CSOR W4231.002 Lecture 12: Apr 28, 2014 Spring 2014

Eleni Drinea

So far we have been looking for and identified efficient algorithms for a variety of problems. We defined efficient algorithms as those with worst-case polynomial running time. In this lecture, we focus on problems for which no **efficient** algorithms are known.[1] To argue about the relative hardness (difficulty) of such problems, we will use **reductions**.

1 Reductions as a tool to design efficient algorithms

Implicitly, we have already used reductions before. Specifically, in the last lecture, we considered the problem of finding a perfect matching in a bipartite graph. Let's denote this problem by BPM. An input instance to BPM is a graph G. We may think of the input instance as a binary string x encoding the graph G, with length $|x|$ bits. In this case, we are looking for an algorithm for BPM which, on input a binary string x, answers "yes" if x encodes a bipartite graph that has a perfect matching and "no" otherwise.[2] We used the following approach to solve BPM.

1. We transformed the input instance x of BPM into an instance y of Max Flow. That is, given any bipartite graph G (instance for BPM), we constructed a flow network G' (instance for Max Flow).
2. We showed that the original bipartite graph has a matching of size k **iff** the derived flow network G' has a flow of value k.
3. We ran the Ford-Fulkerson algorithm on G'; we output "yes" as the answer to BPM on input x if and only if the answer to "Is $|f| = n$ on input y?" on input y is "yes".

Figure 1(a) illustrates the procedure we used to solve BPM. Note that the transformation from G to G' only required polynomial time in $|x|$.

Definition 1 *A reduction is a transformation that for every input x of X produces an equivalent input $y = R(x)$ of Y.*

By equivalent we mean that the answer to $y = R(x)$ considered as an input for Y is a correct "yes/no" answer to x, considered as an input for X.

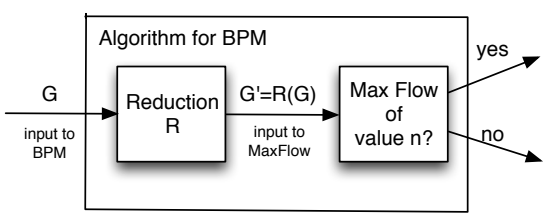
We will require that the reduction is completed in a polynomial in $|x|$ number of computational steps: if the transformation requires superpolynomial time to complete, then the overall algorithm for solving Y is no longer efficient. In general, it is easy to reach absurd conclusions by allowing such reductions!

So to illustrate the definition above better, suppose we had a **black box** for solving problem Y, as in Figure 1. If arbitrary instances of problem X can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves Y, we say that X reduces polynomially to Y and denote this by

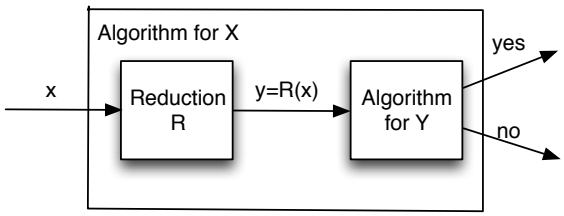
$$X \leq_P Y.$$

[1] However these problems are solvable in exponential time. Note that there exist problems that are much harder, e.g., problems that are not solvable by any algorithm regardless of any computational bounds we impose (*undecidable* problems).

[2] It is important to use a "reasonable" encoding for x: by reasonable we mean any encoding that uses logarithmic number of bits to encode a number, as opposed to, for example, the unary encoding. The point is that different "reasonable" encodings are related polynomially (e.g., using $\log_2 k$ vs $\log_b k$ bits to encode integer k, for integer constant $b > 2$).



(a) Reduction from BPM to Max Flow



(b) Reduction from X to Y

Figure 1: Reductions between problems.

Remark 1 *Note that to solve BPM we required exactly one call to the black box for Max Flow. In all our reductions today, we will be using one black box call.*[3]

The following fact is straightforward.

Fact 1 *Suppose $X \leq_P Y$. If Y is solvable in polynomial time, then X is solvable in polynomial time.*

Since we defined efficient algorithms as those with worst-case polynomial time performance, reductions between problems provides us with a powerful technique for designing efficient algorithms. For example, by reducing Bipartite Matching to Max Flow, Max Flow to *s-t* connectivity and computing Fibonacci numbers to matrix exponentiation, we obtained efficient algorithms for the second problem in each of these pairs.

2 Reductions as a tool to argue about the relative hardness of problems

Note that if $X \leq_P Y$, then Y is at least as hard (difficult) as X: given an algorithm to solve Y, we can solve X. So the contrapositive of Fact 1 provides us a way to conclude that a problem does not have an efficient algorithm if another problem that is known not to have an efficient algorithm reduces to it.

Fact 2 *Suppose $X \leq_P Y$. If X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.*

For the problems we will be discussing today, we have **no proof** that they are not solvable is polynomial time. So we will be using Fact 2 to establish relative levels of difficulty among these problems.

We will now consider two graph-theoretic problems for which no efficient algorithms are known. First we define the notions of independent set and vertex cover in a graph.

Definition 2 (Independent Set) *An independent set in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that there is no edge between any pair of nodes in the set. That is, for all $u, v \in S$, $(u, v) \notin E$.*

Definition 3 (Vertex Cover) *A vertex cover in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that every edge $e \in E$ has at least one endpoint in S.*

See Figure 2 for examples.

It is easy to find a small independent set or a large vertex cover in a graph (*why?*). However finding the largest independent set or the smallest vertex cover is not so easy: the tricky part is which vertices to include since inclusion of some vertices might exclude others that would allow for even larger (smaller respectively) sets. Formally, the problems we will consider are the following.

[3]This type of reduction is called a Karp reduction as opposed to a Cook reduction, which may use a polynomial number of calls to the black box.

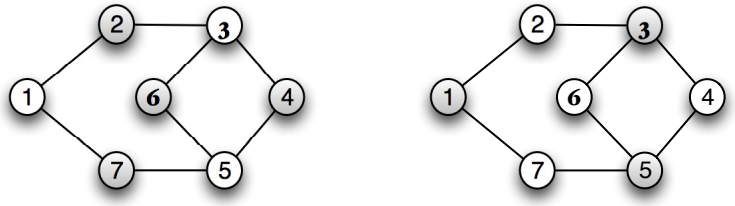


Figure 2: Nodes $\{2, 4, 6, 7\}$ form an independent set of maximum size, while nodes $\{1, 3, 5\}$ form a vertex cover of minimum size.

Definition 4 (Maximum Independent Set problem) *Given G, find an independent set of maximum size.*

Definition 5 (Minimum Vertex Cover problem) *Given G, find a vertex cover of minimum size.*

A brute force approach for these problems would require exponential time: there are 2^n candidate subsets since every vertex may or may not belong to such a subset.

2.1 Decision versions of optimization problems

We will discuss an alternative formulation for optimization problems that will be useful for treating them from a complexity perspective.[4] An optimization problem may be transformed into a roughly equivalent problem with a "yes/no" answer, called the **decision version** of the optimization problem, by

1. supplying a **target** value for the quantity to be optimized; and
2. asking the question whether this value can be attained.

We will denote the decision version of a problem X as $X(D)$ to make it clear that we refer to the decision version. For example,

- Max Flow (D): Given a flow network G and an integer k (the **target** flow), does G have a flow of value at least k?
- Bipartite Matching (D): Given a bipartite graph G and an integer k, does G have a matching of size at least k?
- $IS(D)$: Given a graph G and an integer k, does G have an independent set of size at least k?
- $VC(D)$: Given a graph G and an integer k, does G have a vertex cover of size at most k?

So in a maximization problem the target value is a **lower** bound, while in a minimization problem it is an **upper** bound.

When using reductions to relate problems in terms of their hardness, it is easier to use their decision versions. However, if we are to use the two versions interchangeably, we should establish that they are roughly "equally hard". We will now establish that they are, by using a concrete example as a guide, the Maximum Independent Set problem and its decision version $IS(D)$.

[4]Optimization problems are problems that minimize or maximize an objective. Lately we encountered several such problems: shortest paths, minimum spanning trees, max flow, max matching in bipartite graphs, and now, max independent set and min vertex cover.

1. Suppose we have an algorithm that solves the Maximum Independent Set problem. Then we can solve its decision version IS (D), for every k, as follows. Compute the size m of the maximum independent set and simply answer "yes" if $k \leq m$ and "no" otherwise.

2. Suppose we have an algorithm that solves the decision version $IS(D)$, that is, given an instance $G = (V, E)$ and any integer k, it answers "yes" if G has an independent set of size at least k and "no" otherwise.

   Then we can find the size m of the maximum independent set using binary search. So we only need $\log n$ calls to $IS(D)$ to find m.

Note that since both G and k are inputs to every call to IS(D), and k is an integer hence described with $\log_2 k$ bits, the above algorithm for finding the size of the maximum independent set given a black box for $IS(D)$ indeed runs in time polynomial in the size of the input.

This rough equivalence between optimization problems and decision problems holds for all the problems we will discuss. So from now on, we will refer to decision problems.

Remark 2 *In this context, in the definition of reduction, $y = R(x)$ is equivalent to x means that the answer to $Y(D)$ on input y is "yes" if and only if the answer to $X(D)$ on input x is "yes".*

2.2 A first reduction: IS (D) to VC (D)

We are now ready to establish:

Claim 1 *$IS \leq_P VC$ and $VC \leq_P IS$.*

We will prove the claim by observing a simple one to one correspondence between independent sets and vertex covers in a graph, already hinted to by Figure 2.

Fact 3 *Let $G = (V, E)$ be a graph. Then S is an independent set of G if and only if $V - S$ is a vertex cover of G.*

Proof. We first show the forward direction. If S is an independent set of G, then for all $u, v \in S, (u, v) \notin E$. Now consider the set $V - S$ and edge $(u, v) \in E$. Either $u, v \in V - S$, hence (u, v) is covered by $V - S$. Or, one of u, v is in S and the other is in $V - S$; again, (u, v) is covered by the endpoint that is in $V - S$. Hence $V - S$ is a vertex cover of G.

For the reverse direction, if S is a vertex cover of G, then (by definition) for all $(u, v) \in E$, at least one of u, v is in S. Now consider the set $V - S$ and any two nodes $u, v \in V - S$. If there is an edge $(u, v) \in E$, then this edge is not covered by S. Hence S is not a vertex cover of G, contradiction. Thus no edge exists between any pair of nodes in $V - S$ and $V - S$ is an independent set of G. □

It is now very easy to prove Claim 1.

Given an instance $x = (G, k)$ for $IS(D)$, transform it to an instance $y = (G, n - k)$ for $VC(D)$. This completes the polynomial-time transformation. Also, the two instances are equivalent since, by Fact 3, $IS(D)$ answers "yes" on x if and only if $VC(D)$ answers "yes" on y. Hence $IS(D) \leq_P VC(D)$.

For the second part of the claim, given an instance $x = (G, k)$ for $VC(D)$, transform it into an instance $y = (G, n - k)$ for $IS(D)$. This completes the transformation. Equivalence of the two instances follows again from Fact 3. Thus $VC(D) \leq_P IS(D)$.

3 Boolean expressions and satisfiability

We will now introduce a class of problems formulated in Boolean notation and expressing Boolean logic. We will start by describing the **syntax** of Boolean expressions.

- A **boolean variable** x is a variable that takes values from $\{0,1\}$ (equivalently, $\{F,T\}$, standing for False, True).
- Suppose you are given a set of n boolean variables $\{x_1, x_2, \ldots, x_n\}$.
- Boolean variables may be connected by **boolean connectives** to form **boolean expressions**, such as logical AND $\wedge$, logical OR $\vee$ and logical NOT $\neg$ (similarly to real variables connected in algebra by $+, -, \times$ to form arithmetic expressions).
- A **Boolean expression** may be any of the following
  1. A Boolean variable, e.g., x_i.
  2. The negation of a Boolean expression, denoted by $\neg\phi_1$ or $\overline{\phi_1}$.
  3. The disjunction (logical OR) of two Boolean expressions in parentheses $(\phi_1 \vee \phi_2)$.
  4. The conjunction (logical AND) of two Boolean expressions in parentheses $(\phi_1 \wedge \phi_2)$.

It turns out that every Boolean expression may be rewritten into an equivalent one of a convenient, special form called CNF (conjunctive normal form).

Definition 6 *A Boolean formula ϕ is in CNF if it consists of conjunctions of clauses each of which is a disjunction of literals, where a literal ℓ_i is a variable or its negation.*

In symbols, a formula ϕ with m clauses is in CNF if

$$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

and each clause C_i is the conjunction (logical AND) of a number of literals

$$\ell_1 \vee \ell_2 \vee \ldots \vee \ell_k.$$

An example formula in CNF with $n = 3$ and $m = 2$ is:

$$\phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

From now on, we will be concerned with formulas in CNF.

So far we've discussed the syntax of a boolean formula. The semantics of a boolean formula are similar to that of an arithmetic expression: assigning truth values from $\{0,1\}$ to the variables in the formula will cause the boolean formula to receive a truth value from $\{0,1\}$.

Definition 7 *Let $X = \{x_1, \ldots, x_n\}$. A* **truth assignment** *for X is an assignment of truth values from $\{0,1\}$ to each x_i (so it is a function $v : X \to \{0,1\}$). Is it implied that $\overline{x_i}$ obtains value opposite from x_i.*

A truth assignment **satisfies** a clause if it causes the clause to evaluate to 1. For example: $x_1 = x_2 = 1, x_3 = 0$ satisfies both clauses in the example ϕ above.

A truth assignment **satisfies** a formula if it satisfies every clause in the formula (so every clause evaluates to 1). For example: $x_1 = x_2 = 1, x_3 = 0$ satisfies ϕ, while $x_1 = 1, x_2 = x_3 = 0$ does not satisfy ϕ.

Finally, a formula ϕ is **satisfiable** if it has a satisfying truth assignment.

We may now formulate the following fundamental problem.

Definition 8 (SAT) *Given a formula ϕ in CNF with n variables and m clauses, is ϕ satisfiable?*

A convenient (and not easier, as we shall see in Section 5.2) variant of SAT requires that every clause consists of exactly three literals.

Definition 9 (3SAT) *Given a formula ϕ in CNF with n variables and m clauses such that each clause has exactly 3 literals, is there a truth assignment that satisfies ϕ?*

Note that a brute force algorithm for SAT, 3SAT involves searching over 2^n possible $0-1$ settings of the variables in ϕ.

3.1 3SAT $\leq_P$ IS

We are now ready for our first non-trivial reduction.

Claim 2 *3SAT $\leq_P$ IS (D).*

Proof. We want to reduce 3SAT to $IS(D)$. Therefore, given an arbitrary instance formula ϕ of 3SAT, we need to transform it into a graph G and an integer k in polynomial time, so that the instance (G, k) is a "yes" instance of $IS(D)$ if and only if ϕ is a "yes" instance of 3SAT (i.e., ϕ is satisfiable).

At the heart of such reductions lies understanding some small instance of the problem we wish to reduce to that makes this problem difficult. For $IS(D)$, such an instance is graphs with triangles: we cannot tell which vertex from the triangle we should add to our independent set.

Also, when reducing from 3SAT, we often use "gadgets". Gadgets are constructions that ensure:

1. **Consistency of truth values in a truth assignment:** we need to make sure that once x_i is assigned a truth value, we will henceforth consistently use it under this truth value.
2. **Clause constraints:** since ϕ is in CNF, we must provide a way to satisfy **every** clause for ϕ to be satisfiable. Equivalently, we must exhibit at least one literal that is set to 1 in every clause.

In effect, the consistency and clause constraint gadgets will allow us to derive a valid and satisfying truth assignment for ϕ in the case where the transformed instance is a "yes" instance of our problem. Specifically for $IS(D)$, our gadgets are as follows.

- Given our understanding for the difficulty that triangles cause for $IS(D)$, our clause constraint gadget is straightforward: for every clause, introduce a triangle where every node corresponds to a literal in the clause. Therefore, our graph G consists of m isolated triangles. Trivially, the max independent set in the graph constructed this way has size m: pick one vertex from every triangle. So we will set $k = m$.

  Our plan is to derive a truth assignment from our independent set as follows: when we add a vertex from a triangle to the independent set, we will be setting to 1 the corresponding literal in the corresponding clause. This takes care of clause constraints.

- How about consistency of the truth assignment? Suppose we picked x_1 from the first triangle: nothing prevents us from picking $\overline{x_1}$ from the second triangle. But then we would be setting x_1 to both 1 and 0, which is obviously not a valid truth assignment!

  Note that the only reason for assigning opposite values to some variable x_i is that we added to the independent set a node corresponding to x_i in some triangle and a node corresponding to $\overline{x_i}$ in some other triangle. So we need to guarantee that we cannot add a node corresponding to x_i **and** a node corresponding to $\overline{x_i}$ to our independent set.

  The way to enforce this is to add edges between all occurrences of x_i and $\overline{x_i}$, for every i, in G.

This completes the construction of $(G, k = m)$, our transformed instance for $IS(D)$ (see Figure 3 for an example). Obviously the construction requires time polynomial in the size of ϕ.

We now need to show that ϕ is satisfiable if and only if $(G, k = m)$ is a "yes" instance of $IS(D)$.

- First suppose that G has an independent set of size m. Then every triangle contributes one node to this independent set. So set the literal corresponding to that node to 1. (Any variables left unset by this assignment may be set to 0 or 1 arbitrarily). This completes our truth assignment.

  We need to show that this truth assignment is valid and satisfies ϕ. First it is a valid truth assignment since no two opposite literals appear in our independent set (such literals are connected by an edge in G). Also, since **every** triangle contributes a node to the independent set, our truth assignment ensures that every clause has a true literal, hence every clause is satisfied. So ϕ is satisfied.

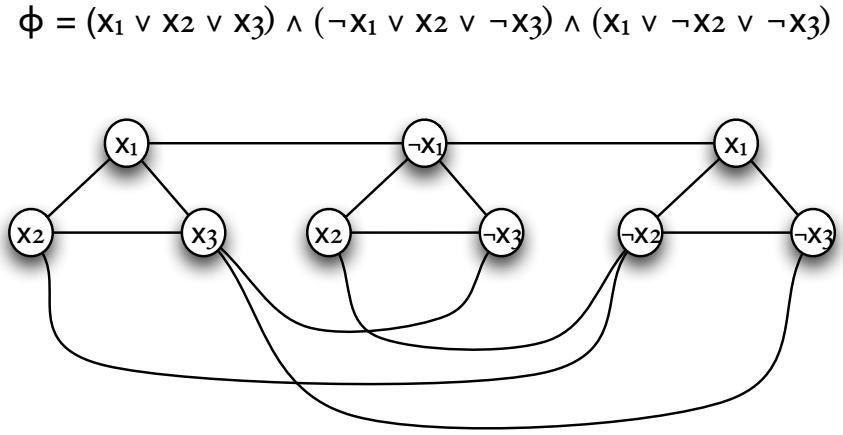


Figure 3: A formula ϕ with $n = m = 3$ and the corresponding graph G with $m = 3$ triangles and $k = 3$.

- Conversely, suppose there is a satisfying truth assignment for ϕ. Then there is (at least) one true literal in every clause. From every triangle, we add to our independent set S a node corresponding to such a literal. Hence S has size m. We claim that S thus constructed is indeed an independent set: since all nodes in S belong to different triangles, if there was an edge between any two nodes in S, these nodes would correspond to opposite literals. But this is not possible since all literals included in S evaluate to 1.

□

We now summarize the general approach for reducing 3SAT to a decision problem $X(D)$.

- Start by toying with small instances of $X(D)$ until one with an interesting behavior is isolated (here, the triangle).
- Given an arbitrary 3SAT formula ϕ, construct (using clause constraints and consistency gadgets) an input instance $x = R(\phi)$ for $X(D)$ in polynomial time.
- Show that x and ϕ are equivalent, that is, ϕ is satisfiable if and only if $X(D)$ on input x answers "yes".

A common mistake when attempting a reduction is to use exponential size constructions: subsets and permutations may lead to such issues.

Finally, note that SAT and 3SAT are fundamental combinatorial search problems: we need to make n independent decisions so as to simultaneously satisfy a set of constraints. So if it is not clear which problem to reduce from, 3SAT is a good choice.

4 The classes $\mathcal{P}$ and $\mathcal{NP}$

So far we've shown that $VC(D)$ and $IS(D)$ reduce to each other and that 3SAT reduces to $IS(D)$. We can further show that $3SAT \leq_P VC(D)$ by using the following fact.

Fact 4 (Transitivity of reductions) *If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.*

However we still haven't characterized how hard these problems really are. To this end, we say that $x \in X(D)$ if x is a "yes" instance of $X(D)$. Then an algorithm A **solves** (or decides) $X(D)$ if for

all x, $A(x)$ ="yes" if and only if $x \in X(D)$. Further, A has a polynomial running time, if there is a polynomial $T(\cdot)$ such that for all input strings x of length $|x|$, the worst-case running time of A on input x is $O(T(|x|))$.

Definition 10 *We define $\mathcal{P}$ to be the set of all problems X(D) that can be solved by polynomial-time algorithms.*

As mentioned already, for problems such as 3SAT and $IS(D)$, although we have no proof that they are not solvable in polynomial time, no polynomial time algorithm has been found despite significant effort (and further evidence of their hardness will be provided shortly when we show that they are $\mathcal{NP}$-complete). So we don't believe that they are in $\mathcal{P}$.

However, if we were given a solution for such a problem, we could **check** if it is correct **quickly**. For example, given an instance $x = (G, k)$ for $IS(D)$ and a candidate solution S, we can verify quickly that S is a correct solution by checking $|S|$ and that there is no edge between any pair of nodes in S. In this case, we could think of S as a **succinct certificate** that $x \in IS(D)$. Note that 3SAT and $VC(D)$ also possess similar succinct (short) certificates (*why?*)) that can be used to verify quickly whether the input instance is a "yes" instance or not.

Definition 11 *An efficient certifier (or verification algorithm) B for a problem $X(D)$ is a poly-time algorithm that takes two input arguments, the instance x and the certificate t (both encoded as binary strings). Further, there is a polynomial function p so that for every string x, we have $x \in X(D)$ if and only if there is a string t such that $|t| \leq p(|x|)$ and $B(x, t) =$ "yes".*

Note that existence of the certifier B does not provide us with any efficient way to **solve** $X(D)$ (*why?*).

Definition 12 *We define $\mathcal{NP}$ to be the set of decisions problems that have an efficient certifier.*

Fact 5 $\mathcal{P} \subseteq \mathcal{NP}$

Proof. Let $X(D)$ be a problem in $\mathcal{P}$. Then there is an efficient algorithm $A(x)$ that solves $X(D)$, that is, $A(x)$ ="yes" if and only if $x \in X(D)$. We want to show that $X(D) \in \mathcal{NP}$, so we want to exhibit an efficient certifier B that takes two inputs s and t and answers "yes" if and only if $x \in X(D)$. The algorithm B that on inputs x, t, simply discards t and simulates $A(x)$ is such an efficient certifier. □

However all efforts to answer **proper** inclusion in Fact 5 have failed so far, so the following question –among the most famous questions asked by theoretical computer science– is still open[5]

$$\mathcal{P} = \mathcal{NP}?$$

To understand why the class $\mathcal{NP}$ might include more problems than $\mathcal{P}$, research has focused on identifying the hardest problems in $\mathcal{NP}$, hence those that are the least probable of being in $\mathcal{P}$. Once again, the notion of reduction is useful.

Definition 13 ($\mathcal{NP}$-complete problems:) *A problem $X(D)$ is $\mathcal{NP}$-complete if $X(D) \in \mathcal{NP}$ and for all $Y \in \mathcal{NP}$, $Y \leq_P X$.*

Fact 6 *Suppose X is $\mathcal{NP}$-complete. Then X is solvable in polynomial time (i.e., $X \in \mathcal{P}$) if and only if $\mathcal{P} = \mathcal{NP}$.*

[5]The general belief is no: it would be too surprising if the tasks of **checking** a solution and **finding** a solution had the same complexity for every problem, and in particular, for the likes of SAT, $IS(D)$, etc.

Showing that a problem is $\mathcal{NP}$-complete establishes that it is among the least likely to be in $\mathcal{P}$: it is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$. Therefore, from an algorithmic perspective, $\mathcal{NP}$-completeness complements algorithm design techniques: if a problem is $\mathcal{NP}$-complete, we should stop looking for efficient algorithms for the problem and instead work on developing approximation algorithms (algorithms that return a solution within certain theoretical guarantees from the optimal solution), or exponential algorithms practical for small instances, or randomized algorithms, or work on interesting special cases, or study the average performance of the algorithm, or examine heuristics (algorithms that work well in practice, yet provide no theoretical guarantees regarding how close the solution they find is to the optimal onet).

5 $\mathcal{NP}$-complete problems

How do we show that a problem is $\mathcal{NP}$-complete?

First, suppose we had an $\mathcal{NP}$-complete problem X. Then we can use transitivity of reductions, established in Fact 4. To show that another problem Y is $\mathcal{NP}$-complete, we only need show that $Y \in \mathcal{NP}$ and $X \leq_P Y$: since for all $A \in \mathcal{NP}$, $A \leq_P X$, by Fact 4, $A \leq_P Y$, so Y is $\mathcal{NP}$-complete.

So **if** we had a first $\mathcal{NP}$-complete problem, discovering a new problem in this class would require an easier kind of reduction since we would only have to reduce one NP-complete problem to this new problem (instead of **every** problem in $\mathcal{NP}$).

5.1 Circuit-SAT is $\mathcal{NP}$-complete

The first problem shown to be $\mathcal{NP}$-complete was Circuit SAT.

Theorem 1 (Cook-Levin) *Circuit SAT is $\mathcal{NP}$-complete.*

The proof of this theorem is beyond the scope of our class (see your textbook if you are interested). However we will define the notion of a combinatorial boolean circuit because we will need it later.

A physical circuit consists of gates that perform logical AND, OR and NOT. We will model such a circuit by a boolean combinatorial circuit which is defined as a labelled DAG. The nodes in the DAG are:

- **Source nodes**: these are either hardwired to 0 or 1, or correspond to the inputs of the circuit and are labelled with some variable.
- **Intermediate** nodes: these correspond to the gates of the circuit and are labelled with $\wedge$ (AND), $\vee$ (OR) or $\neg$ (NOT). If labelled with $\wedge$, $\vee$, they have two incoming and one outgoing edge. If labelled with $\neg$, they have one incoming and one outgoing edge.
- **Sink node**: it corresponds to the output of the circuit, that is, the result computed by the circuit and has no outgoing edges.

See Figure 4 for an example of a boolean circuit with three inputs.

A circuit naturally computes a value as follows. Edges represent wires that carry the value of their tail node. Intermediate nodes perform their label operation on their incoming edges and pass the result along their outgoing edge. The value of the circuit C is the value computed at the output node.

Definition 14 (Circuit-SAT) *Given a boolean combinatorial circuit C, is there an assignment of truth values to its inputs that causes the output to evaluate to 1?*

It is easy to see that Circuit-SAT is in $\mathcal{NP}$. To show that every problem in $\mathcal{NP}$ reduces to Circuit-SAT, recall that every such problem possesses an efficient verification algorithm. The two main ideas in the reduction are:

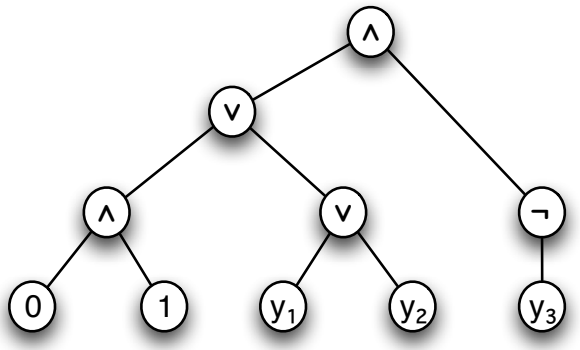


Figure 4: A circuit C with 2 hardwired source nodes, 3 inputs y_1, y_2, y_3 and 5 logical gates.

- Any algorithm that takes a fixed number n of input bits and produces a "yes/no" answer may be represented by a circuit whose output is 1 exactly on those inputs for which the algorithm said "yes". (After all, algorithms are implemented by the computer hardware which, at the very basic level, performs logical AND, OR and NOTs.)
- Moreover, if the algorithm takes a polynomial in the size of the input number of steps to terminate, then the circuit will have polynomial size. (Recall that the certifier is efficient and the certificates are "short".)

5.2 3SAT is $\mathcal{NP}$-complete

Lemma 1 *Circuit-SAT* $\leq_P$ *3SAT*

Given our observation in the beginning of Section 5.1, Lemma 1 establishes that 3SAT, $IS(D)$, $VC(D)$ are all $\mathcal{NP}$-complete. The proof of the lemma also shows that SAT is $\mathcal{NP}$-complete.
Proof. Intuitively, this reduction should not be too difficult: formulas and circuits are just different ways of representing boolean functions. It will be helpful to define two more boolean connectives first.

1. $(\phi_1 \Rightarrow \phi_2)$ is a shorthand for $(\overline{\phi_1} \vee \phi_2)$.

   Equivalently, if $\phi_1 = 1$, the only way for this clause to evaluate to 1 is if $\phi_2 = 1$ as well.

2. $(\phi_1 \Leftrightarrow \phi_2)$ is a shorthand for $((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))$, which may be expanded to $(\overline{\phi_1} \vee \phi_2) \wedge (\phi_1 \vee \overline{\phi_2})$.

   Equivalently, the only way for this clause to be evaluate to 1 is if $\phi_1 = \phi_2$. Otherwise, the clause is false.

We will first reduce Circuit-SAT to SAT. That is, given a circuit C we will construct in polynomial time an equivalent SAT formula ϕ, so that there will be a satisfying truth assignment for ϕ if and only if there is a satisfying truth assignment for C. Then we will transform ϕ into an equivalent instance ϕ' of 3SAT. This will complete the reduction from Circuit-SAT to 3SAT.

Circuit-SAT $\leq_P$ SAT

Consider an arbitrary instance of Circuit-SAT, that is, a circuit C: it consists of source nodes, gate nodes that perform a boolean operation and an output node. Our transformation will be very simple: for every node v in C, we will introduce a variable x_v to ϕ. We will also introduce clauses, which, in

effect, will evaluate to 1 if the values of the variables in the formula correspond to the correct outputs of the nodes in C. Therefore, any satisfying truth assignment for C will immediately imply that ϕ is satisfiable while, if ϕ is satisfiable, it is because it assigned truth values to the inputs of C that resulted in C computing an output with value 1.

In detail, formula ϕ will consist of the conjunction of the following clauses.

1. If v is a source node corresponding to an input (variable) of the circuit C, we do not add any clauses for x_v.

2. If v is a source node that is hardwired to value 0, we want x_v to be 0 in any truth assignment that satisfies ϕ. Hence we introduce clause $(\overline{x_v})$.

3. If v is a source node that is hardwired to value 1, we want x_v to be 1 in any truth assignment that satisfies ϕ. Hence we introduce clause (x_v).

4. If v is the output node, since we want ϕ to evaluate to 1 if and only if C evaluates to 1, we introduce (x_v).

5. If v is a node labelled by NOT and its input edge is from node u, we want x_v to equal $\overline{x_u}$ in ϕ (so that x_v corresponds to the output of node v in C). To guarantee this, we introduce a clause that evaluates to 1 if and only if $x_v = \overline{x_u}$. As discussed above, this clause is
$$(x_v \Leftrightarrow \overline{x_u}).$$

6. If v is a node labelled by OR and its input edges are from nodes u and w, we want x_v to have the same value as $x_u \vee x_w$ in our formula (so that x_v corresponds to the output of node v in C). To guarantee this, we introduce a clause that evaluates to 1 if and only if $x_v = x_u \vee x_w$. Again, this clause is given by
$$((x_u \vee x_w) \Leftrightarrow x_v)$$

7. If v is a node labelled by AND and its input edges are from nodes u and w, we want x_v to have the same value as $x_u \wedge x_w$ in our formula (so that x_v corresponds to the output of node v in C). To guarantee this, we introduce a clause that evaluates to 1 if and only if $x_v = x_u \vee x_w$. Again, this clause is given by
$$((x_u \wedge x_w) \Leftrightarrow x_v).$$

This completes our construction of the clauses of ϕ. For example, for the circuit in Figure 4, we construct the following formula.

$$\phi = (\neg x_1) \wedge (x_2) \wedge (x_6 \Leftrightarrow (x_1 \wedge x_2)) \wedge (x_7 \Leftrightarrow (x_3 \vee x_4)) \wedge (x_8 \Leftrightarrow \neg x_5) \wedge (x_9 \Leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \Leftrightarrow (x_9 \wedge x_8)) \wedge (x_{10})$$

It is easy to observe that the construction is polynomial in the size of the input circuit: we only have as many variables in ϕ as nodes in the circuit and a polynomial number of clauses (at most 3 per node v in C –*why?*). Moreover, every clause consists of at most three literals, once we rewrite the above clauses in CNF (*exercise*).

We claim that our SAT instance is equivalent to the circuit C of the Circuit-SAT instance, that is, there is a truth assignment over the variables in ϕ that causes ϕ to evaluate to 1 if and only if there is a truth assignment over the inputs of C that causes C to evaluate to 1.

- Suppose C is satisfiable. The satisfying truth assignment can be propagated to create truth values at all nodes of C. This set of values clearly satisfies our formula.

- Suppose ϕ has a satisfying truth assignment. Then the values of the variables of ϕ that correspond to inputs in C satisfy C: the SAT clauses just ensure that the values assigned to all nodes of C are the same as what C computes on these nodes. Since $\phi = 1$, $C = 1$ too.

We will now transform this SAT instance to a 3SAT instance: once the clauses with the boolean connective $\Leftrightarrow$ have been expanded into proper CNF clauses, ϕ only contains clauses with one, two or three literals. To generate an equivalent instance of 3SAT given ϕ, we simply duplicate literals in any clause with fewer than 3 literals: the new clause still evaluates to 1 if and only if at least one of its original literals evaluates to 1. □

5.3 The Traveling Salesman Problem

A **tour** in a graph is an order in simple cycle that visits every vertex in the graph once. Consider the following problem.

Definition 15 (TSP) *We are given n cities $\{1, \ldots, n\}$ and a non-negative integer distance d_{ij} between any two cities i and j, such that $d_{ij} = d_{ji}$ (that is, distances are symmetric). We are asked to find the shortest tour of the cities, that is, the permutation π such that the total distance*

$$\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)}$$

is as small as possible and $\pi(n+1) = \pi(1) = 1$, that is, we start at city 1.

A brute-force approach for solving this problem requires considering all $(n-1)!/2$ tours.

It turns out that TSP is $\mathcal{NP}$-complete. This follows from a reduction from a related problem, the problem of finding a Hamiltonian path in a graph: a Hamiltonian path is a simple path that visits **every** vertex in the graph. We can reduce 3SAT to Hamiltonian path to show the latter $\mathcal{NP}$-complete (e.g., see your textbook if you are interested). TSP is $\mathcal{NP}$-complete in the case of asymmetric distances and remains $\mathcal{NP}$-complete when the intercity distances satisfy the triangle inequality (that is, they correspond to distances on the plane).

6 Appendix

6.1 Properties of Boolean expressions

Boolean expressions possess some basic properties such as associativity, commutativity and distribution laws. We summarize them below.

1. $\neg\neg\phi \equiv \phi$
2. $(\phi_1 \vee \phi_2) \equiv (\phi_2 \vee \phi_1)$
3. $(\phi_1 \wedge \phi_2) \equiv (\phi_2 \wedge \phi_1)$
4. $((\phi_1 \vee \phi_2) \vee \phi_3) \equiv (\phi_1 \vee (\phi_2 \vee \phi_3))$
5. $((\phi_1 \wedge \phi_2) \wedge \phi_3) \equiv (\phi_1 \wedge (\phi_2 \wedge \phi_3))$
6. $((\phi_1 \vee \phi_2) \wedge \phi_3) \equiv ((\phi_1 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3))$
7. $((\phi_1 \wedge \phi_2) \vee \phi_3) \equiv ((\phi_1 \vee \phi_3) \wedge (\phi_2 \vee \phi_3))$

8. $\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_2)$
9. $\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2)$
10. $\phi_1 \vee \phi_1 \equiv \phi_1$
11. $\phi_1 \wedge \phi_1 \equiv \phi_1$