-defined model of computation which captures all the power of any algorithm.

2 The class P

We think of an algorithm as being *efficient* if its running time is bounded by a polynomial for all inputs. More precisely, there is a k, such that on any input which is specified by n bits, the algorithm runs in time $O(n^k)$. Naturally, there can be an algorithm whose running time is bounded by a polynomial which is not practical to use. An algorithm whose running time is $O(n^{100})$ is not likely to be efficient in any practical sense. Even an algorithm which is guaranteed to run in time $O(n \log n)$ may not be useful for a given application if n is very large or speed is at a premium. At the same time there are algorithms whose running is $\Omega(2^n)$ in the worse case which are reasonable to use in many situations because they do not really take exponential time on every input. Nonetheless, the distinction between problems which can be solved in polynomial time and those which can not is often a useful one. Typically, algorithms which run in polynomial time, are bounded by a small polynomial. There are not many n^{100} algorithms. Furthermore, many exponential time algorithms are in fact infeasible to run on inputs of any reasonable size.

In order to formally define a complexity class, we first need to choose a model of computation. We will use the Turing machine. We also need to define a resource which we are trying to bound. Time and space are the most common ones, but there are others. Then we also need a function f from the natural numbers to the natural numbers. $TIME(f)$ consist of all problems such that there is an algorithm which solves the problem and runs in time f on a Turing machine. That is, on any input of size n, the Turing machine will halt with the correct answer in $f(n)$ steps. The class *Polynomial Time* is defined to be

$$\mathbf{P} = \cup_{k=1}^{\infty} TIME(n^k).$$

The class **P** is remarkably robust to the chosen model of computation. For example, one can use the many variations of Turing machines which have been considered in the literature and the class **P** remains the same. We could also use a simple Random Access Machine and the class **P** is still unchanged. This follows from the fact that an algorithm which runs in time f on a RAM, can be simulated by a Turing machine in time f^3. It may be the case that a Turing machine takes many more steps to compute a problem than a more sophisticated computing device. However, for models of computation which resemble what we think of as a 'computer', if the computer solves the problem in polynomial time, then a TM can solve the problem in polynomial time. This leads to an enhanced version of Church's Thesis which states that *'any reasonable attempt to model mathematically computer algorithms and their time performance is bound to end up with a model of computation and associated time cost that is equivalent to Turing machines within a polynomial'* [P94].

3 Non-determinism and the class NP

So far, we have only spoken about models of computations which are physically realizable. This next idea is an unrealistic but useful notion of computation. All of the algorithms which we have discussed so far are *deterministic* in that once the algorithm and the input have been determined, each step is completely determined by all the preceding steps. We are now going to look at *non-deterministic* computation in which at each step, the algorithm has a finite number of choices which it could do next. We can think of a deterministic computation as a path which goes from one configuration of the computing device to the next. (The *configuration* captures all the information about the current point in the computation. For example, a configuration of a Turing machine would be denoted by the entire contents of its tape, the current location of the head and the current state). A non-deterministic computation is like a tree of configurations. A non-deterministic algorithm is said to accept the input if *any* of the paths of computation in the tree lead to an accepting final state. In the case of a non-deterministic Turing machine, δ is no longer a function. Instead, we have a *relation*

$$\Delta \subseteq [K \times \Sigma] \times [(K \cup \{\text{``yes', `no'}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}].$$

Since we have decided to use the Turing machine as our model of computation, we will define non-deterministic complexity classes in terms of the Turing machine. A non-deterministic TM is said to solve a decision problem in time $f(n)$ if for every n and every input of size n, every computation path is bounded by $f(n)$ steps. Furthermore, if the input is a 'yes' instance, then at least one of the computation paths is an accepting computation. If the input is a 'no' instance, then none of the computation paths is an accepting computation. The class $NTIME(f)$ are all the problems for which there is a non-deterministic Turing machine which decides the problem in time $f(n)$. We can extend our definition of polynomial time to non-deterministic Turing machines. *Non-deterministic Polynomial Time* is the following class of problems:

$$\mathbf{NP} = \cup_{k=1}^{\infty} NTIME(n^k).$$

Why do we care about non-deterministic models of computation of they are not very realistic? It turns out, that it is a very convenient way to characterize many important problems. For example the class **NP** is can also be defined in a different way.

Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is polynomial decidable if there is a deterministic TM which can decide the language $\{(x,y) \mid (x,y) \in R\}$ in polynomial time. R is said to be *polynomially balanced* if there is a k such that $(x,y) \in R$ implies that $|x| \leq |y|^k$.

Theorem 1 *Let L be a problem such that $L \subseteq \Sigma^*$. $L \in$ **NP** if and only if there is a polynomially decidable, polynomially balanced relation R such that $L = \{x \mid (x,y) \in R$ for some $y\}$.*

We will not prove the theorem here but will discuss its implications. Each problem in **NP** has the property that any 'yes' instance of the problem has a succinct certificate y of its being a yes instance. It may be very difficult to actually find this certificate, but if it were produced, one could check in polynomial time that an instance is in fact a 'yes' instance.

Let's take for example, the problem deciding whether a set of cities with roads connecting them have a tour whose length is at most some value l. If one were to show you a tour of the cities, it would be easy (in polynomial time) to sum up the lengths of the roads along the tour and verify that the length of the tour is in fact at most l. It may be very difficult to actually find the tour, but if it were supplied, the verification could be done easily. Naturally, we would also want the length of the certificate itself to be polynomially bounded by length of the specification of the input since we would like to check the certificate in time which is bounded by a polynomial in the length of the specification of the tour and cities.

Many problems which arise in practice are in **NP**. Typically, a problem is constructing a mathematical object of some kind (whether it be a path, a tour, a subgraph, a solution of equations, etc.) Sometimes we are seeking the optimal among the set of possible objects. Other times we are just seeking an object which satisfies certain specifications. An object which is sought is the certificate. Generally, it is an easy problem to verify that an object satisfies the desired specifications although may it may be difficult to devise the object in the first place.

Now that we have defined the class **NP** which includes most problems of interest, we will be interested in determining which problems in **NP** can be solved in polynomial time.

References

[GJ79] Garey, M.R., Johnson, S.D. *Computers and Intractability: A Guide to the Theory of NP-completeness* W.H. Freeman and Company, New York, 1979.

[P94] Papadimitriou, C. *Computational Complexity* Addison-Wesley Publishing Company, 1994.