The Complexity Classes P and NP

1 Introduction

Some problems have a minimum running time that is exponential in the size of their input simply because the size of their output is an exponential function of the size of the input. There is nothing we can do to reduce their time complexity because they must output exponentially many bytes. There are trivial examples of this:

- The problem in which, given n, all numbers from 1 to 2^n must be printed.
- The problem in which, given n, all $n!$ permutations of the numbers from 1 to n must be printed.

The first is very uninteresting and I am not sure why anyone would want a program to do that. The second is more useful because sometimes we want to generate all such permutations to use as input to another program, perhaps to test that program or to determine the average running time as a function of the input size. A program that prints all permutations of the set of integers from 1 to n, for any n, appears in Listing 1 below.

The fact that a program has an enormous amount of output does not mean the program itself is enormous, as you are well aware, because we can write a very small program with a "small" loop that produces infinite output.

Listing 1: Program that prints all permutation of 1..N for given N

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

/* genperms()  prints all permutations of numbers 1 to size */
void genperms( int a[], int size, int k )
{
    int i,j;
    static int depth = -1;

    depth++;
    a[k] = depth;
    if ( depth == size )  {
        for ( j = 1; j <= size; j++ )
            printf("%d ",a[j]);
        printf("\n");
    }
    for ( i = 1; i <= size; i++ )
        if ( a[i] == 0 )
            genperms(a, size, i);
    depth--;
```

```
    a[k] = 0;
}

int main ( int argc, char* argv[] )
{
    int n;  // number of permutations to generate

    /* Check usage  */
    if ( argc < 2 ) {
        printf("usage: %s n, where n is number of permutations\n",
               argv[0]);
        exit(1);
    }

    /* get user input and convert to number */
    errno = 0;
    n = strtol( argv[1], NULL, 10 );
    if ((errno == ERANGE  )  || (errno != 0 && n == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    /* allocate a dynamic array of size n+1 */
    int* p = malloc((n+1)*sizeof(int) );
    if ( NULL == p ) {
        printf("problem allocating storage\n");
        exit(1);
    }
    /* zero-fill the array */
    memset(p,0,n+1);

    /* call recursive function to generate permutations */
    genperms(p,n,0);

    free(p);
    return 0;
}
```

Problems such as the ones stated above, in which the size of the output is an exponential function of the size of the input, assuming we have some reasonable way to define size, are excluded from the remainder of the discussion here; they must take time proportional to the size of the output.

There are many problems that take a long time not because they produce an exponential amount of output, but because they require searching through an enormous number of candidate solutions to find the true solution. For example, imagine that we are given a complete graph with n vertices and we are told that each time we visit a vertex, we pick up a clue, and that if we pick up the clues in the correct order we win a big prize. The graph's vertices might be cities to visit, or trees in a forest, or train stations. The obvious solution, perhaps the only solution, is to generate all possible simple paths in the graph and check each one. The program in Listing 1 could be used to do just that, but instead of outputting each path, it would stop when it finds the path that collects the clues in the correct order. In the worst case it requires generating all permutations.

This is a rather simple problem and has a rather simple, though inefficient, solution. There are problems for which many known algorithms take an exponential amount of time to find a solution,

but for which no one has yet to find an efficient algorithm. We are about to reconsider our notion of "efficient" because as you will soon see, we have not really seen what a difficult problem is. So far we have treated an $O(n^3)$ algorithm or an $O(n^2)$ algorithm as an inefficient one if we could find an $O(n^2)$ or $O(n \log n)$ alternative algorithm respectively. For the problems we are about to describe, we would be quite happy with an algorithm that ran in $O(n^{40})$ time, because no one has yet to devise an algorithm that can run in time proportional to any polynomial function of n. The best that anyone has been able to do has been to find algorithms that solve these problems in exponential time, meaning in time proportional to c^n for some real number $c > 1$.

Is this bad? Consider a problem for which the fastest known algorithm that solves it runs in time proportional to 2^n. Suppose that with the fastest computers available today, it takes an hour to solve an instance of the problem of size $n = 100$. To solve an instance of size $n = 101$ would take 2 hours, and an instance of size $n = 116$ would take about 7.5 years. Suppose that in ten years, computers will run $100,000,000$ times faster. This would only allow us to solve a problem of size $n = 126$ in an hour's time! We would never be able to solve an instance of this problem of size $n = 200$ with this algorithm.

In general, we call problems such as these intractable:

Definition 1. A problem is ***intractable*** if an exponential amount of time is needed to discover its solution, and the size of its output is no more than a polynomial function of the size of its input.

1.1 Intractable Problems

What kinds of problems are like this, you might wonder. Some are very easy to state. The most well-known problem of this nature is called the ***traveling salesman problem*** (***TSP***), which can be defined as follows:

> A traveling salesperson wants to visit n distinct cities, starting in her home town and returning to her home town, but visiting every other city exactly once. The problem is to find a shortest route that visits every city exactly once and returns to the starting city.

As a graph problem it is stated as follows:

> Given an undirected weighted graph, find a shortest path that visits every vertex exactly once and returns to the first vertex.

It sounds quite simple, and it arises in a variety of contexts, but in fact there are no ways to solve this problem that do not need to find the lengths of a great many (exponentially many) paths through the graph, which are usually called ***tours*** when discussing this problem.

A similar problem is the ***Hamiltonian circuit*** problem:

> Given an undirected graph, is there any path that connects all of the nodes with a simple cycle?

A simple cycle is a cycle in which each vertex appears exactly once except the first and the last, which are the same vertex. For example, $1, 2, 3, 4, 5, 1$ is a simple cycle. It is not hard to cycle

through a graph and visit every node if you do not care about passing through nodes more than once. It is a completely different problem when you do not have this luxury.

The Hamiltonian circuit problem is an example of a decision problem.

Definition 2. A ***decision problem*** is a problem that has a yes/no answer.

The traveling salesman problem requires a path as its answer. The TSP and the Hamiltonian circuit problem are similar, but not exactly the same. The TSP as stated above is a minimization or optimization problem (one that wants a smallest value or a value that minimizes some function f subject to constraints on its variables), but it has a corresponding decision problem:

> Given a finite set of n cities and a bound B, does there exist a tour starting in her home town and returning to her home town, visiting every other city exactly once, of length at most B?

In Chapter 9 we defined spanning trees for graphs. A simple question to ask is whether a graph has a spanning tree in which no vertex has degree larger than some fixed constant k? This sounds like it should not be hard to solve, but it takes an exhaustive search as well. No one has found an efficient algorithm to solve it.

A problem not related to graphs is the ***bin-packing problem***:

> Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of n items of various sizes, $s(u_i)$, and a set of L bins into which these items must be placed, what is the least number of bins needed to pack all of the items?

A corresponding decision problem for this is

> Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of n items of various sizes, $s(u_i)$, $i = 1, 2, \ldots, n$, a set of L bins into which these items must be placed, and a capacity B, is there a distribution of items into bins such that the sum of the sizes of the items in each bin is at most B?

A second problem not related to graphs is called ***circuit satisfiability***:

> Given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?

Notice that this is also a decision problem and that it is another problem that requires exhaustive searching. In fact this problem is a variant of

Boolean satisfiability:

> Given a Boolean expression consisting of AND, OR, and NOT operators, is there an assignment of values to its variables that makes the expression true?

Sometimes there is a very fine line between an easy problem and a hard problem. A good example is the following pair of problems:

- Given a weighted graph $G = (V, E)$, two vertices v and w, and a weight c, is there a simple path from v to w whose weight *is at most* c.
- Given a weighted graph $G = (V, E)$, and two vertices v and w, and a weight c, is there a simple path from v to w whose weight *is at least* c. This is called the ***decision problem version of the longest weighted path problem***.

The first problem has an efficient solution; a breadth-first search of the graph will give an answer in linear time. The second has no known solution that runs in less than exponential time. The best that we can do is generate all possible paths and see whether any are sufficiently long.

2 Preliminaries

We now make more precise the difference between easy and intractable problems. To begin we have to make the idea of input size more precise.

Definition 3. The ***size of the input*** to an algorithm is the number of bits needed to encode the input, using a ***reasonable*** encoding method.

Reasonable generally means using the conventional methods, such as a binary encoding, hexadecimal or base ten or some other constant radix. An unreasonable encoding would be a unary encoding, requiring m bits to encode the number m.

Definition 4. An algorithm A ***solves a problem*** p if, for every input to the algorithm A, it terminates and outputs a solution to p for that input.

In this case we will simple say that algorithm A *solves* p, or that p *is solved by* A.

Definition 5. An algorithm A solves a problem p in ***polynomial time*** if there exists a number k such that, for every input of size n given to the algorithm, it solves the problem for that input in time at most n^k.

Notice that this implies a ***worst case running time***, because it says that for all inputs of size n, the upper bound on the running time is n^k. In particular, whichever one takes the longest is included.

Next we want to define what a deterministic algorithm is. We assume that all instructions have unique addresses, or numbers, and that all variables can be uniquely identified by their addresses in memory.

Definition 6. The ***state*** of an algorithm is a complete description of the values of all variables and memory locations used by the algorithm, together with the address of the instruction that it is about to be executed. The ***initial state*** of the algorithm is the state in which the first instruction is about to be executed for the first time[1] and all of its memory has been created.

Even things like open files are part of the state of an algorithm. If the algorithm has a file open, then the contents of the file are part of its state, as well as the address of the next byte in the file to be read or written. In essence, the state leaves nothing out that affects what might happen next in the computation.

[1] It might be re-executed at a later time.

Definition 7. An algorithm is ***deterministic*** if, given its current state, executing the current instruction in that state uniquely determines the algorithm's next state.

In other words, a deterministic algorithm leaves nothing to chance; given a particular input, it always produces the same output. In essence the collection of input/output pairs of the algorithm is a function. The algorithms we have learned about in this course, and probably most of the algorithms you have learned about in your other undergraduate courses, are all deterministic algorithms. Soon we will see what a nondeterministic algorithm looks like.

Lastly, for the remainder of this chapter, ***we will restrict all problems under consideration to be decision problems***.

3 The Complexity Class P

We define a very important collection of decision problems that have a common property.

Definition 8. The set of all decision problems that can be solved by a deterministic algorithm in polynomial time is the complexity class **P**.

Notice that **P** is a set of problems, not algorithms. **P** is called a ***complexity class***, because it is a class of problems that have the same computational complexity. The computational complexity of a problem is the asymptotic worst case running time of the best algorithm known to solve that problem. The definition of **P** states that a problem belongs to this class **P** if there is some deterministic algorithm that solves it in polynomial time.

Almost all of the problems that you typically study in an undergraduate program in computer science run in polynomial time. The ones that are decision problems belong to the class **P**. Many of the problems have decision problems that correspond to them. These includes problems such as sorting, all of the various shortest-path problems in graphs, algorithms to determine whether graphs are acyclic, strongly-connected, and so on, and searching for keys in trees and tables.

4 The Complexity Class NP

You now have to let your imagination loose a bit, to understand the idea of nondeterminism, because we will use it to describe an imaginary computer or algorithm, one that cannot exist as a physical reality. But it plays a very important role in theoretical computer science and dates back many decades. In a nondeterministic algorithm the next state that it enters is not determined by the current state; it is endowed with the ability to enter any one of a set of multiple next states, and this choice occurs, in a sense, simultaneously.

If a person is walking in a forest along a path that forks into two distinct paths, the person is forced to choose one of the forks and follow it. Suppose that, by some extraordinary means, the person could walk both paths simultaneously, not by making a copy of him or herself, but by cloning time. You might also think of this as cloning the world. Then the person would be walking nondeterministically. It is not the same thing as parallel execution, which would be like copying the person and having each copy walk a fork. It is not the same thing as a probabilistic algorithm, which would make a random guess and follow it. It is an abstraction that we can only imagine.

We apply this idea to algorithms. Imagine an algorithm that can guess a candidate solution to a problem in one stage and then check if the guess is correct in a second stage. We can call the first stage the ***guessing stage***, and the second, the ***checking stage***. The idea is that the algorithm can make all guesses simultaneously and then check them all simultaneously. This means, in effect, that if there is a solution to the problem, that solution will be one of the guesses, and the algorithm will discover this in its checking stage, and the total time it takes to do this is the same as if it made a single guess and checked it.

All of this can be formalized very rigorously, but to do so would require a fair amount of background theory. The original formulation was based upon Turing Machines. We do not need to use Turing Machines to explain the ideas we present here, but we lack rigor by not doing so. *Although it is not necessary to do so, for the remainder of this section we will restrict the problems under consideration to be decision problems.*

As an example, a nondeterministic algorithm could solve the traveling salesperson decision problem by first guessing an arbitrary sequence of cities that start in a given city, end in the same city and visit each city exactly once, and then in the checking stage, it would check whether the guess had length at most B. If there is a tour of length at most B, then one of the guesses will be correct and will be verified by the checking stage and the output will be *yes*. If there is no tour of that length, no guess will be found that is of the correct length, and the output will be *no*. We can make this more formal with the following definition.

Definition 9. A nondeterministic algorithm ***solves a decision problem*** d if,

- For each input to d for which the answer should be yes, there is a guess that it can make in the guessing stage that, when it is checked in the checking stage, will result in the answer "yes", and
- For each input to d for which the answer should be no, there is no guess that it can make in the guessing stage that when it is checked in the checking stage, will result in the answer "yes".

This is a reasonable definition. It basically says that the algorithm always finds a correct guess if it exists and cannot find one if it does not exist. We can define a polynomial-time nondeterministic algorithm as follows.

Definition 10. A nondeterministic algorithm whose input size is n ***solves a decision problem*** d ***in polynomial time*** if there exists a polynomial p such that, for every input to d for which the answer should be yes, there is a guess that leads in the checking stage to a yes answer within $p(n)$ time.

This implies that the size of the guess is also bounded by the polynomial, because it would be impossible to spend a polynomially-bounded amount of time checking a structure whose size was more than polynomial. We can now define the class **NP**:

Definition 11. The set of all problems that can be solved by a nondeterministic algorithm in polynomial time is the complexity class **NP**.

One immediate consequence of this definition is that every problem in **P** is also in **NP**: if a problem can be solved with a deterministic algorithm in polynomial time, it can be solved with a nondeterministic algorithm in polynomial time also. Therefore $\mathbf{P} \subseteq \mathbf{NP}$. The most important open question

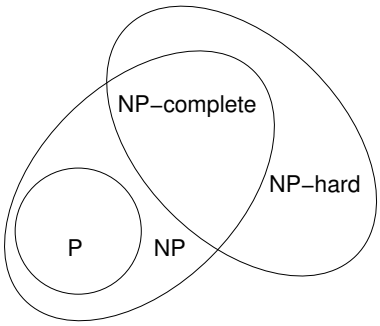

Figure 1: An Euler diagram showing the world if $P \neq NP$

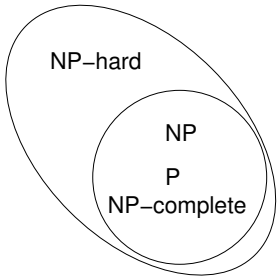

Figure 2: An Euler diagram showing the world if $P = NP$.

in computer science is whether **P** = **NP**, i.e., whether there are problems in **NP** that are not in **P**. No one has proved or disproved this statement, though many have tried. This is a big problem, because there are many important, practical problems known to be in **NP**, not yet known to be in **P**, and if someone were to prove that **P** = **NP**, this would imply that these problems did indeed have efficient solutions. If someone proved that **P** ≠ **NP**, this would not necessarily mean that any specific problem did not have an efficient solution, but that certain ones did not, which leads to the concept of ***NP-completeness***.

4.1 Exponential Time

If a problem is in **NP**, then it can always be solved by an algorithm that runs in exponential time deterministically. To see this informally, suppose that D is a problem in **NP**. Then there is some polynomial $p(n)$ such that instances of size n can be decided by a nondeterministic algorithm A that runs in time $p(n)$. This means that the size of the guess that is handed to the checking stage is no more than $p(n)$ bits in size, otherwise it would take more than $p(n)$ time to check the guess. A deterministic algorithm can enumerate every possible guess of at most that size and check each one in polynomial time. There are at most $O(2^{p(n)})$ such guesses, so such an algorithm can generate the guesses and check them in $O(2^{p(n)})$ exponential time. This shows that every problem in **NP** can be solved by an exponential time, deterministic algorithm.

5 NP-Completeness and NP-Hardness

There are certain decision problems that are known to belong to **NP**, but which may or may not belong to **P**. No one has found efficient algorithms for any of them, nor has anyone proved that

these algorithms cannot exist, and many people have tried to do one or the other. So there is no proof that they are in **P** and there is no proof that they are not. The list of such problems is quite large, but among them are:

1. circuit satisfiability, defined above
2. bin-packing, defined above
3. Hamiltonian circuit, defined above
4. subset sum: Given a set X of integers and an integer t, does X have a subset whose elements sum to t?
5. longest path: Given a non-negatively weighted graph $G = (V, E)$, an integer B, and two vertices u and v, is there a simple path from u to v in the graph of length greater than B?
6. graph isomorphism: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is there a one-to-one, onto function $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$?
7. integer factorization: Given an integer n and an integer m with $1 \leq m \leq n$, does n have a factor q with $1 < q < m$?

There are certain problems for which it has been proved that if any of them can be solved in polynomial time with a deterministic algorithm, then so can all problems known to be in **NP**, meaning that **P** = **NP**. Such problems are called ***NP-hard*** problems. Informally, a problem is NP-hard if it is at least as hard as any problem in **NP**. A problem is ***NP-complete*** if it is NP-hard and also in **NP**.

If someone were able to prove that any single problem known to be in **NP** could not be solved with a polynomial-time algorithm, i.e., that such a problem was in **NP** but not in **P** (**NP** − **P**), this would also imply that every NP-complete problem was in **NP** − **P**. These NP-complete problems are an important tool in understanding the relationship between these complexity classes.

Problems 1 through 5 above have been proven to be NP-complete. Problems 6 and 7 are known to be in **NP** but no one has proved they are NP-complete and no one has proved they are in **P**.

Are there decision problems that are NP-hard but not NP-complete? All undecidable problems fall into this category, such as the ***Halting Problem***[2], but there are also decidable problems that are NP-hard but not NP-complete. One example is called the ***true quantified Boolean formula problem***, which asks whether a boolean formula containing existential and universal quantifiers has an assignment that makes it true. See Figure 1 for a picture of how the sets are related if **P** ≠ **NP**.

5.1 Proving NP-Completeness

Suppose that a decision problem D is known to be NP-complete and we are given a problem D' in **NP**. Suppose that we can find a transformation T that can convert any instance d of the old problem D into an instance d' of the new problem D' in such a way that $d' = T(d)$ has a yes answer

[2]The Halting Problem, stated in informal terms, is the problem in which we are given a program and an input to that program and asked whether the program will eventually halt when run on that input. The original version was stated in terms of Turing Machines.

as an instance of D' if and only if d has a yes answer as an instance of D. Suppose also that the transformation can be done in polynomial time. Then we have a procedure for solving all instances of the original problem D as follows:

1. Transform the instance d into a problem $T(d)$ of D' using polynomial time.
2. Use the algorithm for D' to solve the problem $T(d)$.
3. Report the answer is yes for d if and only if the answer is yes for $T(d)$.

If D' has a polynomial time deterministic algorithm (meaning it is in P), this would imply that the old problem is also in P and that P=NP. In other words, D' is NP-complete.

Example 12. Suppose that we know that the Hamiltonian circuit problem is NP-complete but we do not know whether the traveling salesman problem is. The transformation we use is the following:

Given a graph $G = (V, E)$ with $|V| = n$ that is an instance of the Hamiltonian circuit problem, create a traveling salesman problem instance by creating a city for each vertex in V, and for the distances between the cities, define the distance between city v and city w to be 1 if there is an edge $(v, w) \in E$ and 2 if there is no edge (v, w). Clearly this is a polynomial-time transformation.

Run the algorithm for the traveling salesman problem to decide if there is a tour of weight at most n on this constructed instance. If it finds a tour, this implies that there was a cycle in the original graph that connected all of the vertices in the graph (because the traveling salesman visits all n cities, returning to the starting city, and so traversed all vertices and used n edges of weight 1 each.) If it finds no tour of weight at most n, then it could only find tours whose weight was greater than n. If the total weight of a tour is greater than n, at least one edge on the tour had weight 2. This edge corresponds to an edge that was not in the original Hamiltonian circuit problem graph, so the path does not exist in the original graph. Hence there are no paths in the original graph that form a simple circuit visits every vertex.

If the traveling salesman algorithm has a deterministic polynomial time algorithm, we can use it to solve the Hamiltonian circuit problem with this polynomial-time transformation, which implies that the Hamiltonian circuit problem also has a polynomial-time deterministic algorithm. Since Hamiltonian circuit is NP-complete, TSP must be NP-complete also.

The procedure described informally above is called a ***polynomial-time reduction*** of D to D'. If we can reduce a known NP-complete problem to another problem in NP, then the second problem must be NP-complete as well. This shows that we can extend the set of known NP-complete problems by finding suitable polynomial time reductions of known NP-complete problems. But this is only useful if there is a starting problem known to be NP-complete, one that was not proved to be NP-complete by reduction from some other NP-complete problem! How did it get there?

5.2 Cook's Theorem

In 1971, Stephen Cook proved the existence of the first NP-complete problem. He showed that the Boolean satisfiability problem is NP-complete, not by a reduction but by a direct proof. In short, he proved that if satisfiability has a polynomial-time deterministic algorithm, then so does every other problem in **NP**.

We cannot understand his proof unless we know what a Turing Machine is, but we can describe the basic idea. He showed how the computation performed by a nondeterministic Turing Machine (*NDTM*) could be expressed as a boolean formula in such a way that, the formula has an assignment that makes it true if and only if the Turing Machine outputs a solution to the given problem. Because a Turing Machine can be described in precise mathematical terms, it was possible to create a single boolean expression whose length was a polynomial in the size of the input to the machine such that the expression represented the running of the machine on that input in polynomial time. Because a *NDTM* can be used to solve any problem in **NP** in polynomial time, his construction showed that for every problem in **NP**, there was an instance of the satisfiability problem such that a yes answer to the satisfiability problem on that instance corresponded to a yes answer to the original problem. This established the first NP-complete problem, which is known as *SAT* for short.