COSC 341
Theory of Computing
Lecture 24
Parameterized complexity and approximation

Stephen Cranefield
stephen.cranefield@otago.ac.nz

Lecture slides (mostly) by Michael Albert

Keywords: Parameterized complexity, fixed-parameter tractability, approximation ratio.

What's it all about?

- The basic parameter that we take into account when describing the difficulty of various problems is the input size.
- Many problems have one or more secondary parameters (which may be explicit or implicit).
- For example:
  - In INDEPENDENT-SET, the size, k, of the independent set being sought is a secondary parameter.
  - In SUBSET-SUM, the target, t, or its bit-length is a secondary parameter.

So what?

- Dynamic programming solves SUBSET-SUM in time $O(tn)$, i.e., linearly in the full size of the problem, but exponentially in the size of the parameter. In particular, for small to moderate values of t this is a perfectly practical algorithm for essentially all n.
- On the other hand, for INDEPENDENT-SET we don't know a much better approach than "check all possibilities" which is $O(n^k)$ and is not practical for all values of n even for relatively small k.
- The goal of *parameterized complexity* is to try and understand this difference and its consequences.
- It aims to determine when a problem is fixed parameter tractable, i.e. it can be solved in polynomial time if one parameter is 'fixed' to a constant value.

Fixed-parameter tractability defined

The idea behind fixed-parameter tractability is to take an NP-hard problem, which we don't know any polynomial-time algorithms for, and to try to separate out the complexity into two pieces—some piece that depends purely on the size of the input, and some piece that depends on some "parameter" to the problem. . . .

Generally speaking, a problem is called fixed-parameter tractable if there is some algorithm for solving the problem defined in terms of two quantities: n, the size of the input, and k, some "parameter", where the runtime is:

$$O(p(n)f(k))$$

where $p(n)$ is some polynomial function and $f(k)$ is an arbitrary function in k. Intuitively, this means that the complexity of the problem scales polynomially with n (meaning that as only the problem size increases, the runtime will scale nicely), but can scale arbitrarily badly with the parameter k. This separates out the "inherent hardness" of the problem such that the "hard part" of the problem is blamed on the parameter k, while the "easy part" of the problem is charged to the size of the input.

Example problem: VERTEX COVER

A vertex cover in a graph G is a set of vertices such that every edge has at least one endpoint in the set.

VERTEX COVER
Instance: A graph, G, and a positive integer k.
Problem: Does G have a vertex cover of size k?

Sometimes called DATA CLEANING:

- Vertices are experimental observations
- Edges are between incompatible observations
- Can we clean the data by eliminating a small set?

VERTEX COVER is NP-complete

Given an instance of 3-SAT with v variables and c clauses construct the following graph on $2v + 3c$ vertices:

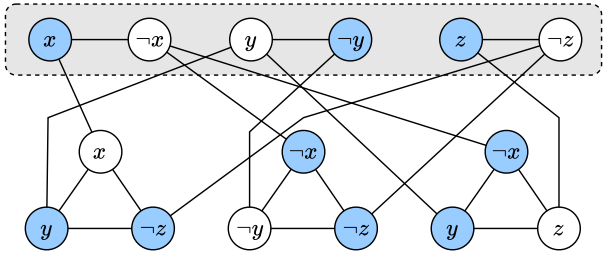
- For each variable, x, two V-vertices labelled x and $\neg x$ connected by an edge.
- For each clause c three C-vertices forming a triangle whose vertices are labelled by the literals in the clause.
- Edges between the V-vertices and the C-vertices that correspond to the same literal.

See next slide.

Theorem

The 3-SAT *instance is satisfiable if and only if the graph above has a vertex cover of size* $v + 2c$.

Example reduction from 3-SAT to VERTEX COVER



The graph represents the reduction of a 3-SAT instance $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z)$ to a vertex cover problem.

The blue vertices form a minimum vertex cover, and the blue vertices in the gray region correspond to a satisfying truth assignment for the original formula.

Diagram from Wikimedia (public domain)

Proof of theorem

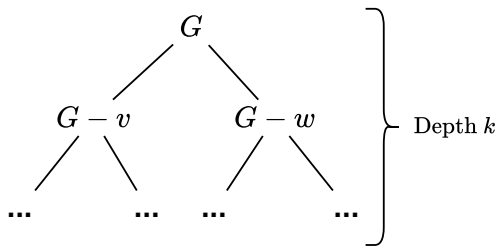
A satisfying assignment can be represented by including exactly one literal in each variable gadget in the vertex cover. In each clause, at least one literal is true under that assignment, so it is adjacent to a vertex that is already in the vertex cover. To cover the three edges in each clause gadget, the vertices for its other two literals must be in the vertex cover. That gives a vertex cover of $v + 2c$ vertices.

In the other direction, suppose a vertex cover of that size exists. At least one vertex in each variable gadget must be included to cover its edge. The other $2c$ vertices in the cover must be in the clause gadgets as that is the only way to cover all three of each gadget's edges. The variable assignment for the 3-SAT instance is given by the vertices of the variable gadgets that are in the vertex cover. The graph's construction ensures that this satisfies each clause.

An $O(2^k n)$ algorithm

To construct a k-vertex cover of G or prove that one doesn't exist.

- Choose an edge e with endpoints v and w.
- Delete v from G and recurse ($k-1$ more times). If you ever reach a graph with no edges, you have found a vertex cover.
- Delete w from G and recurse ($k-1$ more times). If you ever reach a graph with no edges, you have found a vertex cover.
- If neither above succeeded, then no k-vertex cover exists since any vertex cover must contain either v or w.



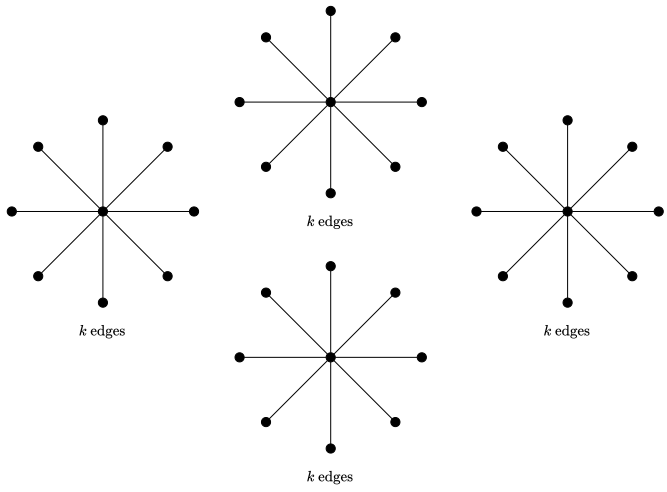
How low can we go? (1)

There are two qualitatively different ways to improve the binary search approach:

- One is *reduction to a problem kernel* (text modified from Wikipedia, Licence CC BY-SA 4.0)
  1. If $k > 0$ and v is a vertex of degree greater than k, remove v from the graph and decrease k by one. Every vertex cover of size k must contain v since otherwise too many of its neighbours would have to be picked to cover the incident edges. Thus, an optimal vertex cover for the original graph may be formed from a cover of the reduced problem by adding v back to the cover.
  2. If v is an isolated vertex remove it because it cannot cover any edges.
  3. If more than k^2 edges remain in the graph, and neither of the previous two rules can be applied, then the graph cannot contain a vertex cover of size k. For, after eliminating all vertices of degree greater than k, each remaining vertex can only cover at most k edges and a set of k vertices could only cover at most k^2 edges. In this case, the instance may be replaced by an instance with two vertices, one edge, and $k = 0$, which also has no solution.
- Applying these rules repeatedly terminates in linear time with a "kernel" graph with at most k^2 edges and $2k^2$ vertices. We can solve this in $O(2^{2k^2} + n + m)$ time for a graph with n vertices and m edges. This is efficient when k is small.

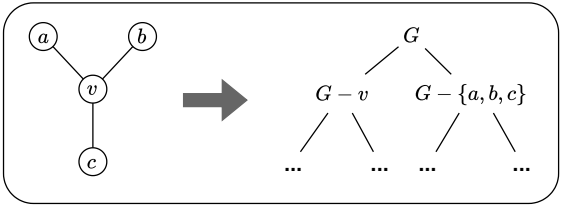
How low can we go? (2)

The maximum number of edges in a graph with k vertices of degree at most k is k^2



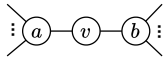
How low can we go? (3)

- The other aproach is to analyse more carefully, and be a bit clever about the recursive choices made—this has the effect of lowering the 2 in 2^k to some smaller value.



Throw out vertices of degree 1. Their neighbour may be a better choice (and no worse)

Vertices of degree 2:



Use $\{a, b\}$ instead of v

- Combining this with the reduction to a problem kernel gives the best current result of something like $O(1.28^k + kn)$ which means the algorithm is practical for (roughly) $k < 190$ and essentially arbitrary n.

When close enough is good enough

In optimisation versions of $\mathbf{NP}$-complete problems, we may be happy with (good enough) approximate answers. Sometimes this is easy, sometimes it's hard, sometimes it's in between!

Given an optimisation problem, and a constant $c > 1$, a c-approximation algorithm is one that produces a feasible solution (i.e., an object of the desired type) whose objective value is within a factor of c of the true optimum.

For maximisation problems, we consider $optimal\ \ performance/performance$ and for minimisation we consider $cost/optimal\ \ cost$.

Weighted Hamilton cycle

- Given a complete weighted graph G (positive weights).
- Find a cycle that visits every vertex and has minimum total weight.
- Assume that the least weight path between any pair of vertices is the direct edge between them (the “metric assumption”).

This is $\mathbf{NP}$-hard.

- Reduce from HAMILTON CYCLE
- Take a given graph G
- Make the weighted graph where all the edges of G have weight 2 and the non-edges have weight 3.
- The minimum total weight of a weighted HC is $2|V(G)|$ if and only if G has an HC.

2-approximating weighted Hamilton cycle

- Find a minimum spanning tree (MST) of G (several polynomial time algorithms exist—see Wikipedia)
- Traverse the tree depth first then return to the root. We traverse each edge of the MST twice so the cost is twice the MST cost (i.e. the sum of its edge weights). The order that nodes are visited defines a Hamiltonian cycle.
- Whenever two adjacent vertices in the cycle were visited by the search in a suboptimal way (backtracking up a subtree and back down another), we could instead have taken a shorter path (e.g. the direct edge between them). Given the metric assumption, the shorter path can cost no more than the path taken in the depth-first search.
- Thus our Hamilton cycle costs no more than twice the sum of the MST cost.
- The minimum possible cost of *any* Hamilton cycle is more than the cost of an MST since a Hamilton cycle with one edge removed is a spanning tree. Therefore *MST cost* $<$ *min. HC cost* $\leq 2 \times$ *MST cost*, giving our factor of 2 approximation.

MAX-3-SAT

- Given a 3-SAT instance, find a truth assignment that maximises the number of satisfied clauses.

How well can we approximate this?

- We know an algorithm in $\mathbf{P}$ that gives an $8/7$-approximation.
  See Q5 in Tutorial 21. The probabilistic algorithm can be turned into a deterministic algorithm. The maximum fraction of satisfied clauses we could hope for would be $8/8$ and $7/8$ is within a factor of $8/7$ of that.
- Håstad demonstrated in 2001 that if there is an algorithm in P that gives an $8/7 - \epsilon$ approximation for any $\epsilon > 0$, then $P = NP$!
- That is beyond the scope of this paper!

Some tips for finding reductions

Quote from StackExchange user JeffE at https://cs.stackexchange.com/q/1249:

> *First, . . . you have to decide which NP-hard problem to reduce to your problem. This is largely a question of "smell". If the number 3 appears anywhere in the problem statement, try reducing from 3SAT or 3Color or 3Partition. (Yes, I'm serious.) If your problem involves finding an optimal subsequence or permutation or path, try reducing from HamiltonianCycle or HamiltonianPath. If your problem asks for the smallest subset with a certain property, try Clique; if it asks for the largest subset with a certain property, try IndependentSet. If your problem involves doing something in the plane, try PlanarCircuitSAT or PlanarTSP. And so on. If your problem doesn't "smell" like anything, 3SAT or CircuitSAT is probably your best bet.*