Exact Algorithms
Lecture 6: Parameterized Complexity Theory – Hardness

26 January 2021 Lecturer: Michael Lampis

Summary: This lecture covers the following:

- Reminder of basics of complexity theory (NP-completeness, P, NP, coNP, PSPACE)
- Introduction the terminology of parameterized hardness (FPT reductions, W-hierarchy)
- Introduction of the basic hardness assumption that k-Clique is not FPT.
- Implications via reductions, notably k-Dominating Set is not FPT.

1 Proving that Something Cannot be Done

The topic of today's lecture is how to prove that certain problems are **hard**, in the sense that **no algorithm** can guarantee to solve them exactly within a certain amount of time. This area is generally called **algorithmic lower bounds** or **algorithmic hardness theory**. Before we begin, let us recall some basic notions of complexity theory, which we will revisit in a more careful way. We will then move on to see equivalent notions in the framework of parameterized complexity.

2 Reminder of Basic Concepts

Recall that a *Decision Problem* is a function $P : \{0,1\}^* \to \{0,1\}$. In other words, a problem is a function that takes as input a string of bits and must decide on a YES/NO answer. The basic question of complexity theory is "Which Decision Problems can be decided efficiently?". In other words, for which problems is it possible to design an efficient algorithm that given an input $I \in \{0,1\}^*$ correctly computes $P(I)$?

What do we mean by an "efficient" algorithm? The classical setting is that, for a specific input, the performance of an algorithm is the number of steps the algorithm needs to terminate on that input. The time complexity $T(n)$ of an algorithm is a function which, for any n, tells us the maximum (i.e. worst-case) number of steps the algorithm needs for any input of n bits. We are mostly interested in $\lim_{n\to\infty} T(n)$, that is, we are interested in *asymptotic worst-case analysis*.

The classical definition of an efficient algorithm is the following: an algorithm is efficient if its running time $T(n)$ is a polynomial in the input size n. Therefore, the classical approach is to consider algorithms with running times such as n^2, $n^3 \log n$, n^{232} etc. as "efficient", while algorithms with running times such as $2^n, 1.1^n, 2^{\sqrt{n}}, n^{\log n}$ are considered "inefficient".

The above definition of "efficient" is motivated by the realities of computer science in the last few decades, and in particular by Moore's law. Given that hardware speed has been known to increase exponentially in time, if for a given problem we have an algorithm that runs in polynomial time (even a terrible polynomial), the size of instances we can handle effectively will grow exponentially with time. Therefore, for any such problem we can sit back, relax, and let the progress of hardware lead to practical solutions of large instances of the problem. For problems for which the best known algorithm is super-polynomial on the other hand, (e.g. 2^n) the progress of hardware technology is not enough: doubling the available computing power only allows us to move from instances of size n to size $n+1$, meaning the size of instances we can solve only increases linearly, rather than exponentially.

Basic Question 1: Can we compute everything efficiently?

Question 1 is one of the few questions that complexity theory has a convincing answer to at the moment. Unfortunately, the answer is no.

Theorem 1. *There exists a function $P : \{0,1\}^n \to \{0,1\}$ such that no algorithm can correctly compute P in time n^c, for any constant c. In fact, a random function has this property with high probability.*

Proof Sketch:

The main idea of this proof is by a counting argument. There are 2^{2^n} different functions from $\{0,1\}^n$ to $\{0,1\}$. How many different polynomial-time algorithms are there? One way to count this is to observe that any computation can be simulated by a digital circuit, with AND, OR and NOT gates (in fact, this is how actual computers work). A computation that takes time n^c must correspond to a circuit with at most n^c gates. How many such circuits exist? By looking at the circuit as a graph we can see that there are at most $2^{n^{c'}}$ different circuits. Since this is much smaller than 2^{2^n} (when n is large) we have that for most function, no corresponding polynomial-size circuit exists. Therefore, a random function is, with high probability, impossible to compute in polynomial time.

□

Despite looking rather disappointing initially, the above theorem is not necessarily bad news. Sure, most Problems are hard to solve. However, most problems are also uninteresting. One could argue that the problems for which we actually want to design algorithms (for example because they have some practical application) are not just "random" problems. They must have some special structure. So the real question is, can we solve these efficiently?

Basic Question 2: Can we compute all "interesting functions" efficiently?

Basic Question 3: What makes a function "interesting"?

A situation that often arises in combinatorial optimization is the following: we are given some input (e.g. a graph) and we want to find some *solution* that achieves something in that input (e.g. we want to find a coloring of the graph's vertices, or an ordering that visits them all). This situation hides within it two separate problems:

1. Given input $x \in \{0,1\}^n$ find a "solution" $y \in \{0,1\}^m$.
2. Once a solution y is given, *verify* that indeed y is a correct solution to x.

Take a minute to think, which of the two is harder? The first sub-problem is what we call the search problem, and the second is called the verification problem.

We say that a problem belongs in the class P if both of the above questions can be solved by a polynomial-time algorithm. We say that it belongs in NP if the verification question can be solved in polynomial time, and if $|y| = poly(|x|)$ for all inputs.

More informally, the class NP contains all problems with the following property: if someone gives us a solution, we know of an efficient way to verify that it is correct. Does this property help us say anything interesting about these problems?

Observe that most natural questions we have seen about graphs are in NP. For example:

1. Is this graph 3-Colorable? (The certificate/solution is a 3-Coloring)
2. Is this graph bipartite? (The certificate/solution is a bipartition)
3. Does this graph have $\alpha(G) \geq B$? (The solution is an independent set of size B)
4. Is this graph chordal? (What is the certificate?)

Note: Can you think of any natural computation problems which do not belong in NP? (This is related to harder complexity classes, such as PSPACE).

Note: Is the following problem in NP? Given a graph, decide if it's NOT 3-Colorable. What is the certificate? What is the difference with deciding if a graph IS 3-Colorable? (This has to do with the class coNP).

Though the above notes point out that there are in fact some interesting functions outside of NP, NP contains all of P, and it also contains the vast majority of interesting optimization questions. Therefore, it becomes a natural question: which problems in NP can we solve efficiently? Perhaps all of them?

Basic Question 4: Is P=NP?

The smart money bets on No. The basic mathematical intuition is that, when given an equation (e.g. $x^2 + 5x - 14 = 0$) it is much harder to find one of its solutions, than to verify that a specific number (e.g. $x = 2$) is a solution. Similarly, one would expect that, given a graph, it should be harder to find a 3-Coloring, that to just check that a 3-Coloring is correct. Furthermore, much effort has been expended in the last 50 years on designing efficient algorithms for problems which are in NP. So far, for many of them, no one has come up with a polynomial-time algorithm.

Nevertheless, any class on complexity theory must at some point mention that even the basic question of P=NP is at the moment open: it is still possible (though believed very unlikely) that there is a clever way to solve *all* problem in NP in polynomial time. For example, the counting argument that we used to prove that some functions are hard, does not apply here, since each function in NP has some associated polynomial-time verification algorithm (and therefore, there are not 2^{2^n} functions in NP, but much fewer).

2.1 Reductions and NP-Completeness

So far, we have not been able to prove that any problem in NP is outside of P. However, we have been able to prove statements of the following form: "If problem A is in P, then so is problem B". Such conditional statements are proven using **reductions**. The idea is that we show the following: suppose that we had a hypothetical polynomial-time algorithm that solves problem A. We could then use it, and with some additional (polynomial-time) computation solve B. If we can show such an algorithm we say that B reduces to A, and write $B \leq_m A$ (this should be read informally as "the complexity of B is lower than that of A", or "B is easier").

One of the main cornerstones of complexity theory is the following theorem, due to Stephen Cook (1972), about the 3-SAT problem. We recall that 3-SAT is the following problem: we are given a 3-CNF formula on n variables, that is, a boolean function $F(x_1, x_2, \ldots, x_n) = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where each C_i is of the form $l_i^1 \vee l_i^2 \vee l_i^3$, and each l_i^j is a literal, that is, a variable or the negation of a variable. We need to decide if there is an assignment to the variables that make F true.

Theorem 2. *For any problem $A \in$NP we have $A \leq_m$3-SAT.*

Proof Sketch:

The idea of the proof is that for A there exists a polynomial-time verification algorithm, which takes the input x and a solution y and outputs 1 if the solution is correct. We construct a formula which simulates this algorithm (viewed, e.g., as a circuit). The input variables of the formula are exactly the solution y, so the formula has a satisfying assignment if and only if a solution exists.

□

We say that a problem A is NP-hard if for all $B \in$NP we have $B \leq_m A$. We therefore have that 3-SAT is NP-hard. The point here is that, if there existed a polynomial-time algorithm for an NP-hard problem, then P=NP.

Thanks to the existence of a single concrete NP-hard problem, we are able to show that other specific problems are NP-hard, using reductions, and therefore that these problems are unlikely to be solvable in polynomial time.

Theorem 3. *Consider the following problem (MaxIS): given a graph G and a number B, determin if $\alpha(G) \geq B$. This problem is NP-hard.*

Proof:

We show a reduction from 3-SAT. Let $C_1, C_2, \ldots, C_m$ be the clauses of the input instance. We construct a graph as follows: for each C_i that involves the literals l_i^1, l_i^2, l_i^3 we construct a K_3, with one vertex representing each literal; similarly, for smaller clauses, we construct a K_2 or a K_1. Finally, for any two vertices of the new graph that correspond to two literals x_i and $\neg x_i$, we connect them with an edge. We set $B = m$.

If the original formula had a satisfying assignment, then the new graph has an independent set of size m. To see this, fix a satisfying assignment, and in the graph, for each clique select a vertex that corresponds to a true literal (if more than one exist, pick one arbitrarily). The set of vertices selected is independent, because we selected one vertex from each clique, and we did not select a variable and its negation (since we only selected true literals).

For the other direction, if an independent set of size m exists in the new graph, it must take exactly one vertex from each clique. Furthermore, it cannot take a variable and its negation. Therefore, we can extract a satisfying assignment to the original formula by setting all literals that correspond to vertices of the independent set to true.

□

2.2 Classes Outside/Beyond NP

Let us also briefly touch on a topic we mentioned above: we often think of NP as the limit of interesting problems, because most practical optimization problems are in NP and because solving NP-complete problemes is already pretty hard. However, the fact that a problem belongs in NP is a highly non-trivial property! If you think about, it's not obvious why, given an instance of a decision problem, verifying that some supposed solution is correct should always be easy. This gives rise to the following related classes:

1. CoNP: the class of decision problems for which NO instances have a polynomial certificate. This class is a mirror image of NP, in the sense that the complement of every NP problem is in CoNP (and vice-versa). A typical problem from this class is VALIDITY: we are given a Boolean formula ϕ and are asked if ϕ is *valid*, that is, if it evaluates to True for all possible assignments. In this problem, if the correct asnwer is NO for a particular formula ϕ, there exists a polynomially-checkable certificate to prove it (the assignment that evaluates to False). However, no obvious certificate exists if ϕ is valid. Observe that ϕ is valid if and only if $\neg\phi$ is not satisfiable, so this problem is in a sense a mirror problem for SAT.

2. Consider the following problem, which we will call NETWORK ROBUSTNESS: given a graph $G = (V, E)$, is it true that deleting any set of s edges will produce a graph with dominating set of size at most k (for given integers s, k)? Intuitively, if the answer is Yes that means that the graph represents a robust network: even if at most s of the links fail, the graph still has a small dominating set. There are several things to notice here: first, no obvious certificate exists for either Yes or No instances. For example, if someone gives me a set E' of s edges and claims that these prove that we have a No instance (i.e. that $G - E'$ has a dominating set of size $> k$). How can we check this without computing the dominating set of $G - E'$ (which is NP-hard)? Second, the problem has a $\forall\exists$ structure (for any set of deleted edges, there exists a dominating set). This is different from the $\exists$ structure of NP problems, and the $\forall$ structure of coNP problems.

   Problems defined using this type of quantifier alternations give rise to complexity classes, such as Σ_2^p (for $\exists\forall$), Π_2^p (for $\forall\exists$), Σ_3^p, Π_3^p (for $\exists\forall\exists$ and $\forall\exists\forall$), etc. We call this collection of classes the polynomial hierarchy, as long as the number of quantifiers used to define the problem is finite (does not depend on the input). The PH is thus built of classes which contain harder and harder problems. The first level of the PH corresponds to NP and coNP and then things get gradually harder as we add more quantifier alternations.

3. So what happens if we consider problems where the alternations between quantifiers could depend on the size of the input? This type of problem, which typically corresponds to 2-player games, happens to coincide with the class PSPACE, the class of problems solvable (deterministically) using polynomial space. Observe that all problems of this class are known to be solvable in exponential time ($2^{poly(n)}$), because if a computer only has $poly(n)$ space, it cannot run for more than $2^{poly(n)}$ time without repeating a configuration.

From the above discussion we have

$$\text{P} \subseteq \text{NP} \subseteq \Sigma_2^p \subseteq \ldots \subseteq \text{PH} \subseteq \text{PSPACE} \subseteq \text{EXP}$$

None of these inclusions are *known* to be strict, but we believe that all these classes are proper subclasses of each other. On the other hand, we do know that P$\neq$EXP (so that's something!). Interestingly, even though these classes seem to be getting harder and harder as we go along, outside of P, the best algorithmic upper bound we can give for any of them is just EXP (that is, time $2^{poly(n)}$).

3 FPT Algorithms and Parameterized Complexity

3.1 Reminder: FPT algorithms

Definition 1. *A parameterized problem is a function from $(\chi, k) \in \{0,1\}^* \times \mathbb{N}$ to $\{0,1\}$. In other words, a parameterized problem is a decision problem, where we are supplied together with each instance a positive integer. This integer is called the parameter.*

Typically, in the examples you have seen so far in this course k is the value of the solution, but as we will see this is not mandatory.

Definition 2. *An algorithm solves a parameterized problem in FPT (fixed-parameter tractable) time if it always runs in time at most $f(k)n^c$, where $n = |\chi|$, f is any function and c is a constant that does not depend on n, k.*

Example: recall that there exists a $2^k n^{O(1)}$ algorithm that decides if a graph has vertex cover at most k. We can therefore say that VERTEX COVER admits an FPT algorithm, where the parameter is the size of the optimal solution.

Example: recall that there exists an algorithm that in time $n^k n^{O(1)}$ decides if a graph contains a clique of size at least k. This algorithm is not FPT. Indeed, this algorithm is pretty stupid (it just tries out all $\binom{n}{k}$ sets of vertices of size k and checks if any of them is a clique).

It is not hard to see (by trying out a few values) that the distance between 2^k and n^k is very large. Furthermore, it is not hard to see that for many optimization problems of the form "find k vertices that do something" obtaining an n^k algorithm is easy. The question then becomes, for which such problems can we improve this to 2^k?

Furthermore, the definition of FPT algorithm allows any function $f(k)$. This may lead to a reasonable algorithm if we get $f(k) = 2^k$, a less reasonable one if $f(k) = k^k$ and a terrible one if

$f(k) = 2^{2^k}$. Therefore, given that we have an FPT algorithm for a problem, the question becomes what is the best $f(k)$ that we can hope for?

Today we will try to tackle the first of these questions: given a parameterized problem which can be solved in time n^k, can we improve this to FPT time (say $2^k n^{O(1)}$ or similar)? If not, how can we prove that this is impossible. The research program we will follow uses ideas similar to the research program of NP-completeness: we identify a problem that we believe is hard (because it captures a large class of problems) and then reduce this problem to others to show that they are also hard.

4 The W-hierarchy and W-hard problems

As mentioned, we know that VERTEX COVER admits an FPT algorithm (with respect to the size of the solution), while it is unknown if such an algorithm exists for problems such as CLIQUE and DOMINATING SET. For these problems, the best known algorithms are (more or less) the algorithms that try out all possible solutions of size k.

In the '90s, this motivated the development of a hardness theory, similar in spirit to the theory of NP-completeness. The base of this theory was the following: suppose that there is no FPT algorithm for CLIQUE. What does this imply? Can we then prove that some other problems are hard? This theory was built on the notion of FPT reductions.

Definition 3. *An FPT reduction is an algorithm which, given an instance (χ, k) of a parameterized problem A, produces an instance (χ', k') of a parameterized problem B such that*

- *(χ, k) is a YES instance if and only if (χ', k') is a YES instance.*
- *$k' \leq f(k)$ for some function f that does not depend on χ.*
- *The reduction algorithm runs in time $g(k)n^{O(1)}$.*

When an FPT reduction from A to B exists we write $A \leq_{FPT} B$. It is not hard to see that, in this case, if an FPT algorithm exists for B, such an algorithm must also exist for A. Furthermore, it is not hard to see that the $\leq_{FPT}$ relation is transitive.

4.1 W-hardness

Suppose that we believe that there is no FPT algorithm for CLIQUE. One way then to resolve the question "Does there exist an FPT algorithm for problem A" is to attempt to design an FPT reduction from CLIQUE to A. If we succeed, that means that A also does not have an FPT algorithm (unless we are wrong about CLIQUE!).

An extensive research program of this type was developed in the '90s. Informally, we will say that a parameterized problem is W-hard, if there is an FPT reduction from CLIQUE to that problem[1].

[1]The precise technical term is W[1]-hard, and it follows from the definition of a hierarchy of complexity classes called the W hierarchy. This is beyond the scope of this class.

Let us see a few W-hard problems:

Theorem 4. *The following problems are W-hard:* REGULAR CLIQUE *(given a regular graph, decide if a clique of size k or more exists),* MULTI-COLORED CLIQUE *(given a graph properly colored with k colors, decide if a clique of size exactly k exists).*

Proof:

We show a reduction from an instance (G, k) of CLIQUE to each problem.

In the first case, let Δ be the maximum degree of G. If all vertices have degree Δ we are done, since the graph is already regular! Suppose then, that this is not the case. We construct a graph G' by taking Δ disjoint copies of G. For each vertex $u \in V$ such that $d(u) < \Delta$ we add to the graph $\Delta - d(u)$ vertices and connect them all to all copies of u. It is not hard to see that the new graph is Δ-regular. We set $k' = k$.

A clique in G is also a clique in G'. So let us examine the converse direction. Suppose, without loss of generality that $k' = k > 2$ (otherwise the problem is solvable in polynomial time). If there is a clique of size k' in G' it must not contain any of the "extra" vertices, because these do not belong in any triangles. Therefore, it must only contain vertices from the copies of G. Since these copies are disconnected, it must contain vertices from one copy of G. Thus, it must be a clique in G.

For the second problem, we construct G' as follows: set $V' = V_1 \cup V_2 \cup \ldots \cup V_k$, where all the V_i are copies of V. We include the following edges: for each $(u, v) \in E$, for each $i \neq j \in \{1, \ldots, k\}$ we add an edge between the copy of u in V_i and the copy of v in V_j. We set $k' = k$.

If there is a clique $\{u_1, u_2, \ldots, u_k\}$ in G then we can find a clique of the same size in G' by selecting u_i in V_i. For the converse direction, suppose that there is a clique of size k in G'. It must contain exactly one vertex from each V_i, since these sets contain no edges. Furthermore, it cannot contain two copies of the same vertex of G, since these are also not connected. Therefore, it must contain k different vertices of G, which are all pairwise connected.

□

Theorem 5. MAX K COVER *is W-hard for parameter k. This is the following problem: we are given a graph and two integers k, l. We are asked if there exists a set of at most k vertices that intersect at least l edges.*

Proof:

Reduction from INDEPENDENT SET on regular graphs, which is W-hard from the standard reduction from CLIQUE. The reduction sets k at the same value and $l = k\Delta$, where Δ the degree of the vertices of G. If there exists an independent set of size k, these vertices will each intersect a disjoint set of edges, so $k\Delta$ edges will be intersected in total. If no independent set exists, any set of k vertices will contain an edge. Thus, the total number of edges covered will be at most $k\Delta - 1$.

□

Observe that MAX K COVER is trivially NP-hard by reduction from Vertex Cover (how?). The above reduction shows that in terms of FPT, MAX K COVER is actually harder than VERTEX COVER.

One of the most intriguing open problems in parameterized complexity theory is pointed to by the next theorem:

Theorem 6. DOMINATING SET *is W-hard with respect to the size of the solution.*

Proof:

Consider a reduction from MULTI-COLORED CLIQUE. We are given a graph $G(V_1, \ldots, V_k, E)$. We construct G' as follows: we keep the same vertices, but turn each V_i into a clique. For each i add two new vertices connected to all of V_i. Now, for each $u, v \in G$ such that $(u, v) \notin E$ we do the following: we add a new vertex uv to G' and, if $u \in V_i$ and $v \in V_j$ we connect uv to $V_i \cup V_j \setminus \{u, v\}$. We set $k' = k$.

If the original graph has a clique S, we select the same vertices in G' as a dominating set. All the V_i's are dominated, since they are now cliques, and so are the two extra vertices we added for each V_i. Furthermore, for each uv vertex, it cannot be the case that both $u \in S$ and $v \in S$, since S is a clique and $(u, v) \notin E$. Therefore, uv must be dominated by some vertex of S.

For the converse, observe that because we added two new vertices to each V_i we must select exactly one vertex from each V_i to dominate them. We now claim that if such a dominating set exists it must be a clique: if it contained a non-edge in G, the corresponding vertex uv in G' would not be dominated.

□