Exact Algorithms
Lecture 6: The Exponential Time Hypothesis

8 February 2016 Lecturer: Michael Lampis

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.

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.

□

3 How hard is 3-SAT?

One could view the whole theory of NP-completeness as more or less equivalent ot the following statement: 3-SAT is hard. One believes that P≠NP if and only if one believed that 3-SAT is a problem that does not admit a polynomial time algorithm. But then this raises the question: what

is the fastest algorithm that we can design for 3-SAT? And more generally, how fast can we hope to solve any NP-hard problem?

The same mathematical intuition that has led to P≠NP being an accepted assumption has thus led to the following hypothesis:

Definition 1. *Exponential Time Hypothesis: There exists a constant $c > 1$ such that no algorithm can solve 3-SAT on n variables in time c^n.*

The ETH plays in the theorey of exponential-time algorithms the same role that the P≠NP assumption plays in the theory of polynomial-time algorithms: it gives us a specific starting point from which, through reductions, we can prove that algorithms for various problems cannot exist.

Is this assumption plausible? A first observation is that if the ETH is true then P≠NP. Therefore, if we accept this, we are accepting a stronger assumption (which is bad). However, after considerable effort has been devoted to designing a fast 3-SAT algorithm, the state of the art has been stuck in c^n for a long time. It therefore seems reasonable to accept the ETH as a working hypothesis. At the very least, when we prove that the existence of an algorithm would contradict the ETH, we are showing that designing such an algorithm requires a breakthrough in SAT-solving.

3.1 Using the ETH

Recall the idea of a reduction: we start with an instance of problem A (of size n) and in polynomial time produce an instance of problem B, such that the answers of the two instances agree. If we can do this, then we have reduced A to B, and therefore showed that B is harder. More precisely, we have shown that if A does not have a polynomial-time algorithm, neither does B. But if we accept this, what is the best algorithm for problem A? And if we knew this, what could such a reduction tell us about the best algorithm for problem B?

What we are trying to achieve here is a refined version of NP-completeness theory. With NP-hardness, all we prove is a qualitative fact: A does not have a polynomial-time algorithm (unless P=NP). Now, we would like to prove a quantitative fact: A needs time at least $f(n)$, for some function f (unless the ETH is false). We can now do this, because the ETH gives us such a quantitative lower bound for 3-SAT. But, we have to be more careful with the parameters of our reduction.

Theorem 4. *If the ETH is true, there exists $c > 1$ such that MaxIS on graphs with n vertices cannot be solved in time $c^{n^{1/3}}$.*

Proof:

The proof is the same as for Theorem 3. Let us however see how the running time bound follows. Suppose the original formula had n variables. Therefore, it has at most $O(n^3)$ clauses (otherwise, some clause is repeated and we can delete it). Therefore, the constructed graph has $N = O(n^3)$ vertices.

Suppose now that the theorem was false, that is, the ETH is true but $\forall c > 1$ MaxIS can be solved in time $c^{N^{1/3}}$. Because $N = O(n^3)$ this means that, for all $c > 1$ we can use this hypothetical $MaxIS$ algorithm and the reduction to solve the original 3-SAT instance in time c^n. But this contradicts the ETH.

□

We now have our first concrete lower bound for a real problem! Are we happy with it?

One thing to observe is that the best algorithm we know for MaxIS takes time c^n, for some constant c. This is much higher than $c^{n^{1/3}}$. This raises the question: which is correct? Is there a better algorithm, or should we try to improve our reduction?

Why does the reduction produce a lower bound of $2^{n^{1/3}}$? The answer is that the reduction "blows up" the instance by a cubic factor. More precisely, in the 3-SAT formula we reduce from, we assume (because of the ETH) that the running time we need is exponential in n. In the MaxIS we make, we measure the running time as a function of the number of vertices N. But $N \approx n^3$, so a c^n lower bound becomes $c^{N^{1/3}}$. Can we make the reduction more efficient? If our reduction constructed fewer vertices (perhaps $O(n)$) we would get a stronger lower bound. But how could we not make at least a vertex for each clause?

3.2 Exponential Time Algorithms – in what?

The previous section touches on a topic that is often crucial when considering exponential time algorithms: what is the main variable for which we measure the running time? In the domain of polynomial time algorithm, such question are often overlooked. For example, consider the case of a graph with n vertices. An algorithm that runs in time polynomial in n, also runs in time polynomial in the number of edges m (since $m < n^2$) and vice-versa. Similarly, if we have a polynomial-time algorithm for 3-SAT, it would not matter if the running time is polynomial in the number of variables or clauses, since the two are polynomially related. Nevertheless, the difference between a 2^n and a 2^{n^2} algorithm is huge. We cannot afford to be so careless when speaking about exponential time algorithms.

This raises the question: why did we define the ETH as we did? What about the following version:

Definition 2. *Exponential Time Hypothesis(2): There exists a constant $c > 1$ such that no algorithm can solve 3-SAT on n variables and m clauses in time c^{n+m}.*

In other words, the ETH2 claims that 3-SAT needs time exponential in **the whole input**. Observe that this is a *stronger* assumption. We have ETH2 $\rightarrow$ ETH $\rightarrow$ P$\neq$NP. To see this, notice that an algorithm that runs in time c^n for any c, also runs in c^{n+m} for any c.

Things are now starting to look suspicious. Is the ETH2 more useful in proving lower bounds? Is it plausible as a hypothesis? Let us first answer the first question:

Theorem 5. *If the ETH2 is true, there exists $c > 1$ such that MaxIS on graphs with n vertices cannot be solved in time c^n.*

Proof:

The proof is again the same as for Theorem 3. The difference is that now we observe that the number of vertices N of the new graph is $O(n+m)$. Therefore, if we could solve the new instance in time c^N for any c, with could also solve 3-SAT in time c^{n+m} for any c.

□

This is much nicer, because now this gives a "tight" answer to what is the correct running time needed to solve MaxIS: it is c^n for some constant c (of course, we don't know what the correct constant is! At least, however, we get a bound on the type of function that is correct).

The question then is whether we should believe this, just because it matches the best known algorithmic result for MaxIS. After all, in order to obtain this lower bound we assumed the ETH2, a complexity assumption that is stronger than the ETH.

Actually, the two are equivalent. This was shown in a paper by Impagliazzo, Paturi and Zane (2001).

Theorem 6. *The ETH is true if and only if the ETH2 is true.*

Proof Sketch:

One direction is easy. To see the direction ETH $\rightarrow$ ETH2 we can prove the contrapositive: if ETH2 is false then ETH is false. Suppose then that for all $c > 1$ we have a c^{n+m} algorithm: we will obtain also a c^n algorithm.

The main observation is the following: if $m = O(n)$ we are done: the c^{n+m} algorithm run in time c_2^n for some (slightly larger) c_2 that only depends on c. So the question is how to deal with instances with many clauses.

The proof now follows from a sparsification lemma, which proceed thourgh branching to produce c^n instances, such that all of them have few clauses, and the whole formula was satisfiable if one of the new instances are satisfiable. The (complicated) details are beyond the scope of this class and can be found in the paper of IPZ.

□

Thanks to the Sparsification Lemma of the above theorem we know that in fact the ETH and the ETH2 are equivalent! This allows us to obtain strong lower bounds on the running time of any algorithm solving various classical problems (3-Coloring, Vertex Cover, Dominating Set,...). All of them in fact need time c^n, where n is the number of vertices of the input graph, and this can simply be derived by a more careful reading of their NP-completeness proof.

So can we conclude that everything that is NP-hard, in fact needs time c^n to solve exactly? Not quite. Consider the case of Planar Max IS.

Theorem 7. *If the ETH is true, there exists $c > 1$ such that no algorithm can solve Max IS on planar graphs in time $c^{\sqrt{n}}$.*

Proof:

The idea is to take the graph constructed in Theorem 3 and attempt to draw it on the plane. Since the graph constructed is probably not planar, some edges will cross. We will replace every edge crossing with a gadget that will eventually make the graph planar.

We proceed in two steps. First, we observe that if in any graph we replace an edge (u, v) with a path of length 3 u, u_1, u_2, v, then the size of the maximum independent set increases by exactly 1. Similarly, replacing an edge by a path of length $2k + 1$ increases the maximum independent set by exactly k. We perform the following: as long as G contains an edge (u, v) that crosses k other edges, with $k > 1$, we replace (u, v) with a path of length $2k + 1$ and increase B by k. Observe that, by appropriately placing the new vertices we can make sure that all edges of the path each cross at most one other edge.

We thus arrive at a non-planar graph where each edge crosses at most one other edge. Suppose that there are four vertices in the constructed graph a, a', b, b' such that the edges (a, a') and (b, b') cross. We remove these two edges from the graph and add 10 new vertices: $a_1, \ldots, a_5$ and $b_1, \ldots, b_5$. We connect the a_i vertices into a C_5, and we do the same for the b_i vertices. We add the edges (a_i, b_i) for all i except $i = 3$. Finally, we add the edges (a, a_1), (a, a_5), (b, a_1), (b, b_1), (a', b_1), (a', b_5), (b', b_5), (b', a_5). For every pair of crossing edges we increase B by 4. We now claim that the optimal independent set of the new graph is essentially the optimal independent set of the old (non-planar) graph, with the addition of some internal vertices of the gadgets added for the crossings. (Details are left to the reader).

Finally, for the running time bound, observe that the number of vertices of the produced graphs is $O(n^2)$, where n is the order of the original graph, since for each crossing we construct $O(1)$ new vertices.

□

As we will hopefully see later, this result is the correct one: there does in fact exist a $c^{\sqrt{n}}$ algorithm for planar Max IS! So, NP-hard problems are not all alike when it comes to the time needed to solve them exactly.