NP-completeness

Chuck Cusack, Megan Sheets, Matt Boettger, and Eric Larson

Revised and Edited by Chuck Cusack

These notes are based on chapter 26 of [1] and lectures from CSCE423/823, Spring 2001. We begin with a few definitions.

Definition 1 *A problem is* **tractable** *if it is solvable by a polynomial-time algorithm.*

Definition 2 *A problem is* **intractable** *if it requires a superpolynomial-time algorithm.*

There is an important class of problems, called **NP-complete**, whose status is unknown. No polynomial-time algorithm to solve them is known, but there is no proof that they are intractable, either. Most theoretical computer scientists believe that NP-complete problems are intractable.

1 Polynomial Time

There are three reasons we consider polynomial-time solvable problems tractable:

1. Very rarely will you come across a problem that requires $\Theta(n^{100})$. Most polynomial-time computable problems require much less time (e.g. $\Theta(n^2), \Theta(n^3), \Theta(n^4)$).
2. For many reasonable models of computation, a problem that can be solved in polynomial-time in one model can be solved in polynomial-time in another.
3. Since polynomials are closed under addition, multiplication, and composition, the class of polynomial-time solvable problems has nice closure properties.

1.1 Abstract Problems

Definition 3 *An* **abstract problem** *Q is a binary relation on a set I of problem instances and a set S of problem solutions.*

Example 1 Shortest-Path
Given an unweighted, undirected graph $G = (V, E)$ and two vertices $x, y \in V$, the Shortest Path *problem is the problem of finding the shortest path from x to y in G. That is, the path from x to y that requires the least number of edges.*

- *An* **instance** *i of* Shortest-Path *is a triple consisting of a graph and two vertices.*

- *A* **solution** *S is a path in G.*
- *The* **abstract problem** *for* Shortest-Path *is the set of pairs (i, s), such that s is a solution for instance i. That is, s is a sequence of vertices $v_1, \ldots, v_k$ in the graph such $v_1 = x$, $v_k = y$, $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots k-1$, and k is the smallest number for which this is true.*

There are two distinct types of abstract problems: **optimization problem** and **decision problems**.

Definition 4 *An* **optimization problems** *is a problem in which some value must be minimized or maximized. These are the types of problems you have probably seen many times in the past.*

Definition 5 *A* **decision problems** *is a problem that has a "yes/no" answer.*

The Shortest Path problem is an optimization problem. There is a related problem, called Path, which is a corresponding decision problem. We can describe the Path problem as follows.

Problem: Path

- **Instance:** A graph $G = (V, E)$, two vertices $x, y \in V$, and a positive integer k.
- **Solution:** A path from x to y.
- **Decision Problem:** Is there a solution of length at most k?
- **Related Optimization Problem:** A path of shortest length from x to y.

Given an instance i of Path, we say that i is a *yes* instance if there is a path from x to y of length at most k, and i is a *no* instance if there is no path from x to y of length at most k. We also use the notation Path(i)=1 (Path(i)=0) if i is a *yes* (*no*) instance. Similar notation is used for all decision problems.

When studying the theory of NP-completeness, we are only concerned with decision problems. This makes the theory more elegant, and much easier to deal with. Although this seems like a strong restriction, it is usually the case that a decision problem can be solved efficiently if and only if the related optimization problem can be solved efficiently. If there is a solution to the optimization problem, the decision problem can often be answered by comparing the answer with a decision parameter (For instance, with Path, find the shortest path, and compare the length to k.) Conversely, if the decision problem can be solved, a binary-search type algorithm generally yields an efficient algorithm to solve the optimization problem.

1.2 Encoding and Concrete Problems

Definition 6 *An* **encoding** *of a set S of abstract objects is a mapping e from S to the set of binary strings.*

Example 2 *Encoding integers*
The set of integers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ can be easily encoded by converting to binary. For instance, $e(17) = 10001$.

Example 3 *Encoding graphs*
A graph can be thought of as a listing of vertices, and a listing of pairs of vertices (the edges). Since each vertex can be labeled with an integer, a graph can be encoded by encoding each vertex and each edge by encoding in binary the indices of the vertices.

It is easy to see that virtually any problem can be encoded, as evidenced by the fact that we can store them in a computer, which certainly stores them with some binary encoding.

Definition 7 *A* **concrete problem** *is a problem whose instance set is the set of binary strings.*

The concept of concrete problems is important in the study of complexity theory in that it gives us a way to phrase any problem in the same language–that of binary strings.

Definition 8 *An algorithm* **solves** *a concrete problem in time $O(T(n))$ if whenever the algorithm is given an instance i of a concrete problem such that $|i| = n$ (that is, the size of the input is n), the algorithm produces a solution in at most $O(T(n))$ time.*

Definition 9 *A concrete problem is* **polynomial-time solvable** *if there is an algorithm that solves the problem in time $O(n^k)$ for some constant k, where n is the size of the problem.*

We now give the first formal definition of the complexity class P.

Definition 10 *The* **complexity class P** *is the set of concrete decision problems that are polynomial-time solvable.*

Since we solve concrete problems and not abstract problems, the encoding of an abstract problem is very important to our understanding of polynomial time. We might expect that any encoding will suffice. However, consider the following scenarios, given an integer n.

- If n is represented in unary (a string of n ones) then an algorithm with complexity $\Theta(n)$ is polynomial on the size of the input (n).
- If we use the more common binary representation then the input length is $k = \lceil \log n \rceil$ so an algorithm with complexity $\Theta(n) = \Theta(2^k)$ is exponential on the size of the input (k).

It appears that the way we encode a problem can dramatically change the complexity of an algorithm to solve the problem. On the surface, it appears that we cannot say anything about the complexity of solving an abstract problem. The following theorem gives us hope.

Theorem 1 *Let Q be an abstract problem on an instance set I, and let e_1 and e_2 be two encoding on I that are polynomially related. That is, given any instance $i \in I$ of an abstract problem, $e_1(i)$ can be computed from $e_2(i)$, and vice-versa, in polynomial time. Then $e_1(Q) \in P$ if and only if $E_2(Q) \in P$.*

In other words, any reasonable encoding (of which unary is not) of an abstract problem will result in similar running times. Since most of the encoding that naturally occur to us are reasonable, we usually do not have to be too concerned with the actual encoding used. We should be sure, however, that there actually is a reasonable encoding.

1.3 Formal Languages

In order to further develop our discussion of complexity classes, we first need to discuss some basic principles of formal languages.

Definition 11 *An* **alphabet** *Σ is a finite set of symbols.*

Definition 12 *A* **language** *L over Σ is a set of strings from Σ.*

Example 4 *Let $\Sigma = \{0, 1\}$. The set $L = \{10, 11, 101, 111, \ldots\}$ is the language of binary representations of prime numbers.*

Definition 13 *Σ^* is the set of all strings over Σ.*

Definition 14 *ε is the empty string. That is, it is the string with no characters.*

Definition 15 *The* **compliment** *of L is $\bar{L} = \Sigma^* - L$. That is, the compliment of L is the set of strings not in L.*

Definition 16 *The* **concatenation** *of two languages L_1 and L_2 is $L = \{x_1 x_2 : x_1 \in L_1, x_2 \in L_2\}$.*

Definition 17 *The* **closure** *of a language is $L^* = \{\varepsilon\} \cup L^1 \cup L^2 \cup L^3 \cup \cdots \cup L^k$ where L^k is the language obtained by concatenating L to itself k times.*

Since any decision problem Q can be be encoded, each instance of Q belongs to Σ^*, where $\Sigma = \{0, 1\}$. Thus, we can define each decision problem as a language L by

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

In other words, L is the set of *yes* instances of Q.

Definition 18 *An algorithm A* **accepts** *(***rejects***) a string $x \in \{0, 1\}^*$ if, given x, it outputs $A(x) = 1$ ($A(x) = 0$).*

If A accepts a language L and $x \notin L$, A does not necessarily return 0 – it may never terminate.

Definition 19 *The language* **accepted** *by an algorithm A is the set*

$$L = \{x \in \{0,1\}^* : A(x) = 1\}.$$

Definition 20 *A language L is* **decided** *by A if, for all $x \in L$, $A(x) = 1$ and for all $x \notin L$, $A(x) = 0$. In other words, A accepts every string in L and rejects every string not in L.*

Definition 21 *A language L is* **accepted (decided) in polynomial time** *by A if, for all $x \in \{0,1\}^*$, A accepts (accepts or rejects) x in time $O(n^k)$ for some k, where n is the size of x.*

We can re-define the complexity class **P**.

Definition 22 *The* **complexity class P** *is the set of languages L that are decided in polynomial time. In other words,*

$$P = \{L \subseteq \{0,1\}^* : \textit{There exists an algorithm } A \textit{ that } \text{decides } L \textit{ in polynomial time}\}$$

It is not hard to show the following equivalent definition of **P**.

Theorem 2

$$P = \{L \subseteq \{0,1\}^* : \textit{There exists an algorithm } A \textit{ that } \text{accepts } L \textit{ in polynomial time}\}$$

Proof: This is only a sketch of the proof. Since a polynomial-time algorithm A exists to accept L, it has running time $O(n^k)$ for some k. In other words, the running time is at most cn^k for some constant c. An algorithm A' to decide L runs algorithm A for at least cn^k steps. If A has accepted x, A' accepts x. If A has not accepted x, it never will, since it has run long enough. Therefore A' rejects x. Thus A' decides L in polynomial time. $\square$

Think about why it might be difficult in practice to actually implement the decision algorithm described in the previous proof.

2 Polynomial-Time Verification

Definition 23 *A* **verification algorithm** *is an algorithm that takes two inputs:*

1. *A binary string x from the language.*
2. *A certificate y (also a binary string).*

Definition 24 *An algorithm* **verifies** *an input x if there exists a certificate y such that $A(x, y) = 1$.*

You can think of a certificate as being a set of data that allows one to easily see that an instance is a *yes* instance. For instance, a certificate for PATH could be a list of vertices.

Definition 25 *The* **language verified** *by A is $L = \{x \in \{0,1\}^* : \exists y \text{ such that } A(x, y) = 1\}$.*

Clearly if $x \notin L$, there should be no certificate y such that $A(x, y) = 1$. Also, the algorithm A does not have to use the input y.

Definition 26 *The* **complexity class NP** *(standing for* nondeterministic polynomial*) is the class of languages that can be verified by a polynomial time algorithm A. That is, a language L is in* **NP** *if there is a polynomial-time algorithm A, and constant c such that*

$$L = \{x \in \{0,1\}^* : \textit{there exists a certificate } y \textit{ with } |y| = O(|x|^c) \textit{ such that } A(x,y) = 1\}$$

Definition 27 *The* **complexity class co-NP** *is the class of languages $\{L : \overline{L} \in \mathbf{NP}\}$. That is, the class of languages whose complement is in* **NP**.

If $L \in P$, then $L \in NP$, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm. Thus $P \subseteq NP$. Unfortunately, the ability to verify a language in polynomial time does not seem to guarantee that it can be determined in polynomial time. We further develop this idea next.

3 NP-completeness and Reducibility

The class NP-complete has the property that if any one NP-complete problem can be solved in polynomial-time, then every problem in NP has a polynomial-time solution. That is, it would imply that $P = NP$. After decades of study, no polynomial-time algorithm has ever been discovered to solve any NP-complete problem, yet nobody has proven one does not exist, either. Some would consider the question of whether or not $P = NP$ the most important open question in theoretical computer science.

Definition 28 *We say that a language L_1 is* **polynomial-time reducible** *to a language L_2, written $L_1 \leq_p L_2$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1 \iff f(x) \in L_2$. We call the function f the* **reduction function** *and the polynomial-time algorithm F that computes f the* **reduction algorithm**.

We will see many examples of polynomial-time reductions in the next section. The following theorem shows one important property of polynomial-time reductions.

Theorem 3 *If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P \Rightarrow L_1 \in P$.*

Proof: Let A_2 be the polynomial-time algorithm to solve L_2, and f a reduction function from L_1 to L_2. An algorithm to solve L_1 can take an input $x \in \{0,1\}^*$, compute $f(x)$ in polynomial time, and use A_2 to determine if $f(x) \in L_2$ in polynomial time. The algorithm returns the result from A_2 since $x \in L_1$ if and only if $f(x) \in L_2$, and the algorithm is clearly polynomial-time. $\square$

Definition 29 *Let L be a language such that $L' \leq_p L$ for every $L' \in NP$. Then we say that L is* **NP-hard**.

In English, Definition 29 says that a language L is **NP-Hard** if *every* problem in **NP** is polynomial-time reducible to L. But how can we prove that every language in **NP** is in polynomial-time reducible to L? We do not even know every language in **NP**! As we will see soon, this is actually possible. Now the moment we have all been waiting for–the definition of **NP-Complete**.

Definition 30 *A language $L \subseteq \{0,1\}^*$ is* **NP-complete (NPC)** *if*

1. *L is* **NP** *and*
2. *L is* **NP-hard***.*

Theorem 4

1. *If any problem in NPC is polynomial-time solvable, then $P = NP$.*
2. *If there exists any problem in NP that is not polynomial-time solvable, then all NPC problems are not polynomial-time solvable.*

Proof:

1. Suppose $L \in P$ and that $L \in NPC$. For any $L' \in NP$ we have $L' \leq_p L$ by property 2 of the definition of NPC. By Theorem 3, we have that $L' \in P$.
2. Suppose there exists an $L \in NP$ such that $L \notin P$. Let L' be any polynomial-time solvable problem in NPC. Then $L \leq_p L'$, and Theorem 3 implies that $L \in P$, contradicting our assumption. □

This theorem is at the heart of why NPC is such an important class of problems. NPC can be thought of as the hardest problems in NP. Hundreds of problems have been proven to be NPC, and thousands of researchers have tried and failed to find a polynomial time algorithm to solve any of them. This leads most people to believe that $NP \neq P$. On the other hand, they have also failed to prove that the best algorithm possible for any of these problems is exponential.

3.1 Circuit Satisfiability

Problem: Circuit-SAT

- **Instance:** A boolean combinational circuit with one output. That is, a circuit consisting of input wires, **AND**, **OR**, **NOT**, and other boolean gates, and one output wire.
- **Decision Problem:** Can boolean values be assigned to the inputs so that the output is 1?

We will give the sketch of a proof that Circuit-SAT is NPC.

Theorem 5 *The* Circuit-SAT *problem is NP-complete.*

Proof: To show that CIRCUIT-SAT is NP we will provide a two-input, polynomial-time algorithm A to verify **CIRCUIT-SAT**. The first input is a boolean combinational circuit C. The certificate will consist of boolean values assigned to the inputs of C.

The algorithm A will compute the output of the circuit by computing the result of each gate from inputs to outputs (which can be done in polynomial time). A will output the the result of the circuit, since the circuit outputs 1 if and only if the certificate is a satisfying truth assignment. Notice that if C is not satisfiable, no satisfying truth assignment exists, and A will always return 0 as required. Clearly A will work if C is satisfiable, since it will return 1 when given a valid certificate. Thus A is a polynomial-time verification algorithm, and CIRCUIT-SAT is NP.

In order to show that CIRCUIT-SAT is NP-hard, we need to show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. The proof of this is beyond the scope of these notes, so we will only give a very sketchy proof of this fact. In fact, if you are convinced of the fact that CIRCUIT-SAT is NP-hard by reading this proof, you are either easily fooled or very bright.

If a language L is in NP, there is a polynomial-time verification algorithm A for L. A verification algorithm can be implemented on a computer, and computers consist of boolean circuits with feedback, memory, etc. The trick is to modify the circuits in a computer to remove feedback, memory, etc., by creating as many copies of the circuit as possible and having all connections go forward. We need one copy for each step of the algorithm. The input wires will contain the values of the inputs, including the certificate, and the output of the circuit will be the output of the algorithm A. With a little thought, you can convince yourself that the size of the circuit is polynomial in the size of the input.

Notice that if L is a *yes* instance, there is some certificate that will results in the value 1 being returned by A, and if L is a *no* instance, there is no certificate that will result in 1 being returned by A. Since our circuit returns the result of A, the circuit is satisfiable (is a *yes* instance) if and only if L is a *yes* instance. Thus, L is reduced to CIRCUIT-SAT. $\square$

We have now established (well, sort of) the existence of at least one NP-Complete problem. As we see next, this is more significant than is immediately obvious.

4 NP-completeness Proofs

The proof that CIRCUIT-SAT is NP-complete relies on a direct proof that for any language $L \in NP$, $L \leq_p$ CIRCUIT-SAT. The next lemma shows that we do not have to go to all of this work for every language we wish to show is NPC.

Lemma 6 *If L is a language such that $L' \leq_p L$ for some $L' \in NPC$ then L is NP-hard. Moreover, if $L \in NP$, then $L \in NPC$.*

Proof: For any language $L'' \in NP$, we know that $L'' \leq_p L'$, since L' is NP-complete. Since $L' \leq_p L$, then $L'' \leq_p L$, since the relation $\leq_p$ is transitive (A polynomial-time reduction followed by a polynomial-time reduction is still polynomial-time). Thus L is NP-hard. If $L \in NP$ we also have $L \in NPC$ by definition. $\square$

By reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L. Thus, the above lemma gives us a nice method for proving that a language L is NP-complete.

1. Prove $L \in NP$.
2. Select a known NP-complete language L'.
3. Describe an algorithm that computes a function f mapping every instance of L' to an instance of L.
4. Prove that the function f satisfies $x \in L' \iff f(x) \in L$ for all $x \in \{0,1\}^*$. There are two common ways to prove that $x \in L' \iff f(x) \in L$ for all $x \in \{0,1\}^*$.
   (a) Show that $x \in L' \implies f(x) \in L$ and that $f(x) \in L \implies x \in L'$.
   (b) Show that $x \in L' \implies f(x) \in L$ and that $x \notin L' \implies f(x) \notin L$.
5. Prove that the algorithm computing f runs in polynomial time.

In the remaining sections we give proofs that several other problems are NP-complete. Figure 1 shows the reductions we use in each of the proofs.

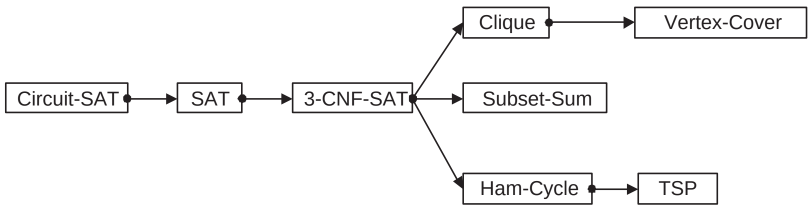

Figure 1: The structure of reductions used in these notes.

4.1 SAT

Problem: SAT

- **Instance:** A boolean formula involving a set of n boolean variable $x_1, \ldots x_n$, m boolean connectives (including **AND** ($\wedge$), **OR** ($\vee$), **NOT** ($\neg$), **implication** ($\rightarrow$), and **iff** ($\leftrightarrow$)), and parentheses.
- **Decision Problem:** Is there a satisfying truth assignment? That is, can each of the variables be assigned truth values so the formula is true?

Theorem 7 SAT *is NP-complete.*

Proof: A certificate for an instance x of SAT consists of a truth assignment for the input variables of x. The verification algorithm replaces each of the variables with the truth assignments from the certificate, and evaluates the value of the expression, which can be done in polynomial time. Since the expression can only evaluate to 1 if the formula is satisfiable, we have a polynomial-time verification algorithm for SAT. That is, SAT is NP.

We will reduce Circuit-SAT to SAT. Let C be an instance of Circuit-SAT. We leave it to the reader to verify that the obvious reduction (replacing circuit with expression) can not always be done in polynomial time.

We reduce as follows. Let C be an instance of Circuit-SAT. First, label each wire (including the inputs) in the circuit. For each gate, create an expression that describes the proper operation of the gate. For instance, if the circuit contains an OR gate with inputs x_1, x_2, and x_3, and output x_4, create the expression $(x_1 \vee x_2 \vee x_3) \leftrightarrow x_4$. The formula we construct is the AND of the expression for each gate and the final output of the gate. Clearly we can construct this formula in polynomial time.

As an example of the construction, consider Figure 2. Notice the inputs are $x_1,\ldots,$ x_5, the values $x_6,\ldots,$ x_{10} are intermediate values, and x_{11} is the output of the circuit. The corresponding expression for this circuit is

$$\begin{aligned} x_{11} \quad \wedge &\quad [(x_1 \wedge x_2) \leftrightarrow x_6] \\ \wedge &\quad [(\neg x_3) \leftrightarrow x_7] \\ \wedge &\quad [(x_4 \vee x_6 \vee x_7) \leftrightarrow x_8] \\ \wedge &\quad [(x_6 \wedge x_8 \wedge x_4 \wedge x_5) \leftrightarrow x_9] \\ \wedge &\quad [(\neg x_5) \leftrightarrow x_{10}] \\ \wedge &\quad [(x_9 \vee x_{10}) \leftrightarrow x_{11}] \end{aligned}$$

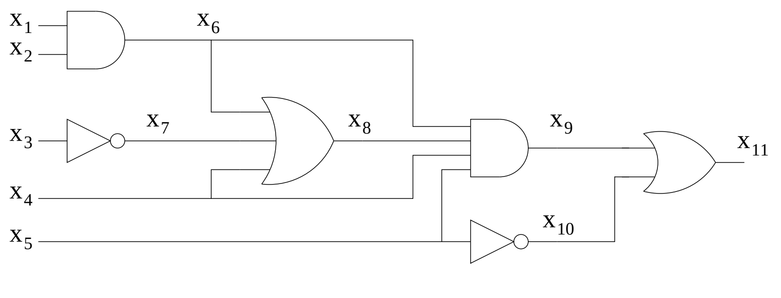

Figure 2: An instance of Circuit-SAT

If x is a satisfiable circuit, then there is a truth assignment for the inputs of the circuit such that the output will be 1. Assigning these same values to the expression, including the correct results for each gate, will result in a value of 1 for the expression as well. Conversely, assume x is not satisfiable. Since the expression contains the output ANDed with the rest of the circuit, and the output cannot be 1, neither can the expression. Thus x is satisfiable if and only if the corresponding expression is. □

4.2 3-CNF-Satisfiability

Definition 31 *A* **literal** *is an occurrence in a boolean formula of a variable or its negation.*

Definition 32 *A boolean formula is in* **conjunctive normal form (CNF)** *if it is expressed as an AND of OR clauses consisting of one or more literals.*

Definition 33 *A boolean formula is in* **3-conjunctive normal form (3-CNF)** *if each clause has three distinct literals (e.g.* $(x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee z)$*).*

Problem: 3-CNF-SAT

- **Instance:** A boolean formula in 3-conjunctive normal form.
- **Decision Problem:** Is there a satisfying truth assignment? That is, can each of the variables be assigned truth values so the formula is true?

Notice that 3-CNF-SAT is just a special case of SAT. It turns out to be more useful to use to reduce from than SAT since it has a well-defined structure. Do not make the mistake of assuming that since it is a special case, it is clearly NPC. In fact the special case 2-CNF-SAT is solvable in polynomial time. As another example, just because finding the roots of a polynomial of degree 1 is trivial, that does not imply that finding the roots of a polynomial of degree 100 is trivial.

Theorem 8 3-CNF-SAT *is NP-complete.*

Proof: Since an instance of 3-CNF-SAT is also an instance of SAT, 3-CNF-SAT is NP.

We will reduce SAT to 3-CNF-SAT. This reduction involves several steps. We will specify each step and give an example to make the reduction easier to understand. Let φ_0 be an instance of SAT. For example,

$$\varphi_0 = [\neg(x_1 \wedge x_2) \vee x_1 \vee \neg x_2 \vee x_3] \leftrightarrow [\neg x_2 \rightarrow x_4]$$

Step 1 Let φ_1 be a full parenthesization of φ_0 so that each connective is applied to at most 2 variables. For our example,

$$\varphi_1 = [(\neg(x_1 \wedge x_2)) \vee ((x_1 \vee (\neg x_2)) \vee x_3)] \leftrightarrow [(\neg x_2) \rightarrow x_4]$$

Step 2 Construct a binary parse tree T_{φ_1} for the formula φ_1 with literals as leaves and connectives as internal nodes. For example, the tree corresponding to φ_1 is given in Figure 3.

Notice that the binary parse tree can be viewed as a circuit for computing the function, with the leaves as input, and the value on top of the root as the result. We label the outputs of each node by variables y_i.

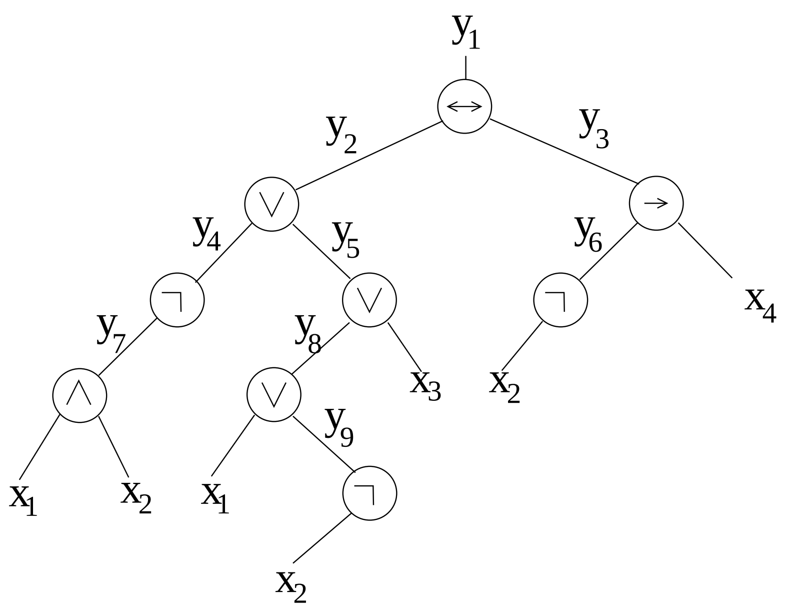

Figure 3: The binary tree T_{φ_1}corresponding to φ_1

Step 3 Rewrite the original formula φ_1 as the AND of the root variable and a conjunction of clauses describing the operation of each node in T_{φ_1}, similar to the construction used in the proof of SAT. In our example, the result is

$$\begin{aligned}\varphi_2 = y_1 \quad & \wedge \quad [y_1 \leftrightarrow (y_2 \leftrightarrow y_3)] \\ & \wedge \quad [y_2 \leftrightarrow (y_4 \vee y_5)] \\ & \wedge \quad [y_3 \leftrightarrow (y_6 \rightarrow x_4)] \\ & \wedge \quad [y_4 \leftrightarrow (\neg y_7)] \\ & \wedge \quad [y_5 \leftrightarrow (y_8 \vee x_3)] \\ & \wedge \quad [y_6 \leftrightarrow (\neg x_2)] \\ & \wedge \quad [y_7 \leftrightarrow (x_1 \wedge x_2)] \\ & \wedge \quad [y_8 \leftrightarrow (x_1 \vee y_9)] \\ & \wedge \quad [y_9 \leftrightarrow (\neg x_2)]\end{aligned}$$

Notice that since each node in the tree has at most two children and exactly one parent, The resulting formula φ_2 obtained is a conjunction of clauses $\varphi_{2,i}$, each of which has at most three literals.

Step 4 We need to write each clause $\varphi_{2,i}$ as an OR of literals. For 1 literal, there is nothing to do. We will show how to do this for clauses with 2 or 3 literals, using $\varphi_{2,3} = y_2 \leftrightarrow (y_4 \vee y_5)$, and $\varphi_{2,5} = y_4 \leftrightarrow (\neg y_7)$ to demonstrate.

4a. For each i, draw the truth table for $\varphi_{2,i}$. In our example, the truth tables are as follows

y_2	y_4	y_5	$\varphi_{2,3}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

y_4	y_7	$\varphi_{2,5}$
0	0	0
0	1	1
1	0	1
1	1	0

4b. Compute the disjunctive normal form (DNF) of $\neg\varphi_{2,i}$ by the usual method. In our examples,

$$\neg\varphi_{2,3} = (\neg y_2 \wedge \neg y_4 \wedge y_5) \vee (\neg y_2 \wedge y_4 \wedge \neg y_5) \vee (\neg y_2 \wedge y_4 \wedge y_5) \vee (y_2 \wedge \neg y_4 \wedge \neg y_5)$$

$$\neg\varphi_{2,5} = (y_4 \wedge y_7) \vee (\neg y_4 \wedge \neg y_7)$$

4c. Compute the CNF $\varphi_{3,i}$ of $\varphi_{2,i}$ by negating $\neg\varphi_{2,i}$ and applying DeMorgan's Laws. In our examples, we get

$$\varphi_{3,3} = (y_2 \vee y_4 \vee \neg y_5) \wedge (y_2 \vee \neg y_4 \vee y_5) \wedge (y_2 \vee \neg y_4 \vee \neg y_5) \wedge (\neg y_2 \vee y_4 \vee y_5)$$

$$\varphi_{3,5} = (\neg y_4 \vee \neg y_7) \wedge (y_4 \vee y_7)$$

Notice that each clause of $\varphi_{3,i}$ is the OR of *at most* three literals.

Step 5 We need to make sure every clause has *exactly* three literals. We introduce two new variables p and q to assist. We replace each $\varphi_{3,i}$, with an equivalent $\varphi_{4,i}$ that has exactly three literals. There are three cases.

- If $\varphi_{3,i}$ has 3 distinct literals, let $\varphi_{4,i} = \varphi_{3,i}$.
- If $\varphi_{3,i}$ has 2 distinct literals, then $\varphi_{3,i} = l_1 \vee l_2$, for some literals l_1 and l_2. Let $\varphi_{4,i} = (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$
- If $\varphi_{3,i}$ has 1 literal l, then let $\varphi_{4,i} = (l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. It is easy to check that for any truth assignment of p and q, $\varphi_{3,i}$ and $\varphi_{4,i}$ have the same truth values.

Step 6 Let φ_4 be the AND of all of the $\varphi_{4,i}$.

Since in each step we assured that the new expressions were equivalent to the old ones, φ_4 is equivalent to φ_0. In other words, φ_4 is satisfiable if and only if φ_0 is satisfiable.

Lastly, we need to show that the reduction can be done in polynomial time. Steps 1, 2, and 3 take linear time in the size of the input, and the formula φ_2 is at most double the size of φ_1, since for each connective we add at most one literal (the output) and one connective (the $\leftrightarrow$). Since each clause $\varphi_{2,i}$ contains at most 3 literals, the truth tables for each contains at most 8 rows, so the DNF for $\neg\varphi_{2,i}$ is no larger than 8 times the size of $\varphi_{2,i}$. The size of

$\varphi_{3,i}$ is the same as that of $\neg\varphi_{2,i}$. Lastly, it is clear that the size of $\varphi_{4,i}$ is no more than 4 times the size of $\varphi_{3,i}$. In all, the size of φ_4 is polynomial in the size of φ_0, and each step can be accomplished in polynomial time. □

4.3 The Clique Problem

Definition 34 *A* **clique** *is a complete subgraph of an undirected graph $G = (V, E)$. That is, a subset $V' \subseteq V$ such that the induced subgraph on V' is $K_{|V'|}$.*

The size of a clique is the number of vertices it contains. A clique of size k is called a k-clique.

Problem: CLIQUE

- **Instance:** A graph $G = (V, E)$, and a positive integer k.
- **Decision Problem:** Does G contain a k-clique?
- **Related Optimization Problem:** Find the largest k such that G contains a k-clique.

Theorem 9 CLIQUE *is NP-complete.*

Proof: Let $V' \subseteq V$ be our certificate. To check the certificate, first make sure $|V'| = k$. Next, for each $u, v \in V'$, check whether or not $(u, v) \in E$, returning 1 if and only if $(u, v) \in E$ for every pair $u, v \in V'$. It should be clear that the verification algorithm works, that a graph with no k-clique does not have a valid certificate, and the verification can be done in polynomial time.

We now reduce 3-CNF-SAT to CLIQUE. Let $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be an instance of 3-CNF-SAT with k clauses. For each clause $C_r = ({l_1}^r \vee {l_2}^r \vee {l_3}^r)$ in φ, we place a triple of vertices ${v_1}^r, {v_2}^r$ and ${v_3}^r$ in V. An edge is placed between two vertices ${v_i}^r$ and ${v_j}^s$ if $r \neq s$ and ${l_i}^r \neq \neg {l_j}^s$. In other words, they are in different clauses, and the corresponding variables are consistent. We state this clearly as a lemma.

Lemma 10 *There is an edge between ${v_i}^r$ and ${v_j}^s$ if and only if it is possible to assign both ${l_i}^r$ and ${l_j}^s$ to be true.*

Clearly we can construct the graph from φ in polynomial time.

Now, assume φ has a satisfying assignment. Then each clause C_r contains at least one literal ${l_i}^r$ that can be made true, and each such literal corresponds to a vertex ${v_i}^r$. Picking one such "true" literal from each clause yields a set V' of k vertices, all of which can be connected to each other by the lemma. Thus V' is a k-clique.

Conversely, let G have a k-clique V'. Clearly V' contains one vertex from each triple. Thus, each clause has a literal corresponding to a vertex in the clique. By the lemma, each of these literals can be assigned the value true, and thus each clause is satisfied, and therefore φ is satisfied. □

4.4 The Vertex-Cover Problem

Definition 35 *A* **vertex cover** *of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ and/or $v \in V'$.*

The size of a vertex cover is the number of vertices in it. Figure 4 shows a simple example of a vertex cover.

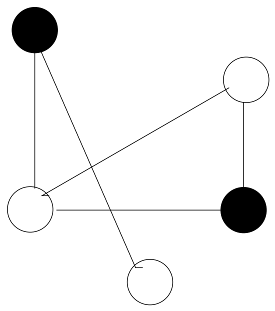

Figure 4: The white vertices represent a vertex cover for the graph

Problem: VERTEX-COVER

- **Instance:** A graph $G = (V, E)$, and a positive integer k.
- **Solution:** A vertex cover of G.
- **Decision Problem:** Does G contain a vertex cover containing k vertices?
- **Related Optimization Problem:** Find the smallest k for which there exists a vertex cover with k vertices.

Theorem 11 VERTEX-COVER *is NP-complete*

Proof: Given a graph $G = (V, E)$ and an integer k, let the certificate be the vertex cover $V' \subseteq V$. The verification algorithm checks that $|V'| = k$ and for each edge $(u, v) \in E$, determines whether or not either $u \in V'$ or $v \in V'$, returning 1 if and only if this is always the case. Clearly this can only return 1 if G contains a vertex cover of size k, and works properly given a valid certificate. It also takes polynomial time.

We now reduce CLIQUE to VERTEX-COVER. Given an undirected graph $G = (V, E)$ the **compliment** of G is defined as $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{(u, v) : (u, v) \notin E\}$. We can compute this in polynomial time.

Suppose G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\bar{G}$. Let $(u, v) \in \bar{E}$. Then $(u, v) \notin E$ which means u and/or v does not belong to V' since all pairs of vertices in V' are connected by an edge E. Thus u and/or v is in $V - V'$, meaning (u, v) is covered by $V - V'$. Thus all edges in $\bar{E}$ are covered by a vertex in $V - V'$. Therefore, the set $V - V'$ is a vertex cover of $\bar{G}$ of size $|V| - k$.

Let $\bar{G}$ have a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. For all $u, v \in V$, if $(u, v) \in \bar{E}$, then $u \in V'$ and/or $v \in V'$. For all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, if u and v are two vertices that are not in the vertex cover V', then $(u, v) \notin \bar{E}$, so $(u, v) \in E$. Since this is true of any pair of vertices from $V - V'$, $V - V'$ is a clique and has size $|V| - |V'| = k$. □

4.5 The Subset-Sum Problem

Problem: SUBSET-SUM

- **Instance:** A finite set of positive integers S, and a target integer t.
- **Solution:** A subset of S.
- **Decision Problem:** Is there a subset of S such that the sum of the numbers in the subset is exactly t?
- **Related Optimization Problem:** Find a subset of S such that the sum of the numbers is as close to t as possible.

Example 5 *Let* $S = \{1, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ *and* $t = 3754$. *One possible solution is* $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$.

Theorem 12 SUBSET-SUM *is NP-complete.*

Proof: We will not give the proof here, but an example that should give enough details for the reader to figure out the general idea. The reduction is from 3-CNF-SAT. Let $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z)$ be an instance of 3-CNF-SAT. Create a table with rows for each variable, each variable's negation, and two rows for each clause. Insert a column for each variable and each clause. Fill in the table following the pattern in Table 1. Interpret each row as a decimal number, including the target. With a little work, you can convince yourself that this is a valid reduction. □

Another proof reduces from VERTEX-COVER.

4.6 The Hamiltonian-Cycle Problem

Problem: HAM-CYCLE

- **Instance:** A graph $G = (V, E)$.
- **Decision Problem:** Does G have a Hamiltonian cycle? That is, does G contain a simple cycle containing all vertices of V?

Theorem 13 HAM-CYCLE *is NP-complete.*

Proof: See page 954-959 of [1] to see that HAM-CYCLE is NP-complete. The reduction is from 3-CNF-SAT.

Clause/Var	x	y	z	Clause 1	Clause 2	Clause 3	
x	1	0	0	1	0	1	*
$\neg x$	1	0	0	0	1	0	
y	0	1	0	1	1	0	*
$\neg y$	0	1	0	0	0	1	
z	0	0	1	0	1	1	*
$\neg z$	0	0	1	1	0	0	
Clause 1	0	0	0	1	0	0	
Slop	0	0	0	2	0	0	*
Clause 2	0	0	0	0	1	0	
Slop	0	0	0	0	2	0	*
Clause 3	0	0	0	0	0	1	
Slop	0	0	0	0	0	2	*
target	1	1	1	4	4	4	

Table 1: Table for the Subset-Sum Problem. Select rows (e.g. the rows indicated by the *) such that the numbers in these rows sum to the target.

4.7 The Traveling-Salesman Problem

Problem: TSP

- **Instance:** A complete graph $G = (V, E)$, a function $c(x, y)$ such that for each $x, y \in V$, $c(x, y)$ is a positive integer called the *cost* to go from x to y, a maximum cost k, a positive integer.
- **Solution:** A tour of G. That is, an ordering of the vertices. $v_1, \ldots, v_n$. The cost of a tour is the sum of the costs of each of the edges in the ordering. That is $C = \sum_{i=1}^{n-1} c(v_i, v_{i+1}) + c(v_n, v_1)$.
- **Decision Problem:** Does G contain a tour of cost at most k?
- **Related Optimization Problem:** Find the lowest cost tour of G.

Theorem 14 TSP *is NP-complete.*

Proof: A certificate is an ordering of the vertices. The verification algorithm checks that the sequence contains each vertex exactly once, sums up the edge costs, and checks to see that this cost is at most k, which can easily be done in polynomial time.

We reduce from HAM-CYCLE. Let $G = (V, E)$ be an instance of HAM-CYCLE. Construct an instance of TSP as follows: form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V\}$, and define the cost function c by

$$c(i,j) = \begin{cases} 0, & \text{if } (i,j) \in E, \\ 1, & \text{if } (i,j) \notin E. \end{cases}$$

Lastly let the cost be 0. Clearly this can be done in polynomial time.

Now we must show that G has a Hamiltonian cycle iff G' has a tour with cost a most 0. Suppose G has a Hamiltonian cycle h. Thus each edge h belongs to E and thus a tour has cost 0 in G'.

Conversely, suppose G' has a tour h' with cost≤ 0. Since the cost of edges in E' are 0 and 1, the cost of h' is exactly 0. Therefore, h' contains only edges in E. Therefore h is a Hamiltonian cycle in G. $\square$

References

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.