CS 787: Advanced Algorithms

NP-Hardness

Instructor: Dieter van Melkebeek

We review the concept of polynomial-time reductions, define various classes of problems including NP-complete, and show that 3-SAT and Vertex Cover are NP-complete. We also briefly discuss tackling NP-hard optimization problems, in particular via approximation algorithms.

1 Polynomial-Time Reductions

We start by defining types of problem, and then move on to defining the polynomial-time reductions.

1.1 Type of problems

Definition 1 (Decision Problem)**.** *A decision problem is a question with a yes-or-no answer.*

Example: The question of whether a Boolean formula has a satisfying assignment of variables, or whether a given natural number is a prime number are examples of decision problems. Another example of a decision problem is the decision version of vertex cover (VC) problem, which is the question of whether there exists a set of at most k vertices in a given graph with all edges incident on at least one vertex from the set. ⊠

Definition 2 (Optimization Problem)**.** *An optimization problem finds an optimal solution among all feasible solutions based on a objective.*

Example: The problem of finding a minimum size set of vertices in a given graph with all edges incident on at least one vertex of set is an example of optimization problem. This is the optimization version of the vertex cover problem (also known as the minimum vertex cover). ⊠

1.2 Reduction

Definition 3 (Polynomial-Time Reducible, $\leq_P$)**.** *Problem A is polynomial-time reducible to problem B ($A \leq_P B$), if an arbitrary instance of problem A can be solved using a polynomial number of computational step, plus a polynomial number of calls to an oracle that solves problem B.*

This formulation has few important consequences. Suppose A is polynomial-time reducible to B, and there exists an algorithm to solve B using polynomial number of computational step. By the reduction, an algorithm to solve A will involve polynomial number of steps, plus polynomial number of calls to the algorithm that solves B that runs in polynomial time. Thus, the algorithm to solve A becomes a polynomial-time algorithm.

The contrapositive of the previous statement will help us in establishing the computation intractability of some problem. The contrapositive states as follows: if A is polynomial-time reducible to B ($A \leq_P B$) and A cannot be solved in polynomial time, then B cannot be solved in polynomial time.

To illustrate this concept of polynomial time reduction, we will look at the decision and optimization versions of the Vertex Cover problem. Given a graph $G = (V, E)$, we say a set of vertices $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one end in S.

Let A be the optimization version of vertex cover problem (or minimum vertex cover), where we need to find the smallest vertex cover.

Let B be the decision version of vertex cover problem, where we need answer the yes-or-no question of whether there exists a vertex cover of size at most k in graph G.

We can trivially show that $B \leq_P A$. We need to find the smallest vertex cover (A). If $k \geq$ the size of smallest cover, then the answer to B is yes; otherwise, the answer is no.

To show $A \leq_P B$, we start with $k = 0$ and increment it until we get a yes answer for the first time. This way we obtain the size of the minimum vertex cover using B as a black-box. To find an actual cover, we take each node out of the graph and run the algorithm for B on the resulting graph to check if a vertex cover of size at most $k-1$ exists; if the algorithm returns true, the vertex is part of the smallest vertex cover. Thus, we add the vertex to the set of nodes representing the vertex cover.

2 Classes of Problems

We now proceed to formalize problems and algorithms. The input to a problem will be encoded as a finite binary string x. The length of the string x is denoted as $|x|$. We define a decision problem T with the *set* of strings on which the answer is "yes." An algorithm A for a decision problem receives an input string x and returns a "yes" or "no." We say that A solves the problem T if it returns the correct answer ("yes" or "no") for all possible input strings.

We say an algorithm runs in polynomial time if there is an polynomial function $p(\cdot)$ so that the algorithm terminates in at most $O(p(|x|))$ step for every input x.

Definition 4 (Class P)**.** *P is the set of all decision problems T for which there exists an algorithm A that runs in polynomial time and solves T.*

Now, we formalize the idea of verifying a solution to a problem efficiently. A "verifying" algorithm for a problem T is different from an algorithm to solve the problem. To verify, the "verifying" algorithm needs the input string x as well as an "certificate" string w that contains evidence that x is a "yes" instance, where $|w| \leq p(|x|)$.

Definition 5 (Class NP)**.** *NP is the set of all decision problems for which there exists an efficient verifier for the "yes" instances of the decision problem.*

$$L \in \mathsf{NP} \quad \Leftrightarrow \quad (\exists\, c > 0)\,(\exists\, V \in \mathsf{P})\;(\forall\, x)\; x \in L \quad \Leftrightarrow \quad (\exists\, w \in \{0,1\}^{|x|^c})\;(x, w) \in V$$

In other words, there exists a polynomial time verifier V of the input string x and "certificate" string w for each problem L in NP.

Example: For Vertex Cover problem, the set of vertices comprising the vertex cover is such a "certificate." $\boxtimes$

Observation: $\mathsf{P} \subseteq \mathsf{NP}$
If we can solve a problem efficiently (every problem in P), it directly follows that it has an efficient verifier. We simply run the efficient algorithm to solve the problem to verify.
Conjecture: $\mathsf{P} \neq \mathsf{NP}$
There are problems in NP for which no efficient algorithm exists. The question of whether $\mathsf{P} = \mathsf{NP}$ is fundamental to the area of algorithms, and it is one of the most famous open problems in computer science. It is generally believed that $\mathsf{P} \neq \mathsf{NP}$. A huge amount of effort has been put into coming up with polynomial time algorithm for hard problems in NP resulting in failure. It is conceivable that this failure is because these problems can not simply be solved in polynomial time.

Definition 6 (NP-hard)**.** *Problem A is* NP*-hard if and only if every problem $B \in \mathsf{NP}$ is polynomial-time reducible to A ($B \leq_P A$).*

Definition 7 (NP-complete)**.** *Problem A is* NP*-complete if and only if A is* NP*-hard and $A \in \mathsf{NP}$.*

Note that by the definition of NP-hard, if A is polynomial-time reducible to B ($A \leq_P B$) and A is NP-hard, then B is also NP-hard. Similarly, by the definition of NP-complete and NP-hard, if A is polynomial-time reducible to B ($A \leq_P B$) and A is NP-complete, then B is also NP-complete.

3 3-SAT

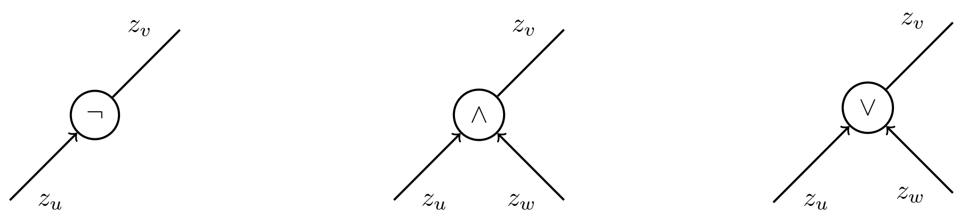
Given: A 3-CNF formula ϕ on variables $x_1, \ldots, x_n$.
Goal: Does there exist a satisfying assignment of variables $x_1, \ldots, x_n$ (i.e., ϕ evaluates to true)?

Theorem 1. *3-SAT is* NP*-complete.*

Proof. 3-SAT is in NP, since given a satisfying assignment we can verify it in polynomial time. We will prove that it is NP-complete by showing Circuit Satisfiability is polynomial-time reducible to 3-SAT assuming Circuit Satisfiability is already known to be NP-complete.

We need to show that we can solve an arbitrary instance of Circuit Satisfiability with a Boolean circuit K by transforming it into an instance of 3-SAT such that satisfiability of K is same as the satisfiability of the transformed 3-SAT instance.

We associate a variable z_v with each node (or gate) v in the circuit K to encode the Boolean value that the circuit will hold at that node. We then define clauses of 3-SAT problem which corresponds to circuit K. The clauses added depend on the type of gates in the circuit:



(a) A NOT gate. (b) An AND gate. (c) An OR gate.

Figure 1: Node (or gate) scenarios in the circuit.

- If node v is a NOT gate (labeled $\neg$) with an edge entering from node u, we need to have $z_v = \bar{z_u}$. Thus, we add clauses $(z_u \vee z_v)$, and $(\bar{z_u} \vee \bar{z_v})$.
- If node v is an AND gate (labeled $\wedge$) with edges entering from nodes u and w, we need to have $z_v = z_u \wedge z_w$. Thus, we add clauses $(\bar{z_v} \vee z_u)$, $(\bar{z_v} \vee z_w)$, and $(z_v \vee \bar{z_u} \vee \bar{z_w})$.
- If node v is an OR gate (labeled $\vee$) with edges entering from nodes u and w, we need to have $z_v = z_u \vee z_w$. Thus, we add clause $(z_v \vee \bar{z_u})$, $(z_v \vee \bar{z_w})$, and $(\bar{z_v} \vee z_u \vee z_w)$.

Thus, this polynomial time construction can be used to solve the Circuit Satisfiability problem of an arbitrary circuit using 3-SAT as a "black-box." □

4 Vertex Cover

Given: A graph $G = (V, E)$, and an integer k.
Goal: Does there exist an vertex cover of size at most k?

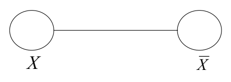
Theorem 2. *Vertex Cover (VC) is* NP*-complete.*

Proof. VC is in NP, since given a vertex cover (S) of size at most k we can verify if it indeed covers all edges (i.e., every edge has one end in S).

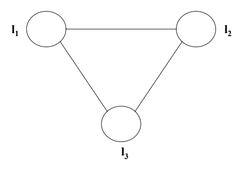
We already know that 3-SAT is NP-complete (and hence NP-hard), so we only need to show that 3-SAT is polynomial-time reducible to VC, i.e., 3-SAT $\leq_P$VC.

We need to transform an arbitrary instance of 3-SAT into an instance of VC such that the yes-or-no answer to VC is the same for 3-SAT. We do this transformation using gadgets for each variable and each clause.

- For each variable x, we create a gadget with two vertices representing x and $\bar{x}$ with an edge in G.
- For each clause $c = \ell_1 \vee \ell_2 \vee \ell_3$, we create a gadget with three vertices representing ℓ_1, ℓ_2, and ℓ_3 with an edge connecting every pair of the three vertices.
- Each vertex in each clause is then connected to the corresponding variable or its negation in variable gadgets.



(a) A variable gadget.



(b) A clause gadget.

Figure 2: Gadgets introduced in 3-SAT.

Notice that $VC(G)$ (the size of the minimum vertex cover) is at least $n + 2m$, where n is the number of vertices, and m is the number of clauses, since to cover the only edge in each variable gadget at least one vertex in the gadget has to be in the vertex cover, and to cover the three edges in each clause gadget at least two vertices has to be in the vertex cover. The following claim will help us solve the given instance of 3-SAT problem using the VC problem of the constructed graph.

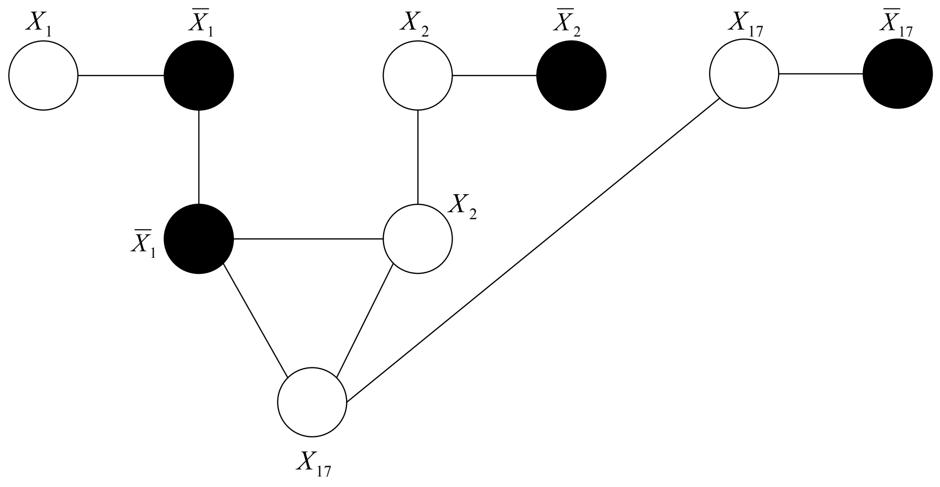


Figure 3: An example for connected gadgets

Claim 1. *Given instance of 3-SAT is satisfiable if and only if the size of minimum vertex cover of the constructed graph is exactly $n + 2m$.*

$$VC(G) = n + 2m \quad \iff \quad \phi \in \textit{3-SAT}$$

Proof. To prove the forward direction (i.e., if $VC(G)$ is $n+2m$ then ϕ is satisfiable), we assign truth value to the vertices of variable gadgets in the VC. For example, if vertex x_i is in VC, we assign x_i true, and if vertex $\bar{x_j}$ is in VC, we assign $\bar{x_j}$ true (or x_j false). Note that there will be exactly one vertex from each variable gadget in the VC, since if none are in VC, it is not a vertex cover, and if both are in the VC, then $VC(G) > n + 2m$. Each clause has two vertices in VC. Thus, they only cover two of the three edges connecting vertices in that clause to variable gadgets. However, since VC must cover all edges, the remaining edge is covered by one vertex from variable gadget, which is assigned true. Thus, every clause has a vertex corresponding to a true valued vertex in variable gadgets. ϕ has a satisfiable assignment.

To prove the reverse direction (i.e., if ϕ is satisfiable then $VC(G)$ is $n + 2m$), we construct VC of size $n + 2m$ was follows:

- In the variable gadgets, we add vertices assigned true in the satisfiable assignment to VC. For example, if x_i is assigned true, we add vertex x_i to VC, and if $\bar{x_j}$ is assigned true, we add vertex $\bar{x_j}$ to VC.
- In each clause, we add two vertices except the first vertex whose corresponding vertex in the variable gadget is assigned true. For example, in a clause $(x_i \vee x_j \vee x_k)$ where x_j is the first variable true, we add clause vertices x_i and x_k to VC.

Since one vertex in each variable gadget is in VC, the only edge in that is covered. Since all the clauses are satisfied, we know there is at least one edge entering the clause gadget from a vertex in VC. Thus, at least one edge entering clause gadget is covered by variable gadget vertices. By excluding the clause vertex whose corresponding variable vertex is assigned true, when including two vertices from each clause in VC, we cover the two remaining edges entering each clause gadget from variable gadgets. By choosing any two vertices in each clause, we also cover the three edges between the vertices in the clause gadget. Thus, vertices selected in VC cover all edges in the graph. Moreover, by choosing one vertex from each variable gadget and two vertices from each clause gadget, VC is of size $n + 2m$. □

Using the above claim, we can solve an arbitrary instance of 3-SAT problem using VC as a "black-box." The graph construction takes only polynomial number of step, and only one call to VC is made in the process. Thus, 3-SAT $\leq_P$VC. □

5 Tackling NP-hard problems

What if you need to solve an instance of an NP-hard problem? There are several recourses.

- Exploiting additional structure of the instance.

  Several NP-complete problems are solvable in polynomial time for interesting subclasses of problems. For example, a simple greedy approach solves the Vertex Cover problem on trees in polynomial time.

  More generally, several NP-complete problems have a natural parameter k (other than the input size) which has a significant impact on the complexity and only takes on small values in practice. For example, in many settings we are only interested in finding a vertex cover of size at most k. Exhaustively trying all subsets of size k yields an algorithm that runs in polynomial time for any fixed k, namely roughly n^k, but this is no good even for relatively small values of k, e.g., $k = 10$. However, there exists an algorithm that runs in time $2^k \cdot n$, which is feasible for $k = 10$. The subarea of parameterized complexity studies which problems allow such algorithms. Time permitting, we will cover some of the results at the end of the course.

- Improved exponential-time algorithms.

  Assuming $\mathsf{P} \neq \mathsf{NP}$ there is no polynomial-time algorithm for an NP-complete problem, but for several NP-complete problems better exponential-time algorithms are known than brute force search. Time permitting, we may discuss some of those algorithms near the end of the course.

- Approximation algorithms.

  Although we may not be able to compute the optimum in polynomial time, we may be able to efficiently find a feasible solution that is within a small factor from optimal. This is the main focus of the rest of the course, to which we give a brief introduction next.

- Heuristics.

  Heuristics are generally used as a last resort in trying to solve NP-hard problems because they do not guarantee an optimal solution or how close to optimal the solution is. They are most often inspired by physical processes. A large group of heuristics can be cast as local search. The following are the high level steps that are generally followed when applying a heuristic: (1) Start from some valid solution. (2) Create a neighborhood structure for the valid solution. (3) Use a criterion to jump to a neighbor. (4) Repeat until (un)happy and return the best solution found.

  We will not cover heuristic in this course.

We first define what we mean by an approximation algorithm for an optimization problem. We use the notation $OPT(x)$ to denote the optimum value of the objective function on input x.

Definition 8 (Approximation Algorithm). *We say that an algorithm is an approximation algorithm with factor $\rho(n)$ for an optimization problem if the algorithm runs in polynomial time and, on input x, produces a feasible solution with objective value at least $\rho(|x|) \cdot OPT(x)$ for maximization problems, and at most $\rho(|x|) \cdot OPT(x)$ for minimization problems, or correctly reports that there is no feasible solution.*

Note that, even though $OPT(x)$ may not be computable in polynomial time, approximation algorithms nevertheless make guarantees relative to $OPT(x)$. The general strategy to do so is to find a bound for $OPT(x)$ (an upper bound for maximixation problems and a lower bound for minimization problems) and relate the quality of the constructed feasible solution to this bound.

The best one could hope for NP-hard optimization problems (assuming $\mathsf{P} \neq \mathsf{NP}$) is that every constant factor $\rho \neq 1$ can be realized.

Definition 9 (PTAS). *A Polynomial Time Approximation Scheme (PTAS for short) is collection of algorithms that realize any $\rho = 1 + \epsilon$ for minimization problems ($\rho = 1 - \epsilon$ for maximization problems) for constant $\epsilon > 0$, and for any fixed ϵ, the running time is $poly(|x|)$.*

In fact, one could hope for even more, namely that the dependency of the running time on $1/\epsilon$ is modest.

Definition 10 (FPTAS). *A Fully Polynomial Time Approximation Scheme (FPTAS for short) is a PTAS with running time $poly(|x|, \frac{1}{\epsilon})$ for any $\epsilon > 0$.*

Although all NP-complete optimization problems are equivalent to each other (up to polynomial reductions) in terms of finding the exact optimum, they behave very differently in terms of efficient approximability. The table below lists the status of the best approximation factor rho that has been achieved for various problems that we will discuss in the sequel of the course, as well as bounds on the best achievable factor under certain complexity theoretic assumptions.

	Asymptotically best ρ known	Lower bound for ρ	Hypothesis
Knapsack Problem	FPTAS	–	–
Euclidean Travelling Salesman Problem	PTAS	no FPTAS	$P \neq NP$
Vertex Cover Problem	2	1.360 $2 - \epsilon$	$P \neq NP$ unique game conjecture
Set Cover Problem	$\ln n$	$\Omega(\log n)$ $(1 - o(1)) \ln n$	$P \neq NP$ $NP \nsubseteq DTime(n^{O(\log \log n)})$
Clique Problem	$\frac{\log^2 n}{n}$	$\frac{1}{n^{1-\epsilon}}$	$P \neq NP$
Travelling Salesman	exponential	no poly time computable factor of number of vertices	$P \neq NP$

In particular, note that Vertex Cover and Clique behave very differently, in spite of the following simple connection.

Claim 2. *S is a VC for graph G iff $\overline{S}$ is a clique in $\overline{G}$.*

Proof. Suppose that G has a vertex cover S. Then for all $u, v \in V$, if $(u, v) \in E$, then $u \in S$ or $v \in S$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin S$ and $v \notin S$, then $(u, v) \in \overline{E}$. In other words, $\overline{S} = V - S$ is a clique in $\overline{G}$.

Conversely, suppose $\overline{G}$ has a clique T, let (u, v) be any edge in E. Then $(u, v) \notin \overline{E}$, which implies that at least one of u or v does not belong to T, since every pair of vertices in T is connected by an edge of $\overline{E}$. Equivalently, at least one of u or v is in $V - T$, which means that edge (u, v) is covered by $V - T$. Since (u, v) was chosen arbitrarily from E, every edge of E is covered by a vertex in $V - T$. Hence, the set $\overline{T} = V - T$ forms a vertex cover for G. □

The explanation for this paradox is as follows: Suppose we have a vertex cover S such that $|S| \leqslant 2OPT_{VC}(G)$. By the above fact, $OPT_{clique}(G) = n - OPT_{VC}(G)$. If we try to get an approximation for the clique problem by complementing S, we end up with $|\overline{S}| = n - |S| \geqslant n - 2 \cdot OPT_{VC}(G) = 2 \cdot OPT_{clique}(G) - n$, which we cannot lower bound by $\rho(n) \cdot OPT_{clique}(G)$ for any $\rho(n)$ substantially smaller than n.

In the rest of the course we will use various techniques to develop approximation algorithms for NP-hard optimization problems: greed, divide-and-conquer, dynamic programming, linear programming, and semidefinite programming.