1 Computational Complexity

Although Vertex Cover and Dominating Set are easy to state, they quickly become very difficult to solve as the size of the graph grows, even by the fastest computers. Formulating them as IP's does not help matters much; there are no fast algorithms for integer programming. These two problems share this property with a number of other classic problems in discrete mathematics.

There appears to be a natural division in the world of combinatorial problems between those that are easy and those that are hard. In this section, we make this idea more precise.

The study of these issues is called *computational complexity*. A general problem, such as Vertex Cover, is formally a collection of *instances* of the problem, in this case one for each particular graph $G = (V, E)$. An algorithm for solving the problem must be general enough to handle all possible instances as input. It is common to describe the complexity of an algorithm solely on the basis of its 'order of magnitude' in terms of growth, using what is called *big-O notation*. This is formally defined below.

Definition 1. *Let $\mathbb{N}$ represent the natural numbers (*i.e. *the positive integers) and $\mathbb{R}^+$ the positive real numbers, and let f and g be functions $\mathbb{N} \to \mathbb{R}^+$. Then $f(n)$ is $O(g(n))$ if $f(n)$ is eventually dominated by some positive multiple of $g(n)$; that is, there exist $m \in \mathbb{N}$ and $C \in \mathbb{R}^+$ such that $f(n) < Cg(n)$ for all $n \geq m$.*

For simplicity, we can restrict our attention to the class of *decision problems*, those for which the answer is either 'yes' or 'no'. Some examples are:

- Prime: Is a given positive integer composite (as opposed to prime)?
- Satisfiability: Given a compound logical expression, is there a set of truth values for the constituent propositions that make the entire compound expression true?
- 2-Colorability: Can all the vertices of a graph be colored with red and blue so that no edge joins two vertices of the same color?

Any optimization problem can be solved indirectly by solving a number of decision problems. For example, we could determine the minimum size of a vertex cover of a graph G by repeatedly asking if there exists a vertex cover of size k, for a series of intelligent choices of positive integer k. If we use binary search, then because the minimum vertex cover has size at most $n = |V|$, we need to solve such a decision problem at most $\lfloor \log_2 n \rfloor$ times. This is such a slow-growing function of n that the complexity of the optimization problem is essentially that of the associated decision problem. This tactic can in fact be applied to any integer programming (IP) problem.

A decision problem is said to belong to the class NP if a 'yes' answer can be quickly verified by means of a *certificate*. The letters NP stand for *nondeterministic polynomial-time algorithm*; the 'polynomial-time' refers only to how long it takes to check the certificate. For example, we could certify that, yes, n is composite by providing two integers s and t that satisfy $n = st$; this multiplication can be verified very quickly by a computer. We could certify that, yes, the compound expression p is satisfiable by providing a list of appropriate truth values for the constituent propositions; this can be quickly verified using binary multiplications and additions. We could certify that, yes, G has a vertex cover of size k by listing the vertices of one such cover; we then simply have to check that each edge has one endvertex in this set.

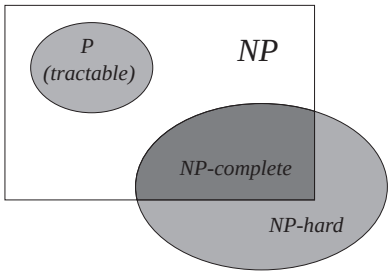
Checking a certificate is quite different from coming up with a certificate. A brute force approach to the k-Vertex Cover problem would involve investigating whether each k-subset of vertices is a cover; for a graph with n vertices, there are $\binom{n}{k}$ such subsets and in a worst-case scenario we would

have to test each one of them. A crucial point is that this number gets very large very quickly as n and k grow; while it may be possible to handle relatively small instances with this method, large instances become intractable even for the fastest computers of today and of the foreseeable future.

Exercise 1: For the values of n and k listed in the table below, compute how long a computer would take to perform $\binom{n}{k}$ elementary computations, assuming it can perform a billion such computations per second. Express your last three answers in terms of years.

n	k
50	10
100	10
100	50
200	50
200	100

For some problems, there are much faster - in particular, polynomial-time - alternatives to the brute force approach. For example, a polynomial-time algorithms exists for determining whether a graph is 2-colorable. In 2002, a polynomial-time algorithm was finally discovered for determining whether a number is composite or prime. This subclass of problems within the class NP is labeled P; more informally, such problems are called *tractable*. Another example of a problem in P is Maximum Matching, encountered in FIXME (old exercise 6). A famous open problem in mathematics asks whether in fact $P = NP$. That is, will we eventually find quick algorithms for solving *every* problem in NP?



One step towards answering this question is the identification of a set of problems that have the amazing property that if any *one* of them can be solved in polynomial time, then so can *every* problem in NP. Such problems are called *NP-hard*. A problem that is both NP-hard and is itself in NP is called *NP-complete*; see the Venn diagram below. In fact, these terms are often used interchangeably. The first problem shown to be NP-complete was Satisfiability (in 1971). Over the years, many bright minds spent a lot of time and effort trying to find efficient algorithms for various NP-hard problems with no success, and as a result the consensus among experts is that P is probably a proper subset of NP.

There are by now many optimization problems that have been shown to be NP-hard, among them Vertex Cover and Dominating Set. Formulating these problems as IP's does not help, because in general, integer programming is also NP-hard. However, we cannot simply give up on large instances of these and other NP-hard problems. They crop up in the real world in many

different fields, such as telecommunications and computational biology, where it is important to get some kind of answer. This has led to the development of techniques like branch and bound. It is important to now that while branch and bound is often successful at solving large-scale IP's in a reasonable amount of time, it comes with no guarantees. It is quite possible to bogged down investigating an exponentially increasing number of branches. Interestingly, however, general linear programming (LP) is in P!

A recent approach to NP-hard problems has focussed on determining what component of the input size is responsible for the computational intractability. For example, the first improvement in an algorithm for k-Vertex Cover had running time $O(2^k n)$; currently the best algorithm has complexity $O(1.271^k + kn)$. Note that in both cases only k contributes to the exponential growth in the running time. This leads to the following definition.

Definition 2. *1. A* parameterized *problem is one for which the size of an instance is an ordered pair (k, n); k is referred to as the* parameter *and n as the* main input size*.*

2. A parameterized problem is fixed-parameter tractable *(or FPT) if there exists an algorithm that solves the problem with complexity $f(k)p(n)$, where p is a polynomial in n (and f is an unrestricted function of k).*

(Note that the latest algorithm for k-Vertex Cover satisfies this definition, because for all positive integers n, $1.271^k + kn \leq (1.271^k + k)n$, and complexity is concerned only with putting an upper limit on the running time.) By contrast, there are no known algorithms for the k-Dominating Set problem that do any better than simply determining whether each possible k-subset of the vertices is a dominating set, and this has complexity $O(n^{k+1})$. In this case, we cannot isolate the parameter k in one multiplicative factor. The following exercise demonstrates what a difference this makes to running time.

> **Exercise 2:** For the values of n and k listed in the table in Exercise FIXME (old 16), compute the ratio $\frac{n^{k+1}}{2^k n}$.

One might hope that for k-Dominating Set, it is not k that is causing the computational complexity, but some other parameter. Maybe if we just reconfigure the input, with some other part of the input size playing the role of the parameter, we can get an algorithm of the required form. In fact, parameters need not even be numbers; they can reflect not just one but several structural properties of the input. However, it has been shown that it is highly unlikely that Dominating Set is FPT - as unlikely, in some sense, as $P = NP$.

Even though the table in the exercise shows that $2^k n$ is clearly more manageable than n^{k+1}, it is still possible for this number to get too large to be computable in a reasonable amount of time. However, a purely empirical observation is that for most parameterized problems encountered in the real world, the value of the parameter k is comparatively small (for example, under 100). This keeps the value of $2^k n$ manageable, while n^{k+1} remains unwieldy.

In fact, the advantage of having a small value of k gets compounded. A crucial feature of FPT problems is that they are particularly amenable to clever pre-processing techniques that (quickly) reduce each instance to a smaller instance of the same problem. If the reduction is drastic enough, it is still practical to apply even exponential-time algorithms to solve the smaller instance. The following makes this more precise.

Definition 3. *A parameterized problem P is* kernelizable *if there are rules for transforming any instance I of size (k, n) in polynomial time to another instance I' of size (k', n') satisfying $k' \leq k$ and $n' \leq h(k')$, where h is an (unrestricted) function of k'.*

The reduced instance I' is the 'kernel' of the original instance; implied in this definition is that a solution to the kernel I' can be 'lifted' to a solution of I. Note that the main input size n' of I' is bounded by a function of the parameter k', which is itself bounded by k; the main input size of the original instance has completely disappeared! This demonstrates that the time required to solve an instance of the problem ultimately depends *only* on the size of the parameter.

The result below shows that this reduction strategy works for all FPT problems.

Theorem 1. *A parameterized problem is kernelizable if and only if it is FPT.*

Hence, in applications where it can be assumed that the parameter is reasonably small, this class of NP-hard problems is for all practical purposes tractable.