Approximations to NP-Hard Problems

There is a sense in which all **NP**-complete decision problems are the same. If A and B are **NP**-complete, it follows that $A \leq_p B$, and hence that we can *translate* any instance of problem A into an instance of problem B with the same answer. The *natural* **NP**-complete problems (ones that are posed for their own sake, rather than being constructed as potential counterexamples) have the additional property of being *isomorphic*. This means that the poly-time reduction can be made to be one-to-one and onto, and makes the problems in this sense simply re-encodings of each other.

But there are ways to look at **NP**-complete problems that make them different. In this section of the course our main concern will be **approximation** of optimization problems – search problems where our goal is to get a solution that is within some fraction of the best possible solution. In some cases we will find good general approximation algorithms, and in others we will be able to prove that the approximation problem is itself **NP**-hard. We will define several categories of optimization problems based on the difficulty of approximating them.

First, though, I want to give a brief example of a difference between **NP**-complete *decision* problems, one that opens up an important research area in the theory of algorithms we won't have time to explore, called **fixed-parameter intractability**. Consider the CLIQUE and VERTEX-COVER problems. Each takes as input a pair (G, k), where G is an undirected graph and k is a number, a **parameter** to the problem, giving the size of the clique or vertex cover we seek.

Recall that the reduction from CLIQUE to VERTEX-COVER takes a pair (G, k) to a pair $(H, n - k)$ where H is the complement graph of G and n is the number of vertices in G or H. It is therefore not so surprising that the behavior of these two problems for *particular* values of k might be different.

Consider the case where k is a constant. If we are looking for a k-clique in G, the obvious way to proceed is to check all $\binom{n}{k}$ sets of size k to see whether any of them is a clique. This takes $O(n^k)$ time, which is bad for all but very small values of k. There is no known algorithm that is appreciably faster, and there are theoretical reasons to think that none exists.

By contrast, consider the case of VERTEX-COVER where k is a constant. If there is a vertex cover of size k, it should be clear that there can be at most $k(n-1)$ edges in the graph, because every vertex has degree at most n. So our first move is to count the edges until or unless we get past $k(n-1)$ – if we do we return ”false”.

Supposing it is still possible that the vertex cover exists, we consider an arbitrary edge (u, v). This edge must be covered, so either u or v must be in the vertex cover. If u is in it, there must be $k-1$ other vertices that cover the edges that aren't incident on u. If v is in it, there must be $k-1$ vertices to cover the edges not incident on v.

We can thus apply a *recursive* algorithm to the two graphs $G \setminus \{u\}$ and $G \setminus \{v\}$, looking for a vertex cover of size $k-1$ in either. Each recursive call will begin by counting the edges in its graph, rejecting if there are more than $(k-1)(n-2)$, and otherwise picking an edge and making two calls on the $k-2$ version of the vertex cover algorithm. If we reach a subcase with no edges, or where we ask for a vertex cover of size 0, we accept.

We make up to 2^k recursive calls, each of which takes $O(kn)$ time if we consider k as a parameter rather than a constant. The total time is thus $O(2^k kn)$, which is *linear* in n for any particular value of k. In particular, look at the two time functions for CLIQUE and VERTEX-COVER when $k = 10$. As far as we know, deciding whether a 10-clique exists takes $O(n^{10})$ time, while deciding whether a 10-vertex cover exists takes $O(10240n)$ time, much better for large n.

We need to study the variety among **NP**-complete problems because we need to know what to do when we are faced with an **NP**-complete problem.

- It may be that what we really want to solve is a tractable subcase of an **NP**-complete problem. Remember that the SUBSET-SUM problem is **pseudopolynomial** – if the numerical parameters are polynomially bounded, then so is the running time. Unfortunately, most **NP**-complete problems are **strongly NP-complete**, meaning that even with small numbers they are not in **P** unless $\mathbf{P} = \mathbf{NP}$. An example is BIN-PACKING, where we have integer-sized items and bins and want to fit the items into a minimum number of bins.
- Sometimes there is a probability distribution on the inputs to the problem, and the average-case time turns out to be polynomial even though the general problem is **NP**-complete. But this is a relatively rare phenomenon, and one that has to come from nature – as far as we know **NP**-complete problems remain difficult for randomized algorithms.

- If we can't expect the exact optimum, we may be able to approximate its performance. The happiest situation is when we have an algorithm that will get us to within as close a multiplicative factor as we like and still be polynomial (with time still depending on the error bound) – this is called a **poly-time approximation scheme**). But this isn't always possible – we want to **classify NP-**complete optimization problems by how hard they are to approximate.

Approximating VERTEX-COVER

Let's now look at approximation algorithms for a particular **NP**-hard optimization problem, VERTEX-COVER. We are given an undirected input graph G and want to find a vertex cover of G that is as small as possible. We know that it is **NP**-hard to get the exact size of the smallest vertex cover, but maybe we can get a reasonably small one.

Let's first consider a natural greedy approximation algorithm. We'll choose a vertex of maximum degree in the graph, thereby covering as many edges as possible with one vertex. Then we recurse on the remaining graph, until all the edges are covered.

The Adler notes give an example where this algorithm does badly. We have a graph with $O(m \log m)$ vertices arranged in m columns. The first column has m vertices, the next (the floor of) $m/2$, the next (the floor of) $m/3$, and so on. Columns $(m/2) + 1$ through m have one node each. By the harmonic number estimate we saw earlier, the total number of nodes is $O(n \log n)$.

Now we have to define edges for our graph. Every vertex in column k has k edges to distinct vertices in column 1, and these are the only edges in the graph. Because of the floor operation on the size, some of the vertices in column 1 don't get an edge from every other column. But the first $m/2$ vertices in column 1 have an edge from every other column and thus have degree $m-1$.

Since every edge involves a vertex in column 1, there is a vertex cover of size m, consisting of every vertex in column 1. What does the greedy algorithm do on this graph? It first takes the single vertex in column m, which has degree m. After this is gone, the vertices in column 1 all have degree at most $m-2$, so the algorithm takes the vertex in column $m-1$. In the same way, there is always a vertex in the last column whose degree is one greater than that of any vertex in column 1. The greedy algorithm does not get all the edges until it has killed every vertex *except* the first column.

We can measure the performance of the algorithm by dividing its score, the number of vertices it uses, by the optimal score, the number of vertices in the optimal vertex cover. This **approximation ratio** is $\Omega(m \log m)/m = \Omega(\log m)$. We'll normally be much happier with a constant approximation ratio, some number greater than 1.

Why do we express the ratio in Ω terms rather than big-O? We haven't ruled out the possibility that this greedy algorithm isn't *even worse* on some other graph.

Let's now look at a better method of approximating the minimum vertex cover. We already know something about **matchings** in an undirected graph – sets of edges that are vertex-disjoint. Using network flow, we can find a maximum matching in a bipartite graph, and by another algorithm we didn't present, a maximum matching in any undirected graph can be found in polynomial time.

But to approximate vertex cover, it suffices to find a *maximal* matching, which is easy to do by a greedy algorithm in linear time. Simply take any edge (u, v), add it to the matching, and then remove vertices u and v from the graph, together with any other edges they touch. Eventually we will be left with an empty graph, and we then know that any other edge in the original graph shares a vertex with one of our matching edges.

Let M be the set of m edges in our maximal matching and consider the set C of all $2m$ vertices touched by those edges. We claim that C is a vertex cover. Any edge in the original graph must contain at least one vertex in C, or we could have added it to the matching and thus M would not be maximal. So we have a vertex cover with $2m$ vertices.

How small could the minimum vertex cover be? To cover an edge (u, v) in the matching, a set must contain either vertex u or vertex v. No vertex can cover more than one of the edges in M, so a valid vertex cover must have at least m edges. Thus C is within a factor of two in size of the minimum vertex cover.

If the input graph is *itself* a matching, then the only maximal matching is the entire graph. In this case the performance ratio of the algorithm is 2 and no better, because it uses $2m$ vertices and there is a vertex cover with only m vertices, consisting of one vertex from each edge.

In fact this is close to the best approximation algorithm known for the minimum vertex cover problem. Hastad has proved that it is **NP**-hard to approximate minimum vertex cover within a ratio of 1.1666 or below.

Approximating MAX-IND-SET

We've seen that while VERTEX-COVER and IND-SET are very similar as decision problems, the behavior of their parameters is quite different. When we consider the two as optimization problems and view the difficulty of approximating them, we find further differences.

First recall the direct relationship between the two problems. An undirected graph G with n vertices has a vertex cover of size k iff it has an independent set of size $n-k$. This is because the complement of any vertex cover must be an independent set, and vice versa.

We can find a vertex cover that is at most twice as big as the optimal one. This gives us a sort of approximation for the optimal (maximum) independent set, but (as it turns out) not a very good one.

Consider a graph with $2k+1$ vertices in k connected components: $k-1$ of size two, with single edges, and one of size 3 (a path of length 2). The minimum vertex cover has size k, taking one vertex from each of the size two components and the middle vertex of the 2-path. Thus the maximum independent set has size $k+1$, with one vertex from each 1-path and the two endpoints of the 2-path.

How well does our approximation algorithm do in this case? It finds a maximal matching, which must consist of the $k-1$ 1-path edges and one of the edges of the 2-path, for k edges in all. It takes the $2k$ endpoints of these edges to get a vertex cover of size $2k$, exactly double the size of the minimum vertex cover.

But the new independent set has size only 1! This is enormously worse than the maximum independent set of size $k+1$ – not within any constant factor at all as k increases. The problem is that the error, which was comparable to the optimal score in the vertex cover case, is much larger than the algorithm's score in the independent set case even though it is the same set of vertices.

So the VERTEX-COVER and MAX-IND-SET optimization problems have very different behavior with respect to approximation. We *can*, however, have a mapping between problems that preserves the optimization properties. Remember the reduction from IND-SET to CLIQUE or vice versa, which mapped (G, k) to (H, k) where H was the complement graph of G. The same mapping from G to H takes instances of the optimization problem MAX-CLIQUE to instances of MAX-IND-SET, and vice versa.

Here an approximate solution to one optimization problem on G maps to an equally good solution to the other problem on H, because our mapping preserves the quality measure of any solution exactly. This problem is usually called MAX-CLIQUE in the literature, and is known to be **NP**-hard to approximate even within a factor of n^ϵ for some positive constant ϵ.