

1. Tractable and Intractable Computational Problems

So far in the course we have seen many problems that have polynomial-time solutions; that is, on a problem instance of size n , the running time $T(n) = O(n^k)$ for some k , with k typically small. For example, we discussed sorting algorithms with $T(n) = O(n \lg n)$, and we showed how to find shortest paths in time $T(|E|, |V|) = O(|E| \lg(|V|))$ using Dijkstra's algorithm. On the other hand, for the TSP we gave a dynamic programming solution that solved the problem in $T(n) = O(n^2 2^n)$, where n is the number of vertices. We did *not* give any polynomial-time bounded solution for this problem.

Problems that can be solved by a polynomial-time algorithm are considered tractable, and those that can not be solved in time bounded by a polynomial in the size of the input are considered intractable. Therefore, sorting and shortest-path are tractable. From our discussion so far we do not know if TSP is tractable (in fact it is unknown at present). The purpose of these notes is to explore this dichotomy between tractable and intractable problems. Our final result will not quite be a neat division into tractable and intractable problems. Rather, we will end up with two important classes: the problems we *know* are tractable (this class will be called P), and the problems which are *suspected* to be intractable (NPC). A very famous unsolved problem is the question of whether or not $NPC \subset P$; that is, whether or not the seemingly intractable problems are in fact intractable.

Why aim for such a categorization? Before you set out to solve a problem, it is useful to determine first if it is in NPC . If it is, you know that most experts believe it unlikely that you will ever find an efficient (polynomial-time) solution. Unless you are very ambitious and have a lot of time to devote to the problem, you may decide at this point to abandon hope for a polynomial-time solution and go with a non-polynomial time algorithm and confine yourself to relatively small problem instances. Alternatively, you can use an approximation algorithm that might give a good answer (but not guaranteed to be the best) in polynomial time. Finally, you might reformulate the original problem to a version that is in P . (Often the distinction between a problem in P and NPC can be subtle.)

We will develop a theory for *decision problems*; that is, problems that have a yes or no answer. For example, we considered the shortest path problem: Given a weighted digraph G and vertices s and t , what is the weight of the lightest path from vertex s to vertex t ? The decision version of this problem is: Given a weighted digraph G , vertices s and t , and integer K , is there a path of weight at most K between s and t ? The decision version is certainly no harder than the original optimization problem. If we can solve the optimization problem, then we only need to compare the optimal solution with K to solve the decision problem.

The class of decision problems for which polynomial time algorithms exist is denoted P .

Suppose that we are given an instance of the decision version of the shortest path problem with a yes answer, and someone offers a "certificate" that demonstrates that the problem is a "yes" instance. What form might this certificate take? Suppose it is a path from s to t that has length $\leq K$. The path has at most $|V| - 1$ edges, and so we can certainly check the certificate in time that is bounded by a polynomial in the size of the problem instance. This problem therefore has the *succinct certificate property*: given a "yes" instance, there is at least one "succinct certificate" (and possibly many) that verifies that the problem is a "yes" instance

in time that is bounded by a polynomial in the size of the problem instance. A “no” instance of the problem has no such certificate.

The decision version of the traveling salesman problem, which we will call **TSP** is: given n cities and distance matrix d and integer K , is there a tour of the n cities with total length $\leq K$? This problem has the succinct certificate property. If we are given a tour (permutation of the n cities), we can check in $O(n)$ time to see if it has length $\leq K$.

The class of yes-no problems that have the succinct certificate property is called NP . (NP stands for *nondeterministic polynomial*.) Clearly $P \subset NP$, since if the problem can be solved in polynomial time, then the solution constructed by the algorithm can certainly be checked in polynomial time.

The decision version of the knapsack problem we discussed in class, which we call **KNAPSACK**, is: Given a weight budget W , n items of value v_i and weight w_i , $i = 1, 2, \dots, n$, and value K , is there a choice of the items that weigh $\leq W$ and have total value $\geq K$? We saw (in homework problem 16.2-2) that the optimization version can be solved in time $O(nW)$, so of course the decision version can also be solved in time $O(nW)$. Does this mean that **KNAPSACK** is in P ?

To answer this question, we must be careful about the size of the input. It is fair to consider the data about the n items to have size $O(n)$, but what about W ? In binary representation it has $\lg(W)$ bits. So the size of the input is really about $n \lg W$. Therefore, the running time is exponential in the size of the problem instance.

The decision version of knapsack does have the succinct certificate property, however, and so it is in NP . Is it in P ? We don't know; we have only argued that the particular algorithm we came up with does not show it to be in P since it is not a polynomial time algorithm.

The field of logic provides many examples of decision problems. The **CIRCUIT-VALUE** problem is: Given a Boolean circuit and its inputs, is the output T? This problem is in P (see below). The **SAT** problem is: Given a Boolean circuit, is there a way to assign values to its inputs so that the output is T? While superficially similar, **SAT** seems to be a much more difficult problem. The obvious solution is to try all inputs, of which there are exponentially many.

The linear programming problem (**LP**) is: Given an $n \times m$ matrix A and an m vector b , are there nonnegative real numbers x_1, x_2, \dots, x_n satisfying $Ax \leq b$?

The integer linear programming problem (**ILP**), is: Given an $n \times m$ matrix A and an m vector b , are there nonnegative integers x_1, x_2, \dots, x_n satisfying $Ax \leq b$?

It turns out that **LP** is in P , while **ILP** appears to be exponential.

We next define a class of decision problems which are the “hardest” problems in the class NP . We first need to introduce *reductions*.

Let A and B be two decision problems. A *reduction* from A to B is a polynomial-time algorithm R which transforms inputs of A to equivalent inputs to B . If x is an instance of problem A , then $R(x)$ is an instance of problem B , and x is a “yes” instance of A if and only if $R(x)$ is a “yes” instance of B . If we have a polynomial-time algorithm for problem B , and a reduction from A to B , then we have a polynomial-time algorithm for problem A . To see this, suppose that our polynomial-time algorithm for problem B has running time $T(n) = O(n^k)$ for some k , and suppose that our reduction R has running time $O(n^j)$. To solve an instance

x of A of length n , first convert x to $R(x)$, an instance of B , in time $O(n^j)$; notice that the length of $R(x)$ is at most $O(n^j)$. Then solve $R(x)$, which takes time $O((n^j)^k) = O(n^{jk})$.

Cook's Theorem: A problem is in **NP** if and only if it can be reduced to **SAT**.

Part of the theorem is easy to prove. If a problem can be reduced to **SAT**, then it is in **NP**. A certificate of any "yes" instance would be the reduction of the instance with a certificate for the resulting input of **SAT**.

Here is a sketch of the plausibility of the other half of the theorem: If a problem A is in **NP** then it can be reduced to **SAT**. Since A is in **NP**, there exists a polynomial-time algorithm to check problem instances and certificates for validity. By combining polynomially many Boolean circuits, we can implement the algorithm that does the validity checking (the input gates correspond to the bits of the problem instance and the certificate). Let x be an instance of problem A . Plug in the correct T/F values for the input gates of the circuit corresponding to the problem instance, and leave unknown the gates corresponding to the certificate. We now have an instance of **SAT** which captures the question of whether a valid certificate for x exists; that is, whether x is a "yes" instance of A . Therefore, we have reduced A to **SAT**.

If a problem is in **NP** and *all* other problems in **NP** reduce to it, then we say that the problem is **NP-Complete**. Thus an **NP**-complete problem is (up to a polynomial) as hard as any other problem in **NP**. Cook's theorem gives us our first example of an **NP-Complete** problem.

2. Examples of Reductions

In these notes we will illustrate polynomial reductions. This allows one to show that a problem is NP-complete if we reduce a known NP-complete problem to it.

Let $\{u_1, u_2, \dots, u_n\}$ be a collection of Boolean variables. A *truth assignment* assigns a value T or F to each variable. We use a bar to denote complement: \bar{u} is T if and only if u is F . If u is a variable, then u and \bar{u} are *literals*.

A *clause* is a set of literals, representing the disjunction (OR) of those literals, for example $(u_2 \vee \bar{u}_4 \vee u_5)$. This clause will be satisfied *unless* $u_2 = F$, $u_4 = T$, and $u_5 = F$. A collection C of clauses is satisfiable if and only if there is a truth assignment that simultaneously satisfies all the clauses in C . We will write such a collection of clauses by connecting them with \wedge (read "AND"); such a Boolean expression is in *conjunctive normal form* (CNF).

The satisfiability problem (abbreviated SAT) is: Given a collection of clauses C over variables U , is there a satisfying truth assignment for C ? Cook's theorem states that SAT is NP-complete. We will now reduce SAT to some other interesting problems, thus showing that they are NP-complete.

A set of clauses is in 3 conjunctive normal form (3-CNF) if each clause has exactly 3 literals. The 3-SAT problem is the special case of SAT in which each clause has exactly three literals.

SAT reduces to 3-SAT Consider any formula F in CNF, consisting of clauses C_1, C_2, \dots, C_m over variables $\{u_1, u_2, \dots, u_n\}$. We shall construct a new formula F' with exactly three literals per clause that is satisfiable if and only if F is. We do this by examining each clause C_i in turn, replacing it by an equivalent set of clauses, introducing new variables if necessary. If C_i already has exactly three literals, we do nothing. Otherwise, there are two cases to consider: C_i has fewer than three literals and C_i has more than three literals.

First suppose that C_i has one or two literals. If $C_i = u_1$, then introduce new variables z_1, z_2, α, β and replace C_i by the clauses $(u_1 \vee z_1 \vee z_2)$ and

$$(\bar{z}_1 \vee \alpha \vee \beta) \wedge (\bar{z}_1 \vee \bar{\alpha} \vee \beta) \wedge (\bar{z}_1 \vee \alpha \vee \bar{\beta}) \wedge (\bar{z}_1 \vee \bar{\alpha} \vee \bar{\beta}) \\ \wedge (\bar{z}_2 \vee \alpha \vee \beta) \wedge (\bar{z}_2 \vee \bar{\alpha} \vee \beta) \wedge (\bar{z}_2 \vee \alpha \vee \bar{\beta}) \wedge (\bar{z}_2 \vee \bar{\alpha} \vee \bar{\beta}).$$

These last clauses force the variables z_1 and z_2 to be false in any truth assignment satisfying F' , so that the clause u_1 is equivalent to its replacement. If $C_i = (u_1 \vee u_2)$, introduce only z_1, α, β and disregard the clauses containing \bar{z}_2 .

If C_i has $k > 3$ literals, say $(u_1 \vee u_2 \vee \dots \vee u_k)$, then introduce new variables z_1, z_2, \dots, z_{k-3} , and replace C_i by

$$(u_1 \vee u_2 \vee z_1) \wedge (\bar{z}_1 \vee u_3 \vee z_2) \wedge (\bar{z}_2 \vee u_4 \vee z_3) \wedge \dots \wedge (\bar{z}_{k-3} \vee u_{k-1} \vee u_k).$$

This last set of clauses is satisfiable if and only if C_i is: If C_i has a truth assignment, let l be the smallest integer such that u_l is true. Then the new clause is satisfied with the same truth assignment and with $z_i = T$ for $i \leq l-2$ and $z_i = F$ for $i > l-2$. Conversely, suppose that the new clause is satisfied by some truth assignment. If C_i is *not* satisfied by the truth assignment, then $u_i = F$ for $1 \leq i \leq k$. But this would imply that the new clause is not satisfied.

It is clear that the reduction can be accomplished in polynomial time.

Let $G = (V, E)$ be a graph. A subset $I \subset V$ is called an *independent set* if no two vertices in I are directly connected by an edge in E . For example, in Figure 2, $\{1, 4, 6\}$ and $\{2, 5\}$ are two examples of independent sets.

The Independent Set problem is: Given a graph G and an integer K , is there an independent set $I \subset V$ containing at least K vertices?

Independent Set is NP-complete.

Proof. It is clearly in NP. We will reduce 3-SAT to Independent Set; i.e., given a Boolean formula F in 3-CNF, produce a graph $G = (V, E)$ and integer K such that G has an independent set of size at least K if and only if F is satisfiable. Here is the construction: Let K be the number of clauses. For each clause, create a node for each literal in the clause, connected in all possible ways (a “triangle” for each clause). Also draw an edge between literals of different clauses if contradictory; e.g., if u_i appears in one clause and \bar{u}_i appears in a different clause, connect the corresponding nodes with an edge.

For example, consider the formula

$$(u_1 \vee u_2 \vee u_3) \wedge (\bar{u}_1 \vee u_3 \vee u_4) \wedge (\bar{u}_2 \vee \bar{u}_4 \vee u_5).$$

The graph is illustrated in Figure 1.

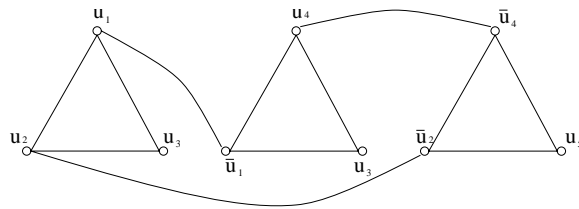


FIGURE 1. Graph used in Independent Set reduction.

If there is an independent set of size at least K , then it must have exactly one node from each group. Think of the chosen node in a group as a literal that satisfies that clause. Since contradictory literals are connected by an edge, no nodes in I are contradictory, so the literals together comprise a satisfying truth assignment (if not used, a variable can be set to either T or F). Conversely, suppose that we have a satisfying truth assignment for F . Pick a T literal from each clause. The corresponding set I of K nodes is an independent set (they are from the same truth assignment, so not contradictory). \square

A *clique* in a graph $G = (V, E)$ is a fully connected set of nodes. For example, in the following graph, $\{1, 2, 3\}$ and $\{3, 4, 5, 6\}$ are cliques, as is $\{2, 3\}$.

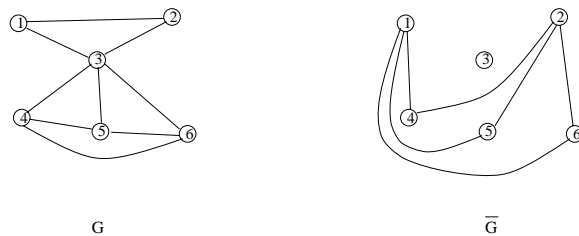


FIGURE 2. A graph and its complement.

The *complement* of a graph G , denoted \bar{G} , is the graph with the same nodes as G , but only the *missing* edges; see Figure 2 for an example.

Clique is NP-complete. This follows from the fact that a clique of size K for G corresponds to an independent set of size K for \bar{G} : If each pair of nodes in a set is connected by an edge (that is, the nodes form a clique), then in the complementary graph none of the nodes will be connected by an edge (so they form an independent set).

Finally, we consider the *multiprocessor scheduling* problem. There are m identical processors working in parallel. There are T discrete time slots, during which a set J of jobs must be scheduled. In addition, there are precedence constraints of the type: job j_i must be processed *before* job j_k . These constraints are represented by an acyclic directed graph (J, A) , where $(j_i, j_k) \in A$ means j_i must be scheduled before j_k .

A feasible schedule is a mapping S from the jobs to time slots, such that at most m jobs are scheduled during any one slot, and all jobs complete within the allotted time T .

Multiprocessor scheduling is NP-complete.

Proof. The problem is clearly in NP, since it is easy to check if a given schedule is feasible. We will now reduce CLIQUE to multiprocessor scheduling.

Consider a graph $G = (V, E)$ and an integer K . We will construct a set J of jobs, an acyclic digraph (J, A) representing precedence constraints, and integers m and T such that there exists a feasible schedule for J if and only if G has a clique of size K .

Let the set of jobs be given by

$$J = V \cup E \cup \{b_1, b_2, \dots, b_{n_b}\} \cup E \cup \{c_1, c_2, \dots, c_{n_c}\} \cup E \cup \{d_1, d_2, \dots, d_{n_d}\}$$

where the numbers n_b, n_c, n_d are chosen such that

$$(2.1) \quad k + n_b = \binom{K}{2} + |V| - K + n_c = |E| - \binom{K}{2} + n_d$$

and $\min\{n_b, n_c, n_d\} = 1$. Clearly such a choice is possible. Let m be the common value in (2.1) and set $T = 3$. (Notation: $\binom{n}{m}$ (read “ n choose m ”) is the number of subsets of size m in a set with n elements.)

To construct the precedence constraint digraph, first add to A all arcs of the form (b_i, c_j) and (c_j, d_k) . Then add all arcs (v, e) for all $v \in V$ and $e \in E$ such that e is incident on v .

We will now argue that the instance of multiprocessor scheduling just constructed has a feasible schedule if and only if G has a clique of size K .

Suppose first that there is a feasible schedule. Since $T = 3$, it must be that $S(b_i) = 1$, $S(c_i) = 2$, and $S(d_i) = 3$ for all i , since the b jobs must precede the c jobs which must precede the d jobs. Also, since $|J| = 3m = Tm$ by construction, all m machines must be busy during all three time slots. Now, in the last period, besides the d_i jobs, only $m - n_d = |E| - \binom{K}{2}$ jobs corresponding to edges can be executed, since if a vertex job were executed, then its incident edges could not be processed until period 4, violating the feasibility of the schedule. We conclude that the remaining $\binom{K}{2}$ “edge” jobs must be executed in the second period, and the

“vertex” jobs are executed in the first and second periods (with $m - n_b = K$ of them processed in the first period). Furthermore, the K vertices corresponding to jobs that are executed in the first period must include the endpoints of all $\binom{K}{2}$ edges correspond to jobs in the second period. But K vertices can include endpoints of $\binom{K}{2}$ edges only if they form a clique of size K . So the existence of a feasible schedule implies the existence of a clique of size K .

Conversely, suppose that G has a clique of size K . Define a feasible schedule S by: $S(b_i) = 1$, $S(c_i) = 2$, $S(d_i) = 3$ for all i , $S(v) = 1$ if v is a member of the clique and $S(v) = 2$ otherwise; $S([v, u]) = 2$ if v and u are in the clique and $S([v, u]) = 3$ otherwise. (Here $[v, u]$ is the job corresponding to the edge between vertices v and u , if there is one.) \square