§11. More on reductions; nondeterministic computation. In Note 10, it was claimed that polynomial-time reductions provide a means for comparing the computational tractability of languages. That claim is made precise in the following theorem, which is an analogue of Theorem 9.2.

Theorem 11.1 *Suppose $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ are languages. If L_1 is polynomial-time reducible to L_2, and L_2 is in the class P, then L_1 is also in the class P.*

proof. Since $L_2 \in$ P, there is a polynomial-time Turing machine M_2 which accepts L_2. Further, since L_1 is reducible to L_2, there is a function $f : \Sigma_1^* \to \Sigma_2^*$ satisfying conditions (a) and (b) of page 49 of Note 10. We shall use these observations to construct a polynomial-time Turing machine M_1 which accepts the language L_1. It will follow that the language L_1 is in P.

On input $x \in \Sigma_1^*$ the machine M_1 operates as follows. First, M_1 invokes the polynomial-time transducer assured by condition (b) to transform the word x into the word $f(x)$. Then M_1 returns its head to the left of the tape and proceeds to behave exactly like M_2. By condition (a), it is clear that M_1 accepts the language L_1.

It only remains to check that the machine M_1 is polynomial time. Suppose that the machine M_2 is of time complexity $p_2(n)$, and the transducer is of time complexity $p_t(n)$ where p_2 and p_t are both polynomials. (Without loss of generality, we assume that $p_2(n)$ and $p_t(n)$ are both increasing functions of n; you should justify this.) The time complexity of M_1 can be calculated by summing the time complexities of the three phases of the computation:

(1) Computing $f(x)$ from x requires at most $p_t(n)$ steps, where n is the length of the input word x.

(2) While computing $f(x)$, the head of M_1 can have strayed at most $p_t(n)$ steps, and hence can be returned to the leftmost square in at most $p_t(n)$ further steps.

(3) The word $f(x)$ was written down within $p_t(n)$ steps, and hence can have length at most $p_t(n)$. In the final phase, M_1 behaves as M_2 would on input $f(x)$. Hence the number of steps performed in the final phase is at most $p_2(p_t(n))$.

Thus the machine M_1 is of time complexity $p_1(n) = 2p_t(n) + p_2(p_t(n))$. Observe that p_1 is a polynomial (whose degree is the product of the degrees of p_2 and p_t). □

Note that in the preceding theorem if L_2 is recognized in time $O(n^d)$ and the reduction is computable in time $O(n^e)$ then the proof shows that L_2 is recognized in time $O(n^{de})$ and this might well be the best that we can do. Thus if we want a class that is closed under polynomial time reductions[23] and that contains those languages that are recognized in time $O(n)$ then we must include at least all of P. One alternative is to restrict reductions to be computable in $O(n)$ time but this is extremely restrictive.

[23]If we have a notion of reducibility between languages then we say that a class $\mathcal{C}$ of languages is closed under that notion if whenever a language L is reducible to some language in $\mathcal{C}$ then L is also in $\mathcal{C}$.

An equivalent statement of Theorem 11.1 is that if L_1 is polynomial-time reducible to L_2, and L_1 is *not* in P, then L_2 is *not* in P. It is in this form that we shall use the theorem. Note the analogy with the use of reductions in establishing undecidability results. By exhibiting a reduction from a non-recursive language L_1 to another language L_2, we demonstrate that L_2 is non-recursive; by exhibiting a polynomial-time reduction from an intractable language L_1 to another language L_2, we demonstrate that L_2 is intractable.

By way of example, consider the problems SAT and CLIQUE that were defined in NOTE 10. Recall that it was shown that SAT is polynomial-time reducible to CLIQUE. Now, if it could be proved that CLIQUE $\in$ P then we could immediately deduce, using Theorem 11.1, that SAT $\in$ P. Equivalently, if it could be proved that SAT $\notin$ P then we could deduce that CLIQUE $\notin$ P. Thus, even if we do not know the absolute computational *complexity*, or difficulty of these two problems, we can at least say something about their relative complexity.

We shall see later, as a consequence of a much more general result known as Cook's theorem, that CLIQUE $\leq_{\mathrm{P}}$ SAT. (Note that this is the converse of the result obtained in §10.3.) Thus, by Theorem 11.1, SAT and CLIQUE are either both in P or both outside P. SAT and CLIQUE are members of a large class of naturally defined problems — drawn from logic, graph theory, and many other areas — which are all pairwise related by polynomial-time reductions. Theorem 11.1 says that the problems in this class are either all in P (i.e., all tractable), or all outside P (i.e., all intractable). Most computer scientists suspect the latter; however nobody has so far been able to *prove* this. The fact that so many apparently difficult problems are related by polynomial-time reductions is a surprising and unexpected phenomenon. To begin to understand it, we must venture into the realm of nondeterministic computation but first we look at another motivation for such a move.

§11.1. Naïve search and P. Many interesting computational problems concern questions about finite structures, typically: 'given a structure of the appropriate type does it have a certain property?, e.g., a sub-structure of a certain type' (CLIQUE provides a typical example). For very many such problems we have the interesting situation that given a proposed solution it is a fairly easy matter to check its correctness (more accurately we can check in time that is polynomial in the length of the problem instance). One method of answering such questions is by *naïve search*, i.e., try all of the finitely many possibilities one after the other. Rather than go into generalities let us illustrate the point with an early example of such a question.

INSTANCE: A finite undirected graph G.

QUESTION: Does G have an Eulerian cycle, i.e., can we start at a vertex v visit every edge exactly once and return to v?

Clearly this problem[24] is solvable: let the edges be $e_1, e_2, \ldots, e_n$. Try all possible permutations of the edges and for each permutation test it to see if it does the job. Although this is an algorithm it is naïve to say the least; there are $m!$ permutations to try! In fact Euler

[24]The problem has its origins in a puzzle based on bridges connecting islands in a river at Königsberg.

proved that the answer is 'yes' if and only if the graph is connected and each vertex has an even number of edges attached. Obviously this condition is easy to check and provides a practical solution to the problem (how can we check efficiently that a graph is connected?).

A general question now suggests itself. Are there problems for which we can do no better than naïve search? If we are to answer this question we need to capture the notion of naïve search in a manageable way. Note that we do not really want to rule out searching altogether since small structures can be searched in acceptable time. The key point is that an uncontrolled naïve search will lead us to consider a number of possibilities that grows faster than any polynomial. Roughly speaking at each decision point we have two or more choices so that if the number of decision points is proportional to the size of the input, say it is at least cn for some constant c, then we have at least 2^{cn} possibilities to consider (this argument is meant only as a rough indication). We are therefore lead to the following simple criterion for ruling out naïve search: given a problem can it be solved in time that is a polynomial function of the size of the input instance?

Euler's success amongst many others shows that certain problems do have structure that can be exploited. As another, deeper, example consider the problem of perfect matchings for bipartite graphs[25]. It is possible to provide a practical polynomial time algorithm based on Flow Networks. In general if a problem is at all non-trivial then finding a polynomial time algorithm for it involves a genuine breakthrough. This observation is another reason for focusing on the class P.

We can model naïve search by Turing machines by means of a very simple relaxation on the transition function δ and we proceed to describe this in the next section. This in turn allows us to formulate the central question of this section in precise terms.

EXERCISE Prove Euler's result as described above.

EXERCISE This exercise is designed to illustrate the point that certain problems can be put into P for very trivial reasons. A *triangle* in a graph is a subgraph consisting of exactly three vertices each of which is joined to the other by an edge of the graph, i.e., a 3-clique. Give a polynomial time algorithm to decide if a given graph contains a triangle (take the number of vertices to be the size of the input).

EXERCISE Clearly SAT and CLIQUE are problems that can be attacked by naïve search. What are the resulting runtimes, measured in terms of any natural definition for the size of an instance?

§11.2. Nondeterministic Turing machines. A *nondeterministic* Turing machine differs from the deterministic Turing machine of NOTE 3 in just one respect. In a nondeterministic machine, transitions are governed, not by a partial *function* $\delta : (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\})$, but by a general *relation* $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$. Just as before, we may envisage δ as a set of quintuples. However, whereas this set of quintuples was

[25]See T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, (1994) for a definition and further details.

previously constrained by the fact that δ was a function, it is now entirely *unconstrained* so that for a given state q and symbol a there might be several quintuples $(q, a, \cdot, \cdot, \cdot)$. All the other definitions of NOTE 3 — configurations, the relation $\vdash$, computations, and the condition for acceptance — remain unchanged. Thus the deterministic Turing machine is a special case of the nondeterministic Turing machine (as it ought to be).

From a definitional point of view, nothing more need be added. However it is worth pausing to consider some of the consequences of generalizing δ from a partial function to a relation. Let $M = (Q, \Gamma, \Sigma, \flat, q_I, q_F, \delta)$ be a nondeterministic TM. For each configuration γ of M, there will in general be several next configurations: $\gamma \vdash \gamma'_1$, $\gamma \vdash \gamma'_2$, $\ldots$, $\gamma \vdash \gamma'_k$, where γ'_1, $\ldots$, γ'_k are all distinct. Thus, for any word $x \in \Sigma^*$, there may be very many possible computations of M on input x. Now, recall that the language accepted by M is defined to be

$$L(M) = \{x \in \Sigma^* : q_I x \vdash^* \alpha q_F \beta, \text{ where } \alpha, \beta \in \Gamma^*\}.$$

Thus, informally, M accepts input x if *any* of its (nondeterministic) computations on input x lead to the accepting state q_F. There may be computations that do not lead to acceptance but this is not relevant as long as at least one leads to acceptance. Of course rejection now means that there is no computation path that leads to acceptance. (This is entirely analogous to searching for a solution; if any one attempt succeeds then we have a solution no matter how many attempts failed beforehand.)

EXAMPLE Consider the machine M_{abc} with $Q = \{q_0, q_1, q_2, q_3\}$, $q_I = q_0$, $q_F = q_3$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \flat\}$, and with transition relation δ specified by the quintuples $(q_0, \mathtt{a}, q_0, \mathtt{a}, R)$, $(q_0, \mathtt{a}, q_1, \mathtt{a}, L)$, $(q_0, \mathtt{a}, q_1, \mathtt{a}, R)$, $(q_0, \mathtt{b}, q_0, \mathtt{b}, R)$, $(q_0, \mathtt{c}, q_0, \mathtt{c}, R)$, $(q_1, \mathtt{b}, q_2, \mathtt{b}, L)$, $(q_1, \mathtt{b}, q_2, \mathtt{b}, R)$, and $(q_2, \mathtt{c}, q_3, \mathtt{c}, R)$. The language L_{abc} accepted by M_{abc} is the set of words $x \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$, such that x contains either `abc` or `cba` as a sub-word. Observe that if $x \notin L_{\text{abc}}$ then M_{abc} has no accepting computations on input x. Observe, also, that if $x \in L_{\text{abc}}$ then M_{abc} has non-accepting computations on input x, such as

$$q_0\mathtt{bacbab} \vdash \mathtt{b}q_0\mathtt{acbab} \vdash \mathtt{ba}q_1\mathtt{cbab},$$

in addition to the accepting computations, such as

$$\begin{array}{rlllllll} & q_0\mathtt{bacbab} & \vdash & \mathtt{b}q_0\mathtt{acbab} & \vdash & \mathtt{ba}q_0\mathtt{cbab} & \vdash & \mathtt{bac}q_0\mathtt{bab} \\ \vdash & \mathtt{bacb}q_0\mathtt{ab} & \vdash & \mathtt{bac}q_1\mathtt{bab} & \vdash & \mathtt{ba}q_2\mathtt{cbab} & \vdash & \mathtt{bac}q_3\mathtt{bab}. \end{array}$$

EXERCISE A very attractive way to view the set of all possible computation paths of a nondeterministic Turing machine on a given input is as a tree. The root is the starting configuration. The children of any vertex labeled by a configuration C are all possible next moves (if any). Draw the first few levels of the tree for the example machine given above starting with various inputs.

Nondeterministic TMs are no more powerful that deterministic TMs in terms of the class of languages which can be recognized.

THEOREM 11.2 *If a language L is accepted by a nondeterministic Turing machine, then L is accepted by a deterministic Turing machine.*

PROOF. Suppose M is a nondeterministic TM accepting L. We construct a three-tape deterministic TM $\widehat{M}$ that also accepts L.

For each state/symbol pair of M, a finite number of possible transitions are available: number these consecutively, starting at zero. Let r be the maximum number of transitions available for any state/symbol pair. A computation of M on input x can be described by a sequence of digits in base r, successive digits specifying the action to be taken in successive steps of the machine M. Not all sequences will specify a valid computation since, in general, fewer than r nondeterministic choices will be available at each transition.

The simulating machine $\widehat{M}$ has three tapes. The first contains the input, and remains unaltered during the computation. The second contains a sequence of digits in base r; this sequence picks out a particular nondeterministic computation of M. The third tape is used for the simulation of M. On tape 2, $\widehat{M}$ cycles through all possible sequences composed of the base r digits: first the sequences of length 1, then all sequences of length 2, all sequences of length 3, and so on. For each generated sequence of length t, the machine $\widehat{M}$ copies the contents of tape 1 across to tape 3 and simulates M for t steps. Each nondeterministic choice of M is resolved by consulting tape 2. The head on tape 2 advances one position for every step of the simulation. If M ever reaches the accepting state then $\widehat{M}$ accepts.

If it is the case that M accepts input x, then $\widehat{M}$ will eventually generate on tape 2 an appropriate sequence of choices and will itself accept. However, if M does not accept input x, then no generated sequence on tape 2 will cause $\widehat{M}$ to accept. □

Note that the simulation employed in the proof of Theorem 11.2 is not very efficient; the runtime blows up exponentially. Thus the proof of Theorem 11.2 does *not* imply that a deterministic TM can perform a computation *as quickly* as a nondeterministic TM.

§11.3. The language class NP. The nondeterministic TM is not intended to be a realistic model of computation. Its importance lies in the fact that it allows us to define a language class, NP, which appears to have great practical significance.

Let M be a nondeterministic TM with input alphabet Σ. If, for all inputs $x \in \Sigma^n$ of length n, the machine M makes at most $T(n)$ transitions before halting, then we say that M is *of time complexity* $T(n)$. Note that this definition is consistent with the definition for *deterministic* machines, which have just one possible computation on any input. As before, we say that a nondeterministic TM is *polynomial time* if it is of time complexity $p(n)$ for some polynomial p. The class of all languages which are accepted by polynomial-time nondeterministic TMs is denoted by NP.

Many natural problems belong to the class NP, SAT being a typical example. To see that SAT $\in$ NP, consider the nondeterministic TM M that operates as follows. Suppose the input to M is a CNF formula ϕ in the variables $x_1, x_2, \ldots, x_k$. The machine M sets aside k squares of workspace on its tape. It then writes down, nondeterministically, a sequence of k binary digits onto the k squares; these digits are interpreted as a truth assignment to the k variables of ϕ. M now checks whether the truth assignment it has

just 'guessed' is a satisfying assignment to ϕ. The checking phase can be carried out in an entirely deterministic fashion: for each clause, M verifies that at least one literal in the clause is true under the chosen assignment. It is clear that checking phase can be completed in polynomial time.

As discussed in §11.1, SAT is one of a large class of *search problems* that are characterized by the search for a *solution* within a *search space* which is potentially of exponential size. For SAT, the search space is the set of all 2^k truth assignments to the k variables of ϕ, and the sought-for solution is a satisfying assignment to ϕ. The enormous size of the search space may make solutions very difficult to *find*. Nevertheless, it is often the case that putative solutions are easy to *verify*; thus, in the case of SAT, it is easy to check whether a given assignment to the variables makes ϕ true. Naturally occurring search problems which have easily verified solutions are good candidates for membership in NP.

We can now give a precise formulation to the question raised in §11.1 regarding naïve search. If there is a language that is in NP but not in P then when deciding membership for this language we cannot do substantially better than naïve search in terms of runtime. If on the other hand it turned out that P = NP then problems like SAT and CLIQUE can be answered in a way that avoids a crude search, i.e., there must be some deep structure to these problems which has so far eluded everybody. The P versus NP question is amongst the most difficult problems of Computer Science and has resisted solution since the early seventies when it was first posed.