§10. Computing against the clock. So far, our main concern has been the classification of problems under two headings: decidable and undecidable. (Actually, three: *semi-decidable* corresponds to recursively enumerable.) That is not the whole story, however. For although a decidable problem is certainly solvable in principle, it may not be solvable in any realistic sense. That is, the time required to obtain a solution to the problem may grow so fast, as a function of input size, that only the smallest inputs can be treated within a reasonable time bound.

This note introduces a division of computational tasks, into 'tractable' and 'intractable', whose significance appears to be independent of the choice of computational model. We identify a language class P that captures the notion of computational tractability in an intuitively appealing way. Although P must be defined relative to a particular machine model (in our case the Turing machine), most computer scientists believe that the *same* class would be obtained whatever reasonable machine model is chosen; indeed this has been proved for many models.

§10.1. The class P. Let M be a Turing machine with input alphabet Σ. If M halts within $T(n)$ steps on all inputs of length n, then we say that M is $T(n)$ *time bounded*, or that M is *of time complexity* $T(n)$. Consider the Turing machine M_{palin} of Note 3. For an input of length n, the machine M_{palin} makes at most $\frac{1}{2}(n+1)(n+2)$ transitions before halting. (Indeed, for a *palindrome* of length n, it makes precisely $\frac{1}{2}(n+1)(n+2)$ transitions.) Thus, M_{palin} is of time complexity $T(n) = \frac{1}{2}(n+1)(n+2)$. Notice that $T(n)$ is simply an *upper bound* on the number of transitions made by M_{palin} on inputs of length n; there is no requirement that this number of transitions is actually achieved for all inputs of length n, or indeed for *any* input of length n. Thus M_{palin} is also of time complexity $3n^2$, or $3n^3$, or even 3^n. These statements, although all true, contain diminishing information.

We say that the Turing machine M is *polynomial time* if M has time complexity $p(n)$ for some polynomial p. (A polynomial is a function $p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$, where d is the *degree* of the polynomial, and the a_i are constant *coefficients*[21].) The machine M_{palin} is clearly polynomial time, since its time complexity is described by a polynomial of degree 2. The class of all languages that are accepted by polynomial-time Turing machines is denoted by P. The language $L_{\text{palin}} \subseteq \{\mathtt{0}, \mathtt{1}\}^*$ consisting of all binary palindromes is a typical member of P. The class P is intended to capture the notion of *computational tractability*. We regard languages *outside* P as computationally *intractable* because the time required to recognize words in these languages grows faster than any polynomial in n.

An important feature of the class P is that it appears to be invariant under changes in the model of computation. Thus the class of languages that can be

[21] Strictly speaking d is the degree provided $a_d \neq 0$. The degree is undefined if all coefficients are 0, i.e., the function always returns 0. This point will not concern us in this course!

recognized by a Turing machine in polynomial time appears to be the same as the class of languages recognized by any other 'reasonable' model of computation in polynomial time. Informally, a model of computation is 'reasonable' in this context if each computational step could be performed by fixed hardware in bounded time. Note that the RAM model, as it stands, is 'unreasonable' because numbers of arbitrary size can be manipulated in a single computational step. However it is possible to deal with this point by charging each arithmetic operation according to the size of numbers involved (e.g., according to the so-called *logarithmic cost criterion*).[22] It is instructive to compare this invariance of P with the analogous feature of the class of recursive languages: it also remains invariant under changes of reasonable model (where 'reasonable' now means 'intuitively computable').

It is quite reasonable to question the claim that the class P captures the notion of 'tractability' if this is to mean 'practically useful algorithm'. After all an algorithm that runs in time $n^{1000000}$ isn't useful by any criterion. Moreover it can be proved that there are problems that can be solve in the given time but not in substantially less, e.g., not in n^{999999} time (admittedly these are not natural problems). It is more accurate to view the class P as an idealization, or approximation, of the notion of 'practically useful'. The fact is that unless we are prepared to tie ourselves down to a specific model, then we have no choice but to allow all polynomial-time problems in our class. However, even if one ignores the tractability issue, the class P is still of immense interest owing to the observation of the preceding paragraph. We will see another important reason for focusing on P later on when we consider non-deterministic computation and its relation to searching.

§10.2. Polynomial-time reductions. To demonstrate that a language L is tractable, i.e., is in the class P, it is enough to exhibit a polynomial-time Turing machine that accepts L. For many languages $L \in \mathrm{P}$ this can be done without great difficulty. But to demonstrate that a language L is intractable, i.e., is outside P, it is necessary to prove that *no* polynomial-time Turing machine accepts L. The latter task seems much more difficult, since (assuming L is recursive) the set of Turing machines that accept L contains machines that are arbitrarily intricate.

However there is a simple way to *compare* the computational tractability of languages, and this involves a refinement of the notion of reduction introduced in NOTE 8. Let L_1 and L_2 be languages over alphabets Σ_1 and Σ_2, respectively. A *polynomial-time reduction from L_1 to L_2* is a function $f : \Sigma_1^* \to \Sigma_2^*$ satisfying:

(a) $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$;

(b) there is a polynomial-time Turing machine transducer that computes f.

[22] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley 1974, p. 12.

If a polynomial-time reduction from L_1 to L_2 exists then we say that L_1 *is polynomial-time reducible to* L_2, and write $L_1 \leq_{\mathrm{P}} L_2$. The reader will probably have guessed that polynomial-time reductions play a similar role in *complexity theory* (the theory of the computationally tractable) as unrestricted reductions do in computability theory. Before exploring this role in detail, we consider an example of a polynomial-time reduction.

§10.3. Satisfying assignments and cliques. We have seen that formal languages and decision problems offer different views of the same thing. The language view is more convenient when developing the theoretical basis of the subject; the decision problem view is more convenient when discussing practical applications. From now on, we shall move freely between the two views, and not distinguish in our notation between languages and decision problems.

Here are two problems. The first, SAT, is taken from propositional logic:

INSTANCE: A Boolean formula ϕ in conjunctive normal form (CNF).[23]

QUESTION: Is there an assignment of truth values to the variables of ϕ that makes ϕ true?

The second, CLIQUE, is a problem from graph theory:

INSTANCE: An undirected graph $G = (V, E)$, and an integer k.

QUESTION: Does G possess a k-clique? (A k-clique is a subset $U \subseteq V$ of size k, such that every pair of distinct vertices in U is joined by an edge.)

Expressed as languages, SAT is the set of encodings of satisfiable formulas in CNF, and CLIQUE is the set of encodings of pairs $\langle G, k\rangle$ where G is an undirected graph, k is a natural number, and G contains a k-clique. From now on we shall not be too explicit about the encodings used; any reasonable encoding will do. For example, graphs might be specified as adjacency matrices in row-major order, and natural numbers presented in binary notation.

We shall demonstrate that SAT $\leq_{\mathrm{P}}$ CLIQUE by exhibiting an explicit polynomial-time reduction from one problem to the other. Note that the two problems are quite different in appearance, so it is surprising at first sight that they should be related in this way. The reduction is required, by condition (a), to map each CNF Boolean formula ϕ to an undirected graph G and integer k such that

$$\phi \text{ is satisfiable} \iff G \text{ has a } k\text{-clique.} \qquad (*)$$

Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_r$, where each clause C_i is given by $C_i = (\alpha_{i1} \vee \alpha_{i2} \vee \cdots \vee \alpha_{i,s_i})$, and each α_{ij} is a literal. Say that a pair of literals is *complementary* if it consists

[23] Recall that a formula is in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals, and each literal is a variable or its negation.

of the negated and un-negated forms of the *same* variable. (E.g., x and $\neg x$ form a complementary pair.) We introduce distinct vertices v_{ij} for $1 \leq i \leq r$ and $1 \leq j \leq s_i$ (with the intention that the vertex v_{ij} corresponds to the literal α_{ij}[24].) Our reduction maps the formula ϕ to the graph $G = (V, E)$ with

$$V = \{v_{ij} \mid 1 \leq i \leq r \text{ and } 1 \leq j \leq s_i\}$$
$$E = \{\{v_{ij}, v_{hk}\} \mid i \neq h, \text{ and the pair } \alpha_{ij}, \alpha_{hk} \text{ is } \textit{not} \text{ complementary}\}.$$

Informally, two vertices are connected by an edge if and only if the corresponding literals occur in different clauses in ϕ and are not complementary. The integer k—which forms part of each instance of CLIQUE—is set equal to r, the number of clauses in ϕ. This completes the description of the reduction.

We now verify that the reduction satisfies condition $(*)$. First, consider the forward implication. Take any assignment of truth values to the variables of ϕ that makes ϕ true. Under this assignment, at least one literal in each clause of ϕ will be made true; select one such literal from each clause. The k vertices of G that correspond to this choice of literals must form a k-clique. (Since all k literals are true, no pair of them can be complementary.)

Now, consider the reverse implication. Suppose that G has a k-clique. Consider the k literals of ϕ corresponding to the k vertices of the clique. Each of the k literals must occur in a different clause of ϕ; moreover, no pair of the k literals is complementary. Therefore, there is an assignment of truth values to the variables of ϕ which makes all k literals, and hence the whole formula ϕ, true.

Finally, we must check condition (b): that the reduction can be computed in polynomial time. To do this formally, we need to be precise about the encodings used for formulas and graphs. But without going that far, it should be clear that reduction requires no great computational effort. Suppose that ϕ contains m literals. The main work is to construct the adjacency matrix of G. Assign sequence numbers in the range $[0, m-1]$ to the literals of ϕ. Then make the (i, j) entry of the adjacency matrix 1 if the literals with sequence numbers i and j are in different clauses and are *not* complementary, and 0 otherwise. The whole matrix may be constructed by embedding this simple test within nested i- and j-loops. It is clear that the whole procedure can be implemented to run in polynomial time. □

EXERCISE Sketch the graph G obtained by applying the above reduction to the formula

$$\phi = (x \vee \neg y) \wedge (\neg x \vee \neg y \vee z) \wedge (y \vee z) \wedge (\neg y \vee \neg z).$$

Take a satisfying assignment to ϕ and from it construct a corresponding 4-clique in G. Now take a different 4-clique in G and construct a corresponding satisfying assignment to ϕ.

[24]There is a subtle point here: it is possible that, e.g., the literals α_{12} and α_{32} are the same, however the vertices v_{12} and v_{32} are distinct.