1 Introduction

I'm just going to scratch the surface of a couple of areas that are NP-completeness adjacent, namely *fixed-parameter tractability* and *approximation algorithms*. This starts to explore the borderline between purely theoretical results and practical problems, particularly in the area of discrete optimisation.

2 Parametrised complexity and fixed-parameter tractability

In many of the problems that we found to be NP-complete there is more than one natural parameter. The full size of the problem (usually called n) is of course the controlling parameter, but there is often a secondary parameter, frequently describing the main item of interest. For example:

INDEPENDENT-SET
Instance: A graph G and a positive integer k
Problem: Is there an independent set in G having (at least) k elements?

Here the main parameter is the size of the graph (whether counted by edges or vertices – since these differ by at most a square), but the secondary parameter is the positive integer k.

Similarly:

SUBSET-SUM
Instance A sequence $v_1, v_2, \ldots v_n$ of positive integers and a target value t
Problem Is there a subset $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} v_i = t$?

The main parameter is (the amount of space needed to describe) the sequence of integers. The (space needed to describe the) target is the secondary parameter.

But it seems that the two parameters are much more closely intertwined in the first case than the second. Specifically, for INDEPENDENT-SET we know of no algorithms that are much better than brute force search and so have complexity $O(n^k)$. On the other hand, for SUBSET-SUM, the standard dynamic programming approach has complexity $O(tn)$, i.e., $O(2^b n)$ where b is the bit-length of t. Note that this means that for a fixed target we

have a linear-time algorithm (because we are thinking of the target as fixed in that case). But, there's a bit more to it than that in that this is not just true, but uniformly true – it's the same algorithm in each case.

From a practical standpoint this means that for reasonable values of the target (e.g., anything less than 1 billion on a standard 2024 laptop) we can solve SUBSET-SUM for *all reasonable* n. But, for independent set, even if we're just looking for 20 independent vertices, we're unlikely to be able to resolve all cases.

Time for a definition or two (see also parametrized complexity in wikipedia).

A subset $L \subseteq \Sigma^* \times \mathbb{N}$ is called a *parametrized problem* or *parametrized language*.

A parametrized language is *fixed-parameter tractable* (FPT) if there is an algorithm for recognising it whose running time on an instance (x, k) is bounded by $O(f(k) \times |x|^c)$ for some constant c where $f : \mathbb{N} \to \mathbb{N}$ is an arbitrary function.

Note that because we allow f to be arbitrary, we can just use k rather than the size of k as an argument since for once we don't care whether it's k, 2^k, or even $\left(2^{2^k}\right)$! (though, the latter if true and necessary would rather restrict our range of k that we could hope to cover practically).

The discussion above shows that the language consisting of positive instances of SUBSET-SUM is fixed-parameter tractable while it seems unlikely that the language corresponding to positive instance of INDEPENDENT-SET is.

There is a hierarchy of "increasingly difficult" parametrised languages called the W-hierarchy. In this hierarchy, the FPT languages are just $W[0]$. It turns out that INDEPENDENT-SET is in $W[1]$ (and is "as difficult as anything" in $W[1]$). The W-hierarchy is contained in NP, so it could collapse completely (if it turns out that $P = NP$).

What is the advantage of knowing that a problem (specifically, an NP-complete problem) is FPT? Mainly, that it opens up the opportunity to explore and improve algorithms for solving it, e.g., to change the associated function $f(k)$ to something that, while still not polynomial, is smaller than an obvious bound. Sometimes, in fact, in principle, always, we can also get algorithms of the complexity like $O(g(k) + n)$ and for those, the limits of practicality will be purely determined by how quickly $g(k)$ grows.

3 Vertex cover

The vertex cover problem is the poster child for FPT problems. A *vertex cover* in a graph is a subset of the vertices which contains at least one of the endpoints of every edge. We also say that a vertex *covers* an edge if it is one of the edge's endpoints.

VERTEX-COVER
Instance: A graph G and a positive integer k
Problem: Is there a vertex cover of G having at most k elements?

For FPT purposes we're really not much interested in problems unless they're in NP and NP-complete. The former is obvious for VERTEX-COVER - if you tell me a proposed vertex cover I can easily check both that it's not too big, and that it does actually cover every edge.

To prove that VERTEX-COVER is NP-complete is also not too difficult. We'll give a reduction from 3-SAT. Let an instance of 3-SAT be given containing v variables (x_1 through x_v) and c clauses (C_1 through C_v). We construct a graph on $2v + 3c$ vertices as follows:

- For each variable x_i there is a pair of vertices labelled x_i and $\neg x_i$ connected by an edge (call these the V-vertices for convenience).
- For each clause C_i there is a triangle of vertices labelled by the literals in the clause (call these the C-vertices).
- There are edges between each V-vertex and each C-vertex with the same label.

Now let's note the following:

- To cover the edges connecting V-vertices requires at least v vertices (since there are v disjoint edges of this type).
- Each individual triangle in the C-vertices requires two vertices to cover it, since if we used one or zero, there would be at least one uncovered edge - so in total this means $2c$ vertices.

So, even before we think about covering the edges between V-vertices and C-vertices we know that we'll require at least $v + 2c$ vertices for a vertex cover of this graph. Can we use exactly that many?

If the 3-SAT instance is satisfiable then we can! Specifically, we can take a satisfying assignment and use the corresponding V-vertices. For each clause this then covers at least one of the three edges connecting the corresponding triangle of C-vertices to the V-vertices, and therefore in that triangle we can choose the other two vertices to cover both the edges of the triangle and any remaining edges between it and the V-vertices.

On the other hand, if the instance is not satisfiable, we can't! That's because we know that we'd need to make v choices in the V-vertices, one per variable, and two choices in each triangle in the C-vertices. But, the first group of choices corresponds to a truth assignment. Since the instance is unsatisfiable, there is at least one clause unsatisfied. That means that none of the C-V connections from that clause's triangle has been made. So, choosing two vertices in that triangle leaves the third connection uncovered.

So that's a reduction of a 3-SAT instance to a VERTEX-COVER instance that can clearly be done in polynomial time, which implies that VERTEX-COVER is NP-complete.

Notably, the size of the vertex cover we were looking at ($v + 2c$) was a large fraction of the total number of vertices in the graph ($2v + 3c$). In some applications it will be clear that the vertex cover is only of interest if it's in some sense "small". That's exactly the situation where we'd like to know if the problem is fixed-parameter tractable. And, in this case, it is.

Here's a simple recursive algorithm $VC(G, k)$ to determine whether G has a vertex cover with k or fewer vertices:

if G has no edges **then**
  return true
else if $k = 0$ **then**
  return false
end if
Choose an edge vw of G
return $VC(G - v, k - 1) \vee VC(G - w, k - 1)$

Here, $G - v$ denotes G with v and all edges that v covers deleted. The algorithm is correct

because the base cases (we have no edges, so all good, or we have edges but no vertices left to use, so bad) are covered. For the recursive case, given an edge a vertex cover must include one endpoint or the other, and the remainder must be a vertex cover of the remaining graph.

What's the complexity of this algorithm? We surely do constant work checking the base cases, and easily can construct $G - v$ and $G - w$ in time proportional to n the size of G. So, the only question is how many recursive calls might be made – but the parameter k decreases by one at each point, and we make only two calls so that's 2^k (with possible off by one or two errors). In other words, the complexity is $O(2^k n)$ which is exactly the kind of thing we want to see to show that it's fixed-parameter tractable.

All this is very standard stuff too, so we already see that, in practice we could easily handle values of k up to 30 or so, for essentially any values of n.

Now that we've gotten started though, we can try and do better! One approach is to *reduce to a problem kernel*. This basically looks at the question: how large a graph can be "interesting" for a vertex-cover problem with parameter k? The main observation here is that if G has a vertex, v, with more than k neighbours and we're looking for a vertex cover of size at most k then, if it exists, it must include v. Otherwise, it would have to include every neighbour of v and that would be too many. So, we can start by using all these (reducing k as appropriate, e.g., if there are 3 such vertices we know we only have $k - 3$ left to use), repeating if possible until we either fail, or reach a problem instance where none of the vertices of the graph have more neighbours than the size of the cover we're looking for.

But now, let's look at a vertex count. If every vertex has degree at most k and we have a vertex-cover of size k then the edges that each vertex in the cover accounts for touch at most an additional k vertices. Together with the vertex in the cover, that makes $k + 1$ and since there are a total of at most k vertices in the cover that would give at most $k(k + 1)$ non-isolated vertices in the graph (this could occur for example if the graph consisted of k k-pointed stars). If we have more than that we can immediately conclude that no cover exists.

In time linear in n we can reduce to this kernel, and then we can use the previous technique on just the kernel. This gives a time bound something like $O(k^2 2^k + kn)$. Using

more graph-theoretic observations (mostly concerning vertices of low degree) and a fair bit of cleverness this can be reduced to the best current result of $O(1.28^k + kn)$ which means the algorithm is practical for (roughly) $k < 190$ and essentially arbitrary n.

4 Hard optimisation problems and approximation

An *optimisation problem* is one where we are given an *objective function* (generally to positive integers, or positive real numbers) on some collection of *feasible solutions* and are asked to identify the (or a) feasible solution that maximises or minimises the objective function. Perhaps the most famous such problem is the travelling salesman (or weighted Hamilton cycle) problem (TSP). Here we are given n vertices and a cost to "travel" from each vertex to any other. The problem is to find the cycle of all the vertices that minimises the total cost (which is the objective function). For our purposes we will also include the *metric condition* - thinking of the costs as providing a weighting of the complete graph then the shortest path between any two vertices is just the single edge between them.

Generally, when we want to say that an optimisation problem is hard we will need to show it is NP-hard rather than NP-complete. The issue is that these problems are generally not in NP (or at least not for any clear reason). That is, if we're told a feasible solution that the genie claims is optimal we have no way to check it. And, it's pretty easy to show that TSP is NP-hard by a reduction from Hamilton-cycle.

Let G be a HAMILTON-CYCLE instance. Construct the TSP instance where each edge of G is given weight 2 and each non-edge weight 3. The optimum TSP solution for this graph has weight $2|G|$ if and only if it uses only edges of G, i.e., if and only if G has a Hamilton cycle.

Frequently in optimisation problems "close enough is good enough". That is, we might be satisfied with a non-optimum solution if it comes with a guarantee that it's close to the true optimum. This leads to the idea of *c-approximation algorithms* formally defined as follows:

Definition 4.1. Let c be a constant greater than 1. An algorithm for an optimisation problem is said to be a *c-approximation* if it provides a feasible solution for which the value of the objective function is provably within a factor of c of the optimum.

TSP with the metric condition has quite a simple 2-approximation algorithm:

- Find a minimum weight spanning tree of G (easily done in polynomial time, see COSC201)
- Take as your cycle a depth-first traversal of that tree

In the cycle this produces, some consecutive pairs of vertices consist of tree edges, while others jump from a leaf of the tree to some other point. But, if we follow the path in the tree instead of the jump we wind up traversing each edge of the tree twice. So, by the metric condition, the cost of the cycle produced is at most twice the cost of a minimum spanning tree. On the other hand the optimum cycle with an edge removed is itself a spanning tree, so its cost is greater than the cost of a minimum spanning tree and therefore our produced cycle is within a factor of 2 of that optimum (as claimed).

With a little extra graph-theoretic cleverness (but not too much) this can be improved to the Christofides–Serdyukov algorithm which is a $3/2$-approximation algorithm for TSP.

It's worth noting that state of the art TSP-solvers do not generally provide a uniform approximation guarantee (like 2 or $3/2$ above) but are, in practice, within a few percent of the true optimum and that there are characteristics of the TSP problem that allow verification of this post hoc. That is, given an approximate solution it may be possible to show "you could only ever do at most 3% better" (or similar).

A contrasting case is MAX-3-SAT, the problem of maximising the total number of satisfied clauses in a 3-SAT instance. Random truth assignments satisfy $7/8$ of the clauses on average, and this can be converted to a deterministic polynomial time algorithm that is guaranteed to satisfy at least $7/8$ of the clauses - basically by just iteratively choosing the truth assignment in such a way as to maximise the expected number of clauses satisfied. Since at most all the clauses can be satisfied, this is an $8/7$-approximation algorithm.

A truly remarkable result in the field is due to Håstad who demonstrated in 2001 that if there is an algorithm in P that gives an $8/7 - \epsilon$ approximation for any $\epsilon > 0$, then $P = NP$. So, unless the landscape of efficient computability is very different from what we think it is, this is the best we can do.