Notes on **NP** Completeness

Rich Schwartz

November 10, 2013

1 Overview

Here are some notes which I wrote to try to understand what **NP** completeness means. Most of these notes are taken from Appendix B in Douglas West's graph theory book, and also from wikipedia. There's nothing remotely original about these notes. I just wanted all this material to filter through my brain and onto paper. I also wanted to collect everything together in a way I like. Here's what these notes do.

- Define the class **P**, using deterministic Turing machines.
- Define the class **NP** and the notion of **NP** completeness, using nondeterministic Turing machines.
- Prove the Cook-Levin Theorem: SAT is **NP** complete.
- Prove that 3-SAT is **NP** complete.
- Prove that graph 3-colorability is **NP** complete.
- Prove that the problem of finding a maximal independent set, or a minimal vertex covering, in a graph is **NP** complete.
- Prove that finding a directed or undirected Hamiltonian path or cycle in a directed or undirected graph is **NP** complete.
- Prove that the traveling salesman problem is **NP** complete.
- Prove that the problem of finding an interior lattice point in an integer polytope is **NP** complete.

2 The Class P

2.1 Deterministic Turing Machines

Informally, a deterministic Turing Machine (DTM) is a finite machine which hovers over an infinite tape, writing and erasing symbols on the tape according to a combination of what is written on the tape and its own internal state. This idealizes how we might do a calculation. At any given step we'll modify what we have written according to a combination of what is underneath our pen and what we are thinking.

Formally, the *tape* is a bi-infinite union of squares, arranged along (say) the x-axis. One symbol is to be written in each square, and the machine, at any given time, is focused (or hovers) on one of the squares. The Turing machine itself is a quintuple (A, S, S_0, F, T), where

- A is a finite alphabet, with some distinguished blank state.
- S is a finite set of internal states of the machine.
- $S_0 \in S$ is the initial state of the machine.
- $F \subset S$ is a finite subset of "ending" or "accepting" states.
- $T : A \times S \to A \times S \times \{-1, 1\}$ is the transition function.

If the machine finds itself hovering over a letter a and in a state s, then it computes $T(a, s) = (a', s', \pm 1)$. It replaces a with a', changes to state s', and moves one unit left or right depending on the sign of ± 1.

An *input* (or *problem*) for the machine is a finite string written on the paper, and a positioning of the head of the machine – that is, where it is initially hovering. Since the tape is infinite, the finite string is padded out with blanks so that every square on the bi-infinite tape has a symbol.

2.2 Polytime Decision Problems

A *decision problem* is simply a problem which says "yes" or "no" for an infinite family of inputs. The problem is *Turing computable* if there is a pair of DTMs with the following property. For each input, one or the other of the machines arrives at an accept state in finite time. In the first case, the answer to the problem is "yes", and in the second case the answer is "no".

One could also formulate this in terms of a single Turing machine, in which the accept states are divided into "yes" states and "no" states. Here are some examples of computable decision problems.

- Does a graph have an Eulerian circuit?
- Does a graph have a Hamiltonian cycle?
- Can a graph be embedded in the plane.
- Can one 3-color a graph.

In each instance of one of these problems, one takes a specific graph and inputs it into the DTMs, and then lets the computation run. Typically, one would encode the graph as an incidence matrix, and then string out the entries of the matrix so that they fit on the tape. More informally, one performs some finite algorithm on the graph which leads to an answers.

The decision problem is called *polytime* if there is a polynomial P such that both DTMs run in at most $P(n)$ steps, given a problem encoded with a length n finite string. Informally, n denotes the "size" of the instance of the problem. Since the definition is insensitive to the details of P, it often suffices to have a very rough description of "size". For intsance, for graphs, the size could be either the number of edges or the number of vertices, and the same problems would be counted as polytime.

The set $\mathbf{P}$ denotes the set of polytime decision problems. The decision problem for Eulerian circuits is in $\mathbf{P}$ because one just checks whether or not all vertices in a given graph have even degree. Such a calculation is certainly a polynomial in the size of the graph.

The decision problem for Hamiltonian cycles seems not to be in $\mathbf{P}$. One way to phrase the famous $\mathbf{P} \neq \mathbf{NP}$ conjecture is simply that *The Hamiltonian cycle problem is not in* $\mathbf{P}$. However, this way of phrasing it ignores many details and makes the $\mathbf{P} \neq \mathbf{NP}$ problem seem kind of random.

3 The Class NP

3.1 Non-Deterministic Turing Machines

Given a set Y, let 2^Y denote the set of subsets of Y. Say that a *set function* from X to Y is a map $f : X \to 2^Y$. A set function from X to Y is a special

case of a relation between X and Y. The set up for a non-deterministic Turing machine (NTM) is exactly the same as for a DTM, except for one point. The map

$$T : A \times S \to A \times S \times \{0, 1\},$$

is replaced by a set map from $A \times S$ into $A \times S \times \{0, 1\}$. We still denote this set map by T.

One can think of a NTM in a deterministic way: Given an input to the machine, the machine makes a tree of calculations. At every step of the calculation, the machine makes *all* transitions allowed by the set map T. The calculation is a success if one of the branches of the tree ends up in F, the set of accept states. One could build an auxillary DTM which makes a breadth-first-search through the calculation tree produced by the NTM. However, for an infinite (growing) family of inputs, the NTM might stop in polytime whereas the DTM might require exponential time.

One way to think of a successful calculation on an NTM is that some oracle feeds the machine a list of choices to make at each stage – or maybe the machine just gets lucky – and then the machine is capable of following these steps and getting to an accept state.

3.2 Non-Deterministic Polytime Problems

A decision problem is said to be a *nondeterministic polytime problem* if there is an NTM, Ω, and a polynomial P with the following properties. If an instance of the problem has size n, then the answer to the decision problem is "yes" if and only if Ω lands in an accept state after at most $P(n)$ steps. We say that the machine is *allowed to run* for $P(n)$ steps on an input of size n.

The Hamiltonian circuit decision problem is a classic problem in **NP**. We can make an NTM which tries every path in a graph of length at most n^2. The machine then rejects paths as soon as the revisit a vertex without having hit all the vertices.

The set **NP** denotes the set of nondeterministic polytime decision problems. The famous $\mathbf{P} \neq \mathbf{NP}$ conjecture is that there exists a problem in **NP** that does not belong to **P**.

3.3 Reduction

Suppose that X and Y stand for two decision problems. A *reduction* from X to Y is a two-part algorithm. The first part of the algorithm converts any given instance x of X into an instance of y of Y such that that

- The number of steps of the conversion is bounded by a polynomial function of x.
- The size of y is bounded by a polynomial function of x.

The second part of the algorithm converts any given solution of y to a solution of x in such a way that the number of steps in the conversion is bounded by a polynomial in the size of y. By *algorithm*, we mean a pair of auxilliary DTMs which do the jobs.

In case there is a reduction from X to Y, we'll write $X \to Y$. Essentially a reduction from X to Y is a procedure whereby one can use a solution to Y to create an essentially equally efficient solution for X. For instance, if $Y \in \mathbf{P}$ and $X \to Y$, then $X \in \mathbf{P}$ as well.

A decision problem is called **NP** *hard* if every problem in **NP** reduces to it. An **NP** hard problem may or may not belong to **NP**. The decision problem is called **NP**-*complete* if it is **NP**-hard and it belongs to **NP**. It is a nontrivial result that **NP** complete problems actually exist.

4 Satisfiability (SAT)

4.1 The Cook-Levin Theorem

In particular, Here is one more example of a problem in **NP**. Let $\wedge$ stand for "or". A *simple clause* is an expression of the form $\pm a_1 \wedge ... \wedge \pm a_n$, where a_i is a boolean variable and $\overline{a}_i$ is its negation. That is a_i is true if and only if $\overline{a}_i$ is false. The clause is true if and only if at least one of the variables is true. For instance $\overline{a}_1 \wedge a_2$ is true if and only if (non-exclusively) either a_1 is false or a_2 is true.

A *compound clause* is an expression of the form $C_1 \& ... \& C_k$, where each C_i is a simple clause. The compound clause is true if and only every one of the component simple clauses is true. The *satisfiability problem* (SAT) asks the following: Given a compound clause, is there a truth-assignment for the variables which makes the whole thing true? One certainly make

a DTM which checks in polytime whether or not a given truth-assignment works for a given clause. One can then build a NTM whose first step is to write down all possible truth-assignments, and whose remaining steps follow the deterministic calculation for each truth-assignment. Hence, SAT is in **NP**.

Theorem 4.1 (Cook-Levin) *Any decision problem in* **NP** *reduces to SAT. Hence SAT is* **NP***-complete.*

Proof: Suppose that: $X \in \mathbf{NP}$ and $\Omega = (A, S, S_0, F, T)$ is a nondeterministic Turing machine solving X. Let x be an instance of X, with input size n. Let $N = p(n)$ be a bound on the length Ω is allowed to run on N. The convention is that, if some computational branch of Ω reaches an accept state before N steps, it just stays there until N steps are reached.

We want to convert x into an instance of SAT. The basic idea is to encode the entire working of Ω on x into a compount clause which is satisfied if and only if the Ω reaches an accept state. We introduce the following variables.

- A_{ijk}. This variable is true iff letter i is written in square j at step k of the calculation.
- H_{ik}. This variable is true iff Ω is hovering over square i at step k of the calculation.
- S_{ik}. This variable is true iff Ω is in state i at step k of the calculation.

We think of a variable assignment as specifying a single branch of the computation tree. These clauses enforce the initial set-up.

- S_{00}. This means that Ω starts in state S_0.
- H_{00}. This means that Ω initially hovers over square 0.
- $S_{i,x(i),0}$. Here $x(i)$ denotes the letter of the input written in square i. Here $|i| \leq p(n)$. The truth of these variables encodes the fact that x is the input to Ω.

These clauses enforce the basic properties of any Turing machine.

- $\overline{S}_{ijk} \wedge \overline{S}_{i'jk}$. This means that the two letters S_i and $S_{i'}$ cannot both be written in square j at step k. In short, there is only one letter per square.

- $\overline{H}_{ik} \wedge \overline{H}_{i'k}$. This means that Ω cannot hover at 2 distinct squares.
- $\overline{S}_{ik} \wedge \overline{S}_{i'k}$. This means that Ω cannot be in 2 states at once.
- $(A_{ijk} \& \overline{A}_{ij,k+1}) \Rightarrow H_{ik}$. This means that the tape can only change in a spot where Ω is focused. This expression is a compound clause because $(a \& b) \Rightarrow c$ is the same as $(\overline{a} \wedge c) \& (\overline{b} \wedge c)$.
- $(A_{ijk} \& H_{jk} \& S_{\ell k}) \Rightarrow \bigvee (A_{i',j',k+1} \& H_{j',k+1} \& S_{\ell',k+1})$. The expression on the right is taken over all triples related to triple on the left by the set function T. Together with the other clauses, this enforces the condition that Ω makes a legal transition at each step. This expression is a compound clause because (setting $B = \bigvee b_i$), the expression $(a_1 \& a_2 \& a_3) \rightarrow B$ is equivalent to $\overline{a}_1 \wedge \overline{a}_2 \wedge \overline{a}_2 \wedge B$.

This last clause enforces the condition that the computation arrives in an accept state.

- $F_{1N} \wedge ... \wedge F_{kN}$, where $F_1, ..., F_k$ are the accept states. Here $N = p(n)$.

If some computational branch of Ω reaches an accept state in at most N steps, then we can assign truth values to the variables so that the compound clause is satisfied. Conversely, if we have a truth-assignment which satisfies the compound clause, we simply use it as a guide for running a branch of Ω. The conversions back and forth involve only $q(n)$ steps, where q is another polynomial in n which depends only on p. This completes the proof. ♠

4.2 Reduction to 3-SAT

Say that a 3-*clause* is an compound clause of the form

$$(a_1 \wedge b_1 \wedge c_1) \& (a_2 \wedge b_2 \wedge c_2) \& ... (a_n \wedge b_n \wedge c_n).$$

3 SAT is the decision problem which asks whether or not the truth values of variables can be set in order to satisfies a given 3-clause. I believe that this theorem is due to Karp. In fact, all the reductions I'm going to explain below are due to Karp.

Theorem 4.2 *3-SAT is* **NP complete.**

Proof: Since SAT is in **NP**, and 3-SAT is a sub-problem of SAT, we know that 3-SAT is in **NP**. To finish the proof, we just have to show that SAT reduces to 3-SAT.

The first step is to observe that

$$(a \wedge b) \Leftrightarrow c$$

is logically equivalent to the 3-clause

$$(\overline{a} \wedge c \wedge c)\&(\overline{b} \wedge c \wedge c)\&(a \wedge b \wedge \overline{c}).$$

Now, observe that the simple clause $a_1 \wedge ... \wedge a_n$ is equivalent to the clause

$$(b \wedge a_3 \wedge ... \wedge a_n)\&\Big((a_1 \wedge a_2) \Leftrightarrow b\Big).$$

Here b is some entirely new variable. The length of the original clause is n and the maximum length of the simple clauses in the replacement is $\max(3, n-1)$. Continuing to make substitutions like this, we produce an equivalent clause in which the maximum length is 3. ♠

5 Graph Colorings

5.1 Basic Definitions

A *proper coloring* of a graph is a color-assignment of the vertices of the graph, so that vertices incident to a common edge have different colors. A graph is *k-colorable* if it has a coloring with k colors. The *k-coloring* decision problem asks, for each graph, G, is G k-colorable. Call this problem k-COLOR.

Lemma 5.1 *2-COLOR is in* **P**.

Proof: One can check whether a graph is 2-colorable by setting some initial node equal to 0 and then doing a breadth-first-search. The color of each new vertex encountered is forced by the color of previous vertices. The graph has a proper 2-coloring if and only if the search ends with no contradictions. This is a polytime algorithm. ♠

We will see that the remainder of the problems are **NP** complete.

5.2 A Particular Graph

This construction is taken straight from West's book. Let G be the graph on the left hand side of Figure 1. We call the 3 leftmost vertices of G the *inputs* and the rightmost vertices of G the *output*. Given $a, b, c \in \{0, 1, 2\}$ let $C(a, b, c)$ denote the set of proper 3-colorings of the G which assign values a, b, c (top to bottom, say) to the inputs. Here are two easily checked facts about G.

- Any coloring in $C(0, 0, 0)$ assigns 0 to the output.
- If $a+b+c \neq 0$ then there exists some coloring in $C(a, b, c)$ which assigns a nonzero value to the output.

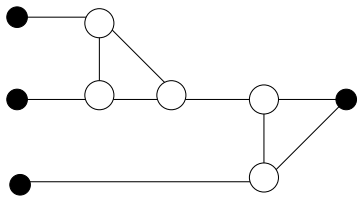

Figure 1: The Graph G_0 with 3 inputs and 1 output.

5.3 3-COLOR

One can easily check in polytime that a given 3-coloring of a graph is proper. Similar to the situation for 3-SAT, this implies that 3-COLOR is in **NP**.

Theorem 5.2 (Karp) 3-*COLOR is* **NP** *complete.*

Proof: Since 3-SAT is **NP** complete, it suffices to reduce 3-SAT to 3-COLOR. Let $C = C_1\&...\&C_n$ be a 3-compound clause. Suppose that C involves the variables $b_1, ..., b_k$. We build a graph G_C as follows.

- G_C has an initial vertex a.
- G_C has vertices $b_1, \overline{b}_1, ..., b_k, \overline{b}_k$. We add to G_C all the triangles $ab_i\overline{b}_i$.
- G_C has vertices $c_1, ..., c_n$. For the ith clause C_i, we insert a copy of G_0, whose inputs are the 3 b (or $\overline{b}$) variables of C_i and whose output is c_i.
- G_C has a final vertex d which is connected to every c_i by an edge. d also connects to a.

The construction of G_C from C takes polynomial time and creates a graph whose size is a polynomial function of the size of C. We claim that G_C is 3-colorable if and only if C is satisfiable.

Suppose first that C is satisfiable. We assume in our construction that the b-variables are given the truth values which satisfy C. Then we color G_C as follows.

- $\gamma(a) = 2$. Here $\gamma(a)$ is the color of a.
- $\gamma(b_i) = 0$ or $\gamma(b_i) = 1$ depending on whether b_i is true or false. Also $\gamma(b_i) + \gamma(\bar{b}_i) = 1$. Thus, each triangle $ab_i\bar{b}_i$ is properly colored.
- From the fact that C is satisfied, and the properties of G_0, we see that it is possible to arrange that $\gamma(c_j) \in \{1, 2\}$ for all j.
- We set $\gamma(d) = 0$.

This is a proper coloring.

Conversely, suppose that G_C has a proper coloring. By changing the names of the colors, if necessary, we arrange that $\gamma(a) = 2$ and $\gamma(d) = 0$. Then $\gamma(b_i) \in \{0, 1\}$ and $\gamma(\bar{b}_i) + \gamma(b_i) = 1$ because each triangle $ab_i\bar{b}_i$ is properly colored. We assign the truth values to the b variables using the coloring. Since $\gamma(d) = 0$, we must have $\gamma(c_i) \neq 0$ for all i. But then it never happens that the inputs to the copy of G_0 ending at c_i are all 0. Hence each clause C_i is satisfied. Hence C is satisfied. The conversion from the coloring to the truth-assignment takes linear time. ♠

6 Independent Sets and Vertex Covers

6.1 Independent Sets

Let G be a graph. An *independent set* of size k, or a *k-independent set*, is a collection of k vertices of G, no two are incident to a common edge. One can fix k and ask the question: Does G have a k-independent set? This decision problem is in **P**: If G has n vertices then one simply checks the independence of each of the n choose k possibilities. There are $O(n^k)$ possibilities, and the check is also polytime.

A more difficult decision problem asks: Given the pair (G, k), does there exist a k-independent set? This time the size of k is allowed to grow, and is taken as part of the input. Call this problem INDEP. Note that there is only one INDEP: it does not have a parameter like the coloring problems. An argument like the ones above shows that INDEP is in **NP**. In this section, we'll show that INDEP is **NP** complete.

Lemma 6.1 *Let n be the number of vertices of G. Let K_k be the complete graph on k vertices. Then G has a proper k-coloring set if and only if $G \times K_k$ (the graph product) has an n-independent set.*

Proof: Suppose that G has a proper k-coloring. Let

$$S_G = \bigcup_{v \in V(G)} (v, \gamma(v)).$$

The union takes place over all vertices of G. Certainly S_G has cardinality n. To show that S_G is independent, suppos that $(v, \gamma(v))$ and $(w, \gamma(w))$ share an edge. Then either $v = w$ or $\gamma(v) = \gamma(w)$. The first case is not possible. In the second case, when $\gamma(v) = \gamma(w)$, there must be an edge between v and w. But this does not happen. Hence S_G is an n-independent set.

Conversely, suppose that S is an n-independent set. Consider two elements of S, namely (v, i) and (w, j). If cannot happen that $v = w$ because then i and j (like all vertices of K_k) share an edge. By the pidgeonhole principle, each vertex of G appears once in the first coordinate of S. We color the vertices of G according to the value of the second coordinate. The independence of S guarantees that adjacent vertices get different colors. For later reference, call this construction $(*)$. ♠

Theorem 6.2 (Karp) *INDEP is* **NP** *complete.*

Proof: It suffices to reduce 3-COLOR to INDEP. To prove that G has a proper 3-coloring, it suffices to check that $G \times K_3$ has an n-independent set. The size of the pair $(G \times K_3, n)$ is a polynomial in n, and the conversion takes a polynomial number of steps. If some NTM shows that $G \times K_3$ has an independent set, when we just trace through the steps of $(*)$ to get our proper 3-coloring. ♠

6.2 Vertex Covers

A *k-covering* of a graph G is a collection of k vertices such that every edge in G is incident to a vertex in the cover.

Lemma 6.3 *Let G be a graph with n vertices. Let M be the size of the maximum independent set and let m be the size of the minimal covering. Then $M + m = n$.*

Proof: Suppose that S is a minimal cover. Consider the complement $S' = V(G) - S$. If two vertices in S' share an edge in G not incident to a vertex of S. Hence S' is independent. If S is an independent set, then $S' = V(G) - S$ is a vertex cover. Otherwise some edge is not incident to a vertex in S' and hence has both ends in S. ♠

The problem COVER asks: Given a pair (G, k) does there exist a vertex cover of G having size k. Arguments like the ones above show that COVER is in **NP**. Using Lemma 6.3 one immediately reduces INDEP to COVER. Hence COVER is **NP** complete.

7 Hamiltonian Paths and Cycles

7.1 The Key Construction

Figure 2 shows a particular directed graph Q. We think of Q as extending the union of the two directed horizontal edges by adding in 4 more edges, making a kind of square.

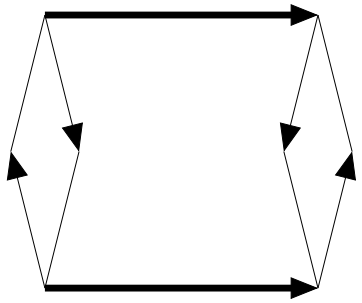

Figure 2: The graph Q.

Let G be a graph and let k be an integer. We're going to construct a directed graph $H = H(G, k)$ such that H has a directed Hamiltonian path from a vertex w_0 to a vertex w_k if and only if G has a k-vertex covering.

Here is the construction. Say that a *flag* of G is a pair (v, e) where v is a vertex of G and e is an edge of G incident to v. Let $v_1, ..., v_n$ be the vertices of G. Let d_i be the degree of v_i. Let $f_{i1}, f_{i3}, f_{i5}, ...$ denote the flags having point v_i. The ordering doesn't matter.

- Add vertices $w_0, ..., w_k$. We call these the *anchors*.
- Add directed paths $P_1, ..., P_n$ of length $2d_1, ..., 2d_n$.
- Add directed edges from w_i to the start of P_j for all $i = 0, ..., k-1$ and all $j = 1, ..., n$.
- Add directed edges from the end of P_j to w_i for all $i = 1, ..., k$ and all $j = 1, ..., n$.
- Let $P_{i1}, P_{i3}, ...$ be the odd consecutive edges of P_i. There are d_i such edges. We extend $P_{ab} \cup P_{cd}$ by a copy of Q whenever f_{ab} and f_{cd} are flags with a common edge.

Lemma 7.1 *If G has a k-vertex cover, then H has a directed Hamiltonian path connecting w_0 to w_k.*

Proof: The idea is to start with a path from w_0 to w_k and then improve it until it is Hamiltonian. We can re-index the vertices so that $v_1, ..., v_k$ of G comprise the vertex cover. Let Γ be the path which starts at w_0, then takes P_1, then moves to w_1, then takes P_2, and so on, until reaching v_k.

Let P_{cd} be any of the odd segments with $c > k$. There is a unique flag f_{ab}, with $a \leq k$ and b odd, so that f_{ab} and f_{cd} share an edge. We modify Γ to that, instead of directly taking S_{ab}, we go the long way around the copy of Q joining S_{ab} to S_{cd}. The longer graph now hits the endpoints of S_{cd}. Doing this for every such S_{cd}, we get out Hamiltonian path. ♠

We'll establish the converse to this result in several steps. We will suppose that Γ is a directed Hamiltonian path of H connecting w_0 to w_k. Our construction above works with the odd edges $P_{i1}, P_{i3}, ...$ Now we work with the even edges $P_{i2}, P_{i4}, ...$ In the next lemma, P_{i0} will stand for an edge

connecting an anchor to the start of P_i. Likewise $P_{i,2d_i}$ will stand for an edge connecting the end of P_i to an anchor. Again, If Γ contains such edges, then they are unique.

Lemma 7.2 *Suppose that Γ contains one of $P_{i,j-1}$ and $P_{i,j+1}$. Then Γ contains both of them.*

Proof: Without loss of generality we may take $i = 1$. For ease of notation, we take $j = 2$, and we will show that $P_{12} \subset \Gamma$ implies that $P_{14} \subset \Gamma$. This argument does not cover the general case, but the general case uses essentially the same reasoning.

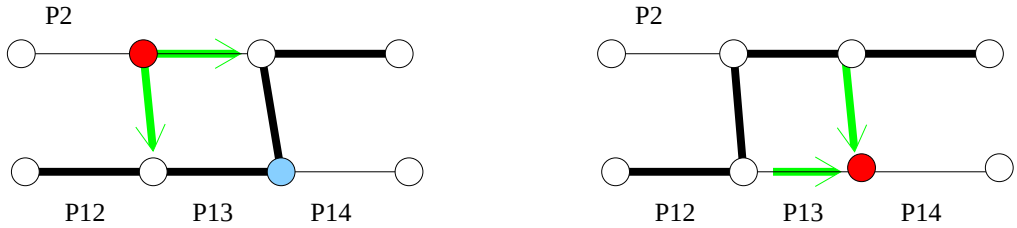

Figure 4: Two cases

There are two cases. Suppose first that $P_{13} \subset \Gamma$. If $P_{14} \not\subset \Gamma$, then Γ must take the path shown on the left side of Figure 4. Since Γ is Hamiltonian, Γ must contain the red vertex. However, there are only two outgoing edges from the red vertex, and these both crash back into Γ.

Suppose that $P_{13} \not\subset \Gamma$. Then Γ takes the path on the right hand side of Figure 4. Since Γ is Hamiltonian, Γ must contain the red dot. But then the two ingoing edges to the red dot emanate from points already on Γ. ♠

Corollary 7.3 *Suppose that any of the following is true:*

- Γ *contains an even edge of P_i.*
- Γ *contains an edge joining an anchor to the start of P_i.*
- Γ *contains an edge joining the end of P_i to an anchor.*

All of the above is true and Γ contains every even edge of P_i.

Corollary 7.4 *G has a vertex cover of size k.*

Proof: Let S denote the set of vertices v_i of G such that Γ contains an even edge of P_i. Suppose S has size at least $k+1$. By Corollary 7.3, Γ contains $k+1$ edges leaving anchors. But there are only k such edges. In fact, S has size exactly k, because it happens exactly k times that Γ leaves an anchor.

Let e be some edge of Γ. We want to show that one endpoint of e lies in S. Suppose this is not the case. Let v_i and v_j be the endpoints of e. Let p_i and p_j be the start points of P_i and P_j respectively. Γ does not contain edges joining anchors to p_i or p_j. Hence, Γ must join p_i to p_j. But then Γ contains P_{i1} and P_{j1}. By Corollary 7.3, Γ contains neither P_{i2} nor P_{j2}. But then Γ joins the starting points of P_{i2} and P_{j2}. This shows that Γ has a 4-cycle, a contradiction. The contradiction shows that S is a vertex cover. ♠

7.2 Directed Hamiltonian Path

Let G be a directed graph, and let v, w be two vertices of G. The decision problem DHP takes as input (G, v, w) and asks if there is a directed Hamiltonian path starting at v and ending at w. Arguments like those above show that DHP is in **NP**.

Theorem 7.5 (Karp) *DHP is* **NP** *complete.*

Proof: The proof amounts to reducing COVER to DHP. Let (G, k) be an input problem for COVER. The construction in the previous section produces a triple (H, w_0, w_k) such that the answer to DHP for (H, w_0, w_k) is yes if and only if the answer to COVER for (G, k) is yes. The size of H is polynomial in the size of G. Moreover, there a polytime algorithm to convert the directed Hamiltonian path in (H, w_0, w_k) to the desired covering of G. ♠

7.3 Hamiltonian Paths and Cycles

Let HP be the vertion of DHP but for undirected graphs. Arguments like those above show that HP is in **NP**.

Theorem 7.6 (Karp) *HP is* **NP** *complete.*

Proof: The idea is to reduce DHP to HP. Let G be a directed graph. We form a graph H as follows. For each vertex a of G we produce a 3-path $a_{-1} - a_0 - a_1$. We then join a_1 to b_{-1} whenever a directed edge of H joins a to b.

If we want to solve DHP for (G, v, w) we solve HP for (H, v_{-1}, w_1). If this latter problem has a solution, then the Hamiltonian path Γ on H has the following properties.

- Since Γ contains v_0, the first edge of Γ connects v_{-1} to v_0.
- The second edge of Γ must connect v_0 to v_1.
- In general, and by induction, once Γ reaches some vertex a_1, the next vertex reached by Γ must be some b_{-1}.
- The last edge of Γ joins w_0 to w_1.

If we just list out the vertices $v_0, ..., a_0, b_0, ..., w_0$ hit by Γ we produce vertices of G which are joined by a Hamiltonian path from v to w. ♠

Finally, we arrive back to the Hamiltonian cycle problem, which we discussed in connection with a naive formulation of the $\boldsymbol{P \neq NP}$ problem. Let HC denote the decision problem which asks, does a graph G have a Hamiltonian cycle. Arguments like those above show that HC is in **NP**.

Theorem 7.7 (Karp) *HC is* **NP** *complete.*

Proof: The idea is to reduce HP to HC. Suppose we want to solve HP for some triple (G, v, w). We form a new graph H simply by adding an extra vertex x and joining it to both v and w. If we can solve HC on H then the resulting Hamiltonian cycle Γ must visit x and hence must contain, consecutively, the edges xv and xw. We just omit these two edges, to get a Hamiltonian path on G which connects v to w. ♠

8 The Traveling Salesman Problem

Let K_n be the complete graph on n vertices. Suppose that each edge of K_n is given a non-negative integer weight. The traditional *traveling salesman*

problem asks for the hamiltonian cycle of K_n having minimum total weight. The decision problem TSP starts with a pair (G, d), where G is a weighted copy of K_n and d is some integer. The problem asks: Does G have a Hamilonian cycle of weight at most d? Arguments like those above shows that TSP is in **NP**.

Theorem 8.1 (Karp) *TSP is* **NP** *complete.*

Proof: The idea is to reduce HC to TSP. Suppose that G is a graph. We let H be a copy of K_n. Note that H has less than n^2 edges. If an edge of H comes from an edge of G, we give it weight 1. Otherwise, we give the edge weight n^2. Note that H has a Hamiltonian cycle of weight at most n^2 if and only if G has a Hamiltonian cycle. So, if we can solve TSP for (H, n^2), then we can solve HC for G. ♠

The original traveling salesman problem is at least as hard to solve as TSP, because a solution to the original problem for G immediately solves TSP for any pair (G, d). When G has integer weights whose sizes are at most polynomial in the size of G, one can solve the original problem by solving TSP a polynomial number of times.

9 Interior Lattice Point

Say that an *integer polytope* is a compact convex polytope $P \subset \boldsymbol{R}^n$ whose faces are the solution sets of integer linear equations. A *lattice point* is a point of $\boldsymbol{Z}^n$. The *interior lattice point* problem asks: Does the interior of P contain a lattice point. The *size* of P in this case is the maximum of

- The number of faces of P.
- The diameter of P.

Lemma 9.1 *ILP belongs to* **NP**

Proof: Using a real linear programming algorithm, such as the simplex method, one can find a vertex of P in polynomial time. One can then specify a cube, of side length on the order of n, which contains P in its interior.

Next, one can build an NTM which checks whether or not $p \in P$ by evaluating the equations defining the sides of P on P. Since the NTM considers all points simultaneously, the NTM runs in polynomial time. ♠

This result is probably due to Karp.

Theorem 9.2 *ILP is* **NP** *complete.*

Proof: The idea is to reduce 3-SAT to ILP. Let $C = C_1\&...\&C_n$ be some 3-compound clause. Let $a_1, \overline{a}_1, ..., a_m\overline{a}_m$ be the variables appearing in these clauses. Let P be the polytope defined by the equations

- $a_j \in [-1, 2]$ and $\overline{a}_j \in [-1, 2]$.
- $a_j + \overline{a}_j \in [0, 2]$.
- $\alpha_i + \beta_i + \gamma_i \geq 0$. Here $\alpha_i, \beta_i, \gamma_i$ are the variables of C_i.

The number of equations defining P is $O(n^3)$, and the diameter of P is at most $4\sqrt{n}$. Hence, the size of P is polynomial in the size of the clause.

If p is a lattice point in the interior of P, then every coordinte of p is either 0 or 1. Hence, we can use p to assign truth-values to the variables. By construction, p assigns true to a_i if and only if p assigns false to $\overline{a}_i$. Also, p cannot assign false to all of $\alpha_i, \beta_i, \gamma_i$. Hence, the truth-assignments specified by p satify the clause. ♠