17 Examples of NP-Complete Problems

17.1 SAT

Section 15.4 defines the Satisfiability Problem for Propositional Logic (SATPL). We have seen that SATPL is in NP, and noticed in (Section 14) that SATPL appears to be difficult to solve.

Here, we look at a restriction of that problem to *clasual* propositional formulas. (The restriction can only make the problem easier, if anything.)

Definition 17.1. A *literal* is either a propositional variable or its negation. We will use lower-case letters such as x, y and z as propositional variables. Rather than writing $\neg y$ to indicate negated variable y, we write $\overline{y}$. Literal x is a *positive literal* and $\overline{y}$ is a *negative literal*.

Definition 17.2. A *clause* is a disjunction ($\vee$) of one or more literals. For example, $x \vee \overline{z} \vee y$ is a clause. A literal by itself is a clause with just one literal.

Definition 17.3. A *clausal formula* is a conjunction ($\wedge$) of one or more clauses. For example, $(x) \wedge (\overline{y} \vee z) \wedge (\overline{x} \vee y \vee \overline{z})$ is a clausal formula. There can be just one clause.

Definition 17.4. *SAT* is the following decision problem.

> **Input.** A clausal propositional formula ϕ.
> **Question.** Is ϕ satisfiable?

For example, is $(x) \wedge (\overline{y} \vee z) \wedge (\overline{x} \vee y \vee \overline{z})$ satisfiable? Of course: choose x, y and z all to be true. It is easy to check whether short propositional formulas are satisfiable. It is the long formulas that present difficulties!

Theorem 17.5. SAT $\in$ NP.

Proof. Theorem 15.5 shows that SATPL is in NP. But SAT is a restriction of SATPL. The evidence checker for SATPL also works for SAT.

♢

Cook and Levin independently showed that SAT is NP-complete. The proof is too long for this course, so we will need to accept it as proved.

Theorem 17.6. (Cook/Levin Theorem) SAT is NP-complete.

That gives us evidence (but not a proof, since we don't know whether P $\neq$ NP) that there is no polynomial-time algorithm for SAT.

17.2 Proving NP-Completeness

SAT is proved NP-complete using a difficult kind of reduction called a *generic reduction*. If all you know about X is that $X \in$ NP, you can ask someone to give you an evidence checker for X. The generic reduction from X to SAT converts that evidence checker into a propositional formula. You can think of it as building a simple computer from logic gates that do nots, ands and ors.

But we don't need to do a generic reduction for every proof of NP-completeness.

Theorem 17.7. Suppose that language $B \in$ NP, A is NP-complete and $A \leq_p B$. Then B is NP-complete.

Proof. $B \in$ NP, so it suffices to show that $X \leq_p B$ for every language $X \in$ NP. Since A is NP-complete, we know that $X \leq_p A$ for every language $X \in$ NP. But $A \leq_p B$ and relation $\leq_p$ is transitive (Theorem 16.4), so $X \leq_p B$ for every language $X \in$ NP.

♢

17.3 3-SAT

We can restrict the satisfiability problem further.

Definition 17.8. A propositional formula is in *3-clausal form* if it is in clausal form and has exactly 3 literals per clause. For example, $(x \vee y \vee z) \wedge$

$(\overline{y} \vee z \vee \overline{w})$ is in 3-clausal form. A propositional formula in 3-clausal form is called a *3-clausal propositional formula*.

Definition 17.9. *3-SAT* is the following decision problem.

Input. A 3-clausal propositional formula ϕ.
Question. Is ϕ satisfiable?

Theorem 17.10. 3-SAT is NP-complete.

Proof. Clearly, 3-SAT is in NP. (Use the same evidence checker as for SATPL.) So, by Theorem 17.7, it suffices to reduce SAT to 3-SAT.

We need a polynomial-time algorithm that takes a clausal formula ϕ and builds a 3-clausal formula ϕ' such that ϕ is satisfiable if and only if ϕ' is satisfiable. Our algorithm will convert each clause of ϕ separately.

Clauses that already have 3 literals are left alone. Clauses with fewer than 3 literals are easy to deal with by duplicating one or more of the literals. For example, clause $(A \vee B)$ is equivalent to $(A \vee A \vee B)$.

That only leaves *long clauses*, which have more than 3 literals. As long as there is at least one long clause, we find one with $n > 3$ literals and replace it by a clause that has $n-1$ literals, plus a clause with 3 literals. By repeating that, we can get rid of all of the long clauses. It is just a matter of ensuring that each step preserves satisfiability.

Suppose that ϕ contains clause

$$C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$$

where $n > 3$. Create a new variable u and replace C by pair of clauses

$$C' = (\ell_1 \vee \cdots \vee \ell_{n-2} \vee u) \wedge (\overline{u} \vee \ell_{n-1} \vee \ell_n)$$

yielding new formula ϕ_1. We need to show that ϕ_1 is satisfiable if and only if ϕ is satisfiable, showing that the modification of ϕ preserves satisfiability.

Claim 1. If ϕ_1 is satisfiable then ϕ is satisfiable. In fact, every truth-value assignment that satisfies ϕ_1 also satisfies ϕ.

Proof of Claim 1. Suppose ϕ_1 is satisfiable. Choose a truth-value assignment a that makes ϕ_1 true. That assignment must make all of the clauses other than C in ϕ true, since those clauses also occur in ϕ_1. We just need to

argue that assignment a also makes clause C true. Be sure to notice that, because a makes all clauses in ϕ_1 true, it makes both clauses in C' true.

Suppose $a(u) = \text{F}$. Then, because a makes clause $(\ell_1 \vee \cdots \vee \ell_{n-2} \vee u)$ true, a must make at least one of $\ell_1, \ldots, \ell_{n-2}$ true. But that means a makes clause C true.

Suppose that $a(u) = \text{T}$. Then, because a makes clause $(\overline{u} \vee \ell_{n-1} \vee \ell_n)$ true, a makes at least one of ℓ_{n-1} and ℓ_n true. Again, a makes clause C true.

Claim 2. If ϕ is satisfiable then ϕ_1 is satisfiable.

Proof of Claim 2. Suppose that ϕ is satisfiable, and choose a truth-value assignment a that makes ϕ true. Clause C must have at least one true literal.

If a makes ℓ_i true where $i \leq n-2$, then extend a by adding $u = \text{F}$. The new truth-value assignment makes clause $(\ell_1 \vee \cdots \vee \ell_{n-2} \vee u)$ true because ℓ_i is true, and it makes clause $(\overline{u} \vee \ell_{n-1} \vee \ell_n)$ true because $u = \text{F}$.

If a makes ℓ_i true where $i > n-2$, then extend a by adding $u = \text{T}$. You can check that both clauses of C' must be true.

$\diamondsuit$

17.4 The Vertex Cover Problem

Recall from Section 15.4.2 that a vertex cover of a simple graph is a set C of vertices so every edge is incident on at least one vertex in C. The Vertex Cover Problem VCP is the following decision problem.

> **Input.** A simple graph G and a positive integer k.
> **Question.** Does there exist a vertex cover C of G where $|C| \leq k$?

It is worth asking whether there is an obvious polynomial-time algorithm for VCP. One idea is to use a *greedy algorithm*, which tries to optimize globally by optimizing locally. Since we want to select as few vertices as possible to cover all of the edges, it makes sense to start by selecting a vertex with highest degree, since it covers as many edges as possible with the first pick. After that, remove the selected vertex and all of the edges that it covers, and repeat, again selecting a vertex with the highest degree.

That algorithm seems appealing, but does it work? Look at graph G_1 in Figure 17.1. The diagram shows a vertex cover of G_1 of size 5, but G_1 also has a vertex cover of size 4. (Can you find it?) G_1 has a vertex of degree 4, and all other vertices have degree 2 or 3. But the degree 4 is not part of any smallest vertex cover of G_1! If you are trying to determine whether G_1 has a vertex cover of size at most 4, you will be led astray by selecting the degree 4 vertex. Something is wrong with the greedy Vertex Cover algorithm.

It is tempting to try to patch the greedy algorithm. But is that worthwhile? The following theorem shows that it is a waste of time.

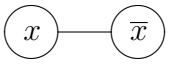
Theorem 17.11. VCP is NP-complete.

If the conjecture P $\neq$ NP is true then there does not exist a polynomial-time algorithm for VCP. Even if the conjecture is wrong, finding a polynomial-time algorithm for VCP is as difficult as proving that P = NP, since the existence of such an algorithm implies P = NP.

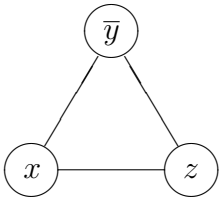
Proof of Theorem 17.11 Section 15.4.2 shows that VCP is in NP. We only need to reduce a known NP-complete problem to VCP. Let's show that 3-SAT $\leq_p$ VCP.

We need a polynomial-time algorithm that takes a propositional formula ϕ in 3-clausal form and builds a pair consisting of a simple graph G and a positive integer k, where ϕ is satisfiable if and only if G has a vertex cover of size at most k.

The first step is construction of G. There are three parts. Part 1 consists of a pair of vertices for each variable that occurs in ϕ, which we call a *vertex gadget*. If x is a variable, add the following, where one vertex is labeled x and the other is labeled $\overline{x}$.



Part 2 consists of three vertices for each clause of ϕ, all connected to one another and labeled by the three literals in the clause, which we call a *clause gadget*. For clause $(x \vee \overline{y} \vee z)$, we add

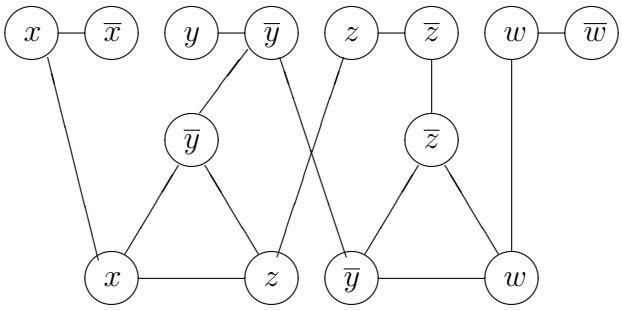


Part 3 does not add any vertices, but adds edges between part 1 vertices and part 2 vertices. Specifically, each part 2 vertex is connected to the part 1 vertex that has the same label.

Here is an example. Suppose

$$\phi = (x \vee \overline{y} \vee z) \wedge (\overline{y} \vee \overline{z} \vee w).$$

Then graph G looks like this:



That finishes the description of G. Suppose that ϕ has v variables and c clauses. Any vertex cover of G will need to have size at least $v + 2c$, one to cover each vertex gadget and two to cover each clause gadget. Let's choose $k = v + 2c$, not leaving any room for extra vertices in the vertex cover.

We need to prove that G has a vertex cover of size $v + 2c$ if and only if ϕ is satisfiable. Let's do that in two parts, proving the 'if' and the 'only if' parts separately.

Claim 1. If ϕ is satisfiable then G has a vertex cover of size $v + 2c$.

Proof of Claim 1. Suppose ϕ is satisfiable, and let a be a truth-value assignment that makes ϕ true. Here is how to select a vertex cover of G of size $v + 2c$.

(a) For each variable x, if $a(x) = \mathrm{T}$ then select the vertex gadget-vertex labeled x. Otherwise, select the vertex-gadget vertex labeled $\overline{x}$. That puts one vertex for each vertex gadget in the vertex cover, which covers the edges within vertex gadgets.

(b) For each clause $C = (\ell_1 \vee \ell_2 \vee \ell_3)$, find a literal ℓ_i that truth-value assignment a makes true. Select the clause gadget vertices that correspond to the other two literals, leaving the vertex labeled ℓ_i unselected. That covers all edges within clause gadgets.

There is no room to select any more vertices, so the part 3 edges between clause gadgets and vertex gadgets need to be covered by the vertices that have already been selected. The unselected vertex u in a clause gadget corresponds to a true literal ℓ_i (under truth-value assignment a). A part 3 edge connects u to a vertex-gadget vertex v labeled ℓ_i, and, since ℓ_i is true, vertex v was selected, and edge $\{u, v\}$ is covered by v. No more vertices need to be added.

Claim 2. If G has a vertex cover of size $v + 2c$ then ϕ is satisfiable.

Proof of Claim 2. Suppose G has a vertex cover S of size $v + 2c$. We know that S must select exactly one vertex from each vertex gadget and exactly two vertices from each clause gadget. Define truth-value assignment a so that $a(x) = \mathrm{T}$ if the vertex-gadget vertex labeled x is in S, and choose $a(x) = \mathrm{F}$ if the vertex-gadget vertex labeled $\overline{x}$ is in S.

Consider a clause gadget C. It must have one vertex u that is not in vertex cover S. Suppose u is labeled by literal ℓ. There is an edge in G between u and a vertex-gadget vertex v that is also labeled ℓ. Since S is required to cover all edges, and S does not contain u, S must contain v.

But truth-value assignment a has been defined so that literal ℓ is true; that is, if v is labeled x then $a(x) = \mathrm{T}$, and if v is labeled $\overline{x}$ then $a(x) = \mathrm{F}$, making $\overline{x}$ true. Therefore, the clause of ϕ that corresponds to clause gadget C has a true literal, namely ℓ.

The two claims show that the algorithm described above is a mapping reduction from 3-SAT to VCP. (It should be obvious that the algorithm runs in polynomial time.)

$\diamondsuit$

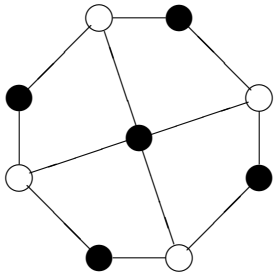


Figure 17.1 Graph G_1. The solid vertices are an independent set of G_1 and the empty circles are a vertex cover of G_1.

17.5 The Independent Set Problem

Definition 17.12. Suppose $G = (V, E)$ is a simple graph. An *independent set* of G is a set $S \subseteq V$ such that no two members of S are connected by an edge. That is, if u and v are different members of S, then$\{u, v\} \notin E$.

Definition 17.13. The *Independent Set Problem* (ISP) is the following decision problem.

Input. A simple graph $G = (V, E)$ and a positive integer k.
Question. Does G have an independent set of size at least k?

Look at graph G_1 in Figure 17.1. Some vertices are solid black and some are circles. Notice that the solid vertices are a vertex cover of G and the empty circles are an indenpendent set of G. Is that a coincidence? Think about it.

Theorem 17.14. Suppose $G = (V, E)$ is a simple graph and $S \subseteq V$. S is a vertex cover of G if and only if $\overline{S}$ is an independent set of G.

Proof. Suppose that $G = (V, E)$. Saying that S is a vertex cover of G is equivalent to the following logical statement.

$$\forall u \forall v(\{u, v\} \in E \to (u \in S \vee v \in S)).$$

Using the law of the contrapositive, that is equivalent to

$$\forall u \forall v(\neg(u \in S \vee v \in S) \to \{u, v\} \notin E).$$

Using DeMorgan's law and the definition of $\overline{S}$, that is equivalent to

$$\forall u \forall v((u \in \overline{S} \wedge v \in \overline{S}) \to \{u, v\} \notin E).$$

That is exactly what it means for $\overline{S}$ to be an independent set of G.

$\diamondsuit$

Theorem 17.15. VCP $\leq_p$ ISP.

Proof. Suppose that G has n vertices. For any set of vertices S of G, $|\overline{S}| = n - |S|$. That means $f(G,k) = (G, n-k)$ is a polynomial-time reduction from VCP to ISP, since

$$\begin{aligned} (G,k) \in \text{VCP} &\leftrightarrow G \text{ has a vertex cover of size at most } k \\ &\leftrightarrow G \text{ has an independent set of size at least } n-k \\ &\leftrightarrow (G, n-k) \in \text{ISP} \end{aligned}$$

$\diamondsuit$

Corollary 17.16. ISP is NP-complete.

Proof. It is clear that ISP $\in$ NP. Theorem 17.15 shows that known NP-complete problem VCP polynomial-time reduces to ISP.

$\diamondsuit$

17.6 The Clique Problem

Another NP-complete problem about graphs is the Clique Problem.

Definition 17.17. Suppose that $G = (V, E)$ is a simple graph. A set $S \subseteq V$ is a *clique* if every pair of vertices in S are adjacent. That is, S is a clique if for all pairs of different vertices u and v in S, $\{u, v\} \in E$.

Definition 17.18. The *Clique Problem* (CP) is the following decision problem.

Input. A simple graph G and a positive integer k.
Question. Does G have a clique of size at least k?

Definition 17.19. Suppose $G = (V, E)$ is a simple graph. Then $\overline{G} = (V, \overline{E})$ is the complement of G, formed by complementing the set of edges. That is, $\overline{G}$ has an edge between different vertices u and v if and only if G does not have an edge between u and v.

The following Theorem 17.20 is immediate from the definitions of independent sets and cliques.

Theorem 17.20. Suppose $G = (V, E)$ is a simple graph. S is an independent set of G if and only if S is a clique of $\overline{G}$.

Theorem 17.21. ISP $\leq_p$ CP

Proof. By Theorem 17.20, function $f(G, k) = (\overline{G}, k)$ is a polynomial-time reduction from ISP to CP.

♢

17.7 The Subset Sum Problem

The Subset Sum Problem is a generalization of the Partition Problem that we looked in Section 15.

Definition 17.22. The *Subset Sum Problem* (SSP) is the following decision problem.

Input. A list $x_1, \ldots, x_n$ of positive integers and a positive integer K.
Question. Does there exist an index set $I \subseteq \{1, \ldots, n\}$ so that

$$\sum_{i \in I} x_i = K?$$

We will show that SSP is NP-complete by showing that SSP is in NP and that 3-SAT $\leq_p$ SSP.

Theorem 17.23. SSP $\in$ NP.

Proof. The question in the definition of SSP is a question of existence. That suggests using I, the thing whose existence is questioned, as the evidence. Here is a polynomial-time evidence checker for SSP.

Evidence checker for SSP	
Input.	List $x_1, \ldots, x_n$ of positive integers and positive integer K
Evidence.	Index set $I \subseteq \{1, \ldots, n\}$
Requirement.	$\sum_{i \in I} x_i = K$?

$\diamondsuit$

Theorem 17.24. 3-SAT $\leq_p$ SSP.

Proof. Like the proof of Theorem 17.11, showing that 3-SAT $\leq_p$ VCP, this proof requires some thought and some gadgetry. A polynomial-time reduction from 3-SAT to SSP is a polynomial-time computable function $f(\phi) = (L, K)$ where ϕ is a propositional formula in 3-clausal form, $L = x_1$, $\ldots$, x_n is a list of positive integers and K is a positive integer so that ϕ is satisifiable if and only if $(L, K) \in$ SSP.

Writing a program for the reduction is not very informative. It is much easier to understand the reduction from an example. Suppose that ϕ is

$$(x \vee y \vee \overline{z}) \wedge (\overline{y} \vee z \vee \overline{w}) \wedge (w \vee \overline{x} \vee z)$$

with clauses c_1, c_2 and c_3. The result (L,K) of $f(\phi)$ is shown in Figure 17.2. List L is broken into two parts.

Part 1 of list L has two numbers N_x and $N_{\overline{x}}$ for each variable x. Think of those numbers written in base 10, with each number having two sections, the *variable section* and the *clause section*. The variable section has a column for each variable and the clause section has a column for each clause.

1. In the variable section, N_x and $N_{\overline{x}}$ each have a 1 in the column for x, with all other digits in the variable section being 0.

2. In the clause section, N_x has a 1 in column c_i if x occurs in clause c_i, and it has a 0 in column c_i otherwise. Similarly, $N_{\overline{x}}$ has a 1 in column c_i if $\overline{x}$ occurs in clause c_i, and a 0 in column c_i otherwise.

Part 2 of list L has two numbers $P_{i,1}$ and $P_{i,2}$ for each clause c_i, which both contain only a 1 in the column that corresponds to c_i, as shown in Figure 17.2. They are used as *padding*.

We need to show that ϕ is satisfiable if and only if there is a way to select numbers from list L whose sum is exactly K. As before, we prove the 'if' part and the 'only if' part separately.

Claim 1. If ϕ is satisfiable then there is a way to select numbers from list L whose sum is K.

	w	z	y	x	c_1	c_2	c_3
N_x:				1	1	0	0
$N_{\overline{x}}$:				1	0	0	1
N_y:			1	0	1	0	0
$N_{\overline{y}}$:			1	0	0	1	0
N_z:		1	0	0	0	1	1
$N_{\overline{z}}$:		1	0	0	1	0	0
N_w:	1	0	0	0	0	0	1
$N_{\overline{w}}$:	1	0	0	0	0	1	0
$P_{1,1}$:					1	0	0
$P_{1,2}$:					1	0	0
$P_{2,1}$:					0	1	0
$P_{2,2}$:					0	1	0
$P_{3,1}$:					0	0	1
$P_{3,2}$:					0	0	1
K:	1	1	1	1	3	3	3

Figure 17.2. List L consists of N_x and $N_{\overline{x}}$ for each variable x plus $P_{i,1}$ and $P_{i,2}$ for each clause c_i. Numbers are written in base 10. Notice that the sum can never involve a carry since there are no more than five 1s in any column.

Proof of Claim 1. Suppose that a is a truth-value assignment that makes ϕ true. It tells which numbers to select to make a sum of K. First, select a true literal from each clause. If literal x is selected, put N_x into the list of selected numbers. If literal $\overline{x}$ is selected, put $N_{\overline{x}}$ into the list of selected numbers. If neither x nor $\overline{x}$ is selected, it does not matter; put N_x into the list of selected numbers.

Notice that the sum of the selected numbers has exactly one 1 in each column of the variable section, so the variable section of the sum K is correct.

Now we need to make sure the section of K consisting of 3's is correct. Because each clause contains a true literal, there must be at least one 1 in each clause column. But the total number of 1s in a single clause column in part 1 can be at most 3 since each clause contains 3 literals. If a clause column has one 1, then select both of the padding (part 2) numbers for that column to make a total of exactly 3. If there are two 1's, select one of the padding numbers. If there are three 1's, do not select any of the padding numbers for that clause.

The sum of the selected numbers is exactly K.

Claim 2. If it is possible to select numbers from list L whose sum is K then ϕ is satisfiable.

Proof of Claim 2. Because each variable column must sum to 1, exactly one of N_x and $N_{\overline{x}}$ must have been chosen for each variable x. Define truth-value assignment a so that $a(x) = \mathrm{T}$ if N_x is selected and $a(x) = \mathrm{F}$ if $N_{\overline{x}}$ is selected.

The selected numbers must sum to 3 in each clause column. At most two 1s in column i can come from padding numbers. The third must come from N_x, where x occurs in clause c_i, or from $N_{\overline{x}}$ where $\overline{x}$ occurs in clause c_i. That means c_i contains a true literal under truth-value assignment a.

The two claims show that the algorithm described above is a mapping reduction from 3-SAT to SSP. It should be obvious that the algorithm runs in polynomial time.

$\diamondsuit$

17.8 Graph Coloring Problems

Let's look at some known NP-complete problems without proving them NP-complete.

Definition 17.25. Suppose that G is a simple graph and k is a positive integer. Say that G is *k-colorable* if it is possible to color each vertex of G with one of k colors so that no two adjacent vertices have the same color.

Definition 17.26. The *Graph Coloring Problem* is the following decision problem.

Input. A simple graph G and a positive integer k.
Question. Is G k-colorable?

The Graph Coloring Problem is clearly in NP. The question asks whether *there exists* a way to color the vertices of G so that no two adjacent vertices have the same color. The obvious evidence to request is the coloring.

Evidence checker for Graph Coloring	
Input	Simple graph G and positive integer k.
Evidence	Assignment A of one of k colors to each vertex of G.
Requirement	Every edge of G connects two vertices that are assigned different colors in color assignment A.

If you try to color some graphs by hand, you can get an idea of how difficult Graph Coloring can be. The Graph Coloring Problem is known to be NP-complete. In fact, it is NP-complete even if k is fixed at 3.

Definition 17.27. The *3-Coloring Problem* is the following decision problem.

Input. A simple graph G.
Question. Is G 3-colorable?

Graph Coloring is so difficult, it can even be restricted further and remain NP-complete. A graph is *planar* if it can be drawn in the plane (on a piece of paper, if you like) so that no two edges cross one another.

Definition 17.28. The *Planar 3-Coloring Problem* is the following decision problem.

Input. A planar simple graph G.
Question. Is G 3-colorable?

The Planar 3-Coloring Problem is NP-complete. But that does not mean that all graph coloring problems are NP-complete. For example, 2-coloring is easy. (Try an example.) Also, if G is known to be a planar graph, then 4-coloring is trivial: the answer is always yes, by the following.

Theorem 17.29. (The 4-Color Theorem) Every planar graph is 4-colorable.

17.9 Hamilton Cycles and Hamilton Paths

Definition 17.30. Suppose that G is a simple graph. A *simple cycle* in G is a cycle that does not contain any vertex more than once. A *Hamilton Cycle* is a simple cycle that contains every vertex.

Not every graph has a Hamilton cycle. You should be able to find a graph that has a Hamilton cycle and another that does not.

Definition 17.31. The *Hamilton Cycle Problem* is the following decision problem.

Input. A simple graph G.
Question. Does G have a Hamilton cycle?

Imagine that G has been drawn on paper (possibly with edges crossing). The Hamilton Cycle Problem asks whether it is possible to draw a cycle, following the edges, that hits every vertex exactly once, without lifting your pencil off the paper.

It is easy to show that the Hamilton Cycle Problem is in NP. The obvious evidence is a Hamilton cycle.

Evidence checker for Hamilton Cycle	
Input	Simple graph G with n vertices.
Evidence	Sequence $v_1, \ldots, v_n$ of vertices of G.
Requirement	$v_1, \ldots, v_{n-1}$ contains every vertex exactly once, $v_1 = v_n$ and for $i = 1, \ldots, n-1$, $\{v_i, v_{i+1}\}$ is an edge of G.

The Hamilton Cycle Problem is known to be NP-complete. A related problem, also NP-complete, is the Hamilton Path Problem.

Definition 17.32. Suppose that G is a simple graph. A *simple path* in G is a path that does not contain any vertex more than once. A *Hamilton Path* is a simple path that contains every vertex (exactly once).

Definition 17.33. The *Hamilton Path Problem* is the following decision problem.

Input. A simple graph G.
Question. Does G have a Hamilton path?

17.9.1 Euler Cycles

A problem that is at least superficially related to the Hamilton Cycle Problem is the Euler Cycle Problem. (Leonard Euler's last name is pronounced "Oiler".)

Definition 17.34. Suppose that G is a simple graph. An *Euler Cycle* in G is a cycle that uses each *edge* exactly once. (The cycle can contain a particular *vertex* several times.)

Not every graph has an Euler cycle. You should be able to find a graph that has an Euler cycle and another that does not.

Definition 17.35. The *Euler Cycle Problem* is the following decision problem.

Input. A simple graph G.
Question. Does G have an Euler cycle?

How difficult is it to determine whether a graph contains an Euler cycle? It is easy to see that the Euler Cycle Problem is in NP.

Evidence checker for Euler Cycle	
Input	Simple graph G with n vertices.
Evidence	Sequence $v_1, \ldots, v_n$ of vertices of G.
Requirement	$v_1 = v_n$, for $i = 1, \ldots, n-1$, $\{v_i, v_{i+1}\}$ is an edge of G (so $v_1, \ldots, v_n$ is a cycle) and cycle $v_1, \ldots, v_n$ uses each edge exactly once.

So we have an *upper bound* on the difficulty of solving the Euler Cycle Problem: it is in NP, so it is, to within a polynomial, no worse than SAT. But that is not a *lower bound*. It might be that the Euler Cycle Problem is easy to solve.

And in fact, it is! Graph G has an Euler cycle if and only if every vertex has even degree. That is easy to check. Not only is the Euler Cycle Problem in P, but it is solvable in time $O(n)$.

There is a lesson in that. You cannot inspect a problem and conclude, based on its similarity to another problem, that it is an easy or a difficult problem. To show that a problem is in P, find a polynomial-time algorithm for it, and *make sure that the algorithm works.* To show that a problem is NP-complete, show that it is in NP and that a known NP-complete problem reduces to it in polynomial time. There are no shortcuts.

prev **next**