NP completeness

May 9, 2019
• We have talked about problems that are $O(n)$ or $O(n^2)$, or $O(n \log n)$, and so on.

• When we talk like this, $n$ is some measure of the size of the problem. For instance if we are talking about sorting a set, then $n$ will be the number of elements of the set.

• If we are talking about a graph $G = (V, E)$, then it is reasonable to let $n$ be something like $|V| + |E|$ etc.

• In general, a problem instance $a$ is encoded in some way, and $n$ is just another name for $|a|$, which is the length of the encoding.
Definition

The class P is the class of problems for which there is a number $k$ and an algorithm which solves the problem and whose running time is $O(n^k)$ where $n$ is the size of the instance of the problem.

- This is called the class of *polynomial-time* problems.
- All the problems we have seen in this course so far are in P, almost always with a very small exponent.
Class P contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time.

We can semi-formally think of a Turing machine as an algorithm that solves a particular problem (it’s not a real machine!)

In a deterministic Turing machine, at every state of the algorithm (a combination of the input and stage of the computation) we have at most one way to proceed.

Everything we saw thus far falls into this category. Even “random” numbers.

The polynomial time is with respect to the space it takes to represent the input.

We’ll define decision problems shortly.
Reducing a problem to an instance of another problem is a common practice – we have also seen it in this course.

Example – we showed how to find an arbitrage by building an instance of Bellman-Ford algorithm.

This was done by constructing a graph for the arbitrage problem and replacing the weights by their $-\log$
The reduction we are concerned with here is called *polynomial reduction*.

We consider a specific class of problems called *decision problems* which are problems for which the answer is “yes” or “no”.

Some problems are naturally decision problems. For instance, there is the Hamiltonian cycle problem: given an undirected graph $G = (V, E)$, is there a simple cycle that contains every vertex in $V$?

An *instance* of the Hamiltonian cycle problem is an undirected graph $G$, together with the question, “Does $G$ have a Hamiltonian cycle?”
Other problems, such as optimization problems, are not naturally decision problems but they can be formulated as such.

For instance, we say that an independent set in an undirected graph $G = (V, E)$ is a subset $V_1$ of the vertices $V$ such that no two vertices in $V_1$ are joined by an edge in $E$.

The “maximal independent set” problem is, given a graph $G$, to find the largest independent set in $G$.

This is not a decision problem but we can formulate it “Given a positive integer $k$, is there an independent set $V_1$ in the graph of size $k$?”

The pair $(G, k)$ is an instance of the independent set problem.
Given two decision problems (like HAMILTONIAN CYCLE and INDEPENDENT SET, for example). Let us call them $A$ and $B$.

Suppose we have a function $f$ having the following properties:

- $f: A \to B$. That is $f$ maps instances of $A$ into instances of $B$.
- $f$ is implemented by an algorithm that runs in polynomial time.
- If $a$ is an instance of the problem $A$, then the time to compute $f(a)$ is $O(|a|^k)$ for some $k$, where $|a|$ is the size of the instance $a$.
- For each $a \in A$, $a$ has the same answer ("yes" or "no") as $f(a)$.

In such a case, we say that $A$ is polynomial-time reducible to $B$, and we write $A \leq_P B$. 

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
If $A \leq_P B$, then not only is $f(a)$ computable from $a$ in polynomial time, but $|f(a)|$ is a polynomially bounded function of $|a|$. In other words – $|f(a)| = O(|a|^k)$ for some $k$.

This is because for some $k, f$ only runs for $O(|a|^k)$ time and therefore cannot output an encoding for $f(a)$ that is longer than that.

If $A \leq_P B$, and if $B$ is a problem in P, then $A$ is also in P. We can take any instance $a$ of $A$, transform it in polynomial time by $f$ to a problem $f(a)$ in $B$ whose size is no more than some fixed power of $|a|$, and then solve that problem by a polynomially bounded algorithm (since $B$ is in P).
Polynomial Reducibility – Some Properties

- In other words, suppose $|f(a)| = O(|a|^p)$, and suppose the algorithm for solving problem instances of $B$ is $O(|b|^q)$.
- Then the cost of solving $f(a)$ is $O\left((|a|^p)^q\right) = O(|a|^{pq})$.
- Since $f(a)$ has the same answer ("yes" or "no") as $a$ does, solving $f(a)$ solves $a$. Therefore the polynomially bounded algorithm for $B$ yields a polynomially bounded algorithm for $A$.
- In the other direction: if $A \leq_P B$ and $A$ is in some sense hard to solve, then $B$ must be also.
- For instance, if we knew that there was no polynomial-time algorithm for $A$, then we would also know that there could be no polynomial-time algorithm for $B$ – as such an algorithm for $B$ would immediately yield one for $A$ as well.
Suppose that $X \leq_P Y$. Which of the following can we infer?

1. If $X$ can be solved in polynomial time, then so can $Y$.
2. $X$ can be solved in poly time iff $Y$ can be solved in poly time.
3. If $X$ cannot be solved in polynomial time, then neither can $Y$.
4. If $Y$ cannot be solved in polynomial time, then neither can $X$. 
Which of the following poly-time reductions are known?

1. $\text{FIND} - \text{MAX} - \text{FLOW} \leq_P \text{FIND} - \text{MIN} - \text{CUT}$.
2. $\text{FIND} - \text{MIN} - \text{CUT} \leq_P \text{FIND} - \text{MAX} - \text{FLOW}$
3. Both A and B.
4. Neither A nor B.
• **Design algorithms:** If $X \leq_P Y$ and $Y$ can be solved in polynomial time, then $X$ can be solved in polynomial time.

• **Establish intractability:** If $X \leq_P Y$ and $X$ cannot be solved in polynomial time, then $Y$ cannot be solved in polynomial time.

• **Establish equivalence:** If both $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$. In this case, $X$ can be solved in polynomial time iff $Y$ can be.
There is a very large class of problems for which no polynomial-time algorithm has been found. However, they can be checked in polynomial time. For instance, no polynomial-time algorithm is known for the Hamiltonian cycle problem but if we get a Hamiltonian cycle as a list of vertices for a graph $G$, it would be easy to check that it was indeed a Hamiltonian cycle (or wasn’t):

- Check that the list included all the vertices once and none twice, and
- And that between each two consecutive vertices in the list (and between the first and the last) there was an edge in the graph.
The Class NP

- This can obviously be done in linear time
- The class NP of problems is the class of problems that can be checked in polynomial time.
- Certainly $P \subseteq NP$
- But there are many problems in NP that are not necessarily "easy" to solve and there are some problems that are as hard as any problems in NP.
The Class NP – (semi) Formal Definition

- Class NP (Nondeterministic Polynomial) contains all decision problems that can be decided by a non-deterministic Turing machine using a polynomial amount of computation time.
- In a non-deterministic Turing machine, at every state of the algorithm (a combination of the input and stage of the computation) we have at most one way to proceed.
- At each step of the computation every choice spawns a set of possible steps.
- We have a tree of computations rather than just one linear sequence.
- If the longest path leading to termination is polynomial in the size of the input, the problem is decided in polynomial time by the machine.
- This is definitely not a very practical model...
Equivalence of the Two Definitions

- If a problem can be decided by a *non-deterministic Turing machine* using a polynomial amount of computation time.
- We have a tree of computations rather than just one linear sequence.
- If the longest path leading to termination is polynomial in the size of the input, the problem is decided in polynomial time by the machine.
- Polynomial verification means that if you are given a path in the tree (a possible solution), you can follow it and tell whether it is true or not.
Definition

A problem $A$ is *NP-hard* iff for any problem $B$ in NP, $B \leq_P A$

Definition

A problem $A$ is *NP-complete* iff

- $A$ is in NP, and
- $A$ is NP-hard.

From this it follows that all problems that are NP-complete are polynomially equivalent, in the sense that if $A$ and $B$ are NP-complete, then

$$A \leq_P B$$

$$B \leq_P A$$
Due to the polynomial equivalence of all NP-complete problem it is enough to show that if one of them can be solved in polynomial time, all of them can be.

In this case $P = NP$.

It is still an open problem, since no one has been able to prove one or another.

However, most researchers believe that $P \neq NP$. 
4. Are there any changes you wish to recommend in the course?
   No, MP, no one likes it

5. Other comments?
   P ≠ MP

5. Other comments?
   why did i do this
To show that a problem $A$ is NP-complete, it is enough to show that

- $A$ is in NP, and
- for some problem $B$ that is NP-complete, $B \leq_P A$

The reason is if $C$ is any problem in NP, we must then have $C \leq_P B \leq_P A$ which shows that $A$ is in fact NP-complete.

We are now going to give some examples of problems that are NP-complete.
The following problem is known to be NP-complete.

In fact, it is historically the first problem that was proved to be NP-complete.

Showing it is beyond the scope of this course, but it suffices to see that it is clearly in NP, and all the very best algorithms experts in the world have tried to find a polynomial-time algorithm for it, and have failed.

So it is reasonable to assume that it is NP-complete.
It is a problem in mathematical logic, which sounds very abstract but closely related to problems in chip design.

We have a set of Boolean variables. Let us call them \( \{ v_1, v_2, \ldots, v_n \} \), such that each variable can take on either the value True or False.

We make Boolean expressions using these variables and three operators:

\[ \lor \quad \text{this means } "or"
\]

\[ \land \quad \text{this means } "and"
\]

\[ \overline{v} \quad \text{this means } "not } v"
\]

and parentheses, which we use in the usual way.

An expression such as \( a \lor b \) is called a disjunction, and an expression of the form \( a \land b \) is called a conjunction.
A literal is a very simple expression which is either $v$ or $\bar{v}$ for some variable $v$.

**Definition**

A Boolean expression $e$ is in *conjunctive normal form* (CNF) if it is of the following form: $e = c_1 \land c_2 \land \cdots \land c_m$ where each $c_k$ is a clause, which by the same definition is of the form $c_k = (z_{1}^{(k)} \lor z_{2}^{(k)} \lor \ldots \lor z_{n_k}^{(k)})$ where each $z_{j}^{(k)}$ is a literal.

For instance, the expression

$$e = (v_1 \lor \bar{v}_2 \lor \bar{v}_3 \lor v_4) \land (v_1 \lor v_2 \lor \bar{v}_5) \land (v_3 \lor v_4 \lor v_5) \land (v_2 \lor v_4 \lor \bar{v}_5)$$

is an expression in CNF.
An expression in CNF is *satisfiable* iff there is an assignment of True and False values to each of the variables $v_j$ which makes the expression True.

In this example if we set $v_1 = v_4 = True$, then $e$ will be True, regardless of the values of the other variables. So $e$ is satisfiable.

The problem SAT is, given an expression in conjunctive normal form, to determine if it is satisfiable.

This is a remarkably difficult problem. There is no known way to definitively solve it other than by exhaustive search.

On the other hand, it is clearly in NP – if someone tells you a solution, you can check that solution in linear time.
3-SAT is a restricted form of SAT in which all clauses have exactly 3 literals in them.

It doesn’t simplify the problem... 3-SAT is just as hard as SAT, and is therefore NP-complete.

I will not show it here, but it involves converting every non 3-CNF to 3-CNF in a polynomial number of steps, possibly adding more variables.

But only a polynomial number of variables.
Independent Set

Given a graph $G = (V, E)$ and an integer $k$, is there a subset of $k$ (or more) vertices such that no two are adjacent?

In the example there is an independent set of size 6

Obviously, INDEPENDENT-SET is in NP
Given an instance $\Phi$ of 3-SAT, we construct an instance $(G, k)$ of INDEPENDENT-SET that has an independent set of size $k = |\Phi|$ iff $\Phi$ is satisfiable.
3SAT $\leq_P$ INDEPENDENT-SET, Construction

- $G$ contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.
- This is a polynomial time construction.

$k = 3$ and $\Phi = (\bar{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor x_4)$
Consider any satisfying assignment for $\Phi$.
Select one true literal from each clause/triangle.
This is an independent set of size $k = |\Phi|$
Let $S$ be independent set of size $k$.
$S$ must contain exactly one node in each triangle.
Set these literals to true (and remaining literals consistently).
All clauses in $\Phi$ are satisfied.

$k = 3$ and $\Phi = (\bar{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \bar{x}_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor x_4)$
Vertex Cover (VC)

- Given a graph $G = (V, E)$ and an integer $k$, is there a subset of $k$ (or fewer) vertices such that each edge is incident to at least one vertex in the subset?
- Obviously, VC is in NP (why?).
- An independent set of size 6 and a VC of size 4.
Consider the following graph G. Which are true?

1. The white vertices are a vertex cover of size 7.
2. The black vertices are an independent set of size 3.
3. Both A and B.
4. Neither A nor B.
Vertex Cover (VC) is NP-Complete

- INDEPENDENT \(-\) SET \(\equiv_P\) VERTEX \(-\) COVER.

Pf. We show \(S\) is an independent set of size \(k\) iff \(V - S\) is a vertex cover of size \(n - k\).

\(\Rightarrow\)
- Let \(S\) be any independent set of size \(k\).
- \(V - S\) is of size \(n - k\).
- Consider an arbitrary edge \((u, v) \in E\).
- \(S\) is independent, so either \(u \notin S\), or \(v \notin S\), or both.
- either \(u \in V - S\), or \(v \in V - S\), or both.
- Thus, \(V - S\) covers \((u, v)\).
**INDEPENDENT \(- SET \equiv_p VERTEX \(- COVER.**

**Pf.** We show $S$ is an independent set of size $k$ iff $V - S$ is a vertex cover of size $n - k$.

$\Leftarrow$

- Let $V - S$ be any vertex cover of size $n - k$.
- $S$ is of size $k$.
- Consider an arbitrary edge $(u, v) \in E$.
- $V - S$ is a vertex cover, so either $u \in V - S$, or $v \in V - S$, or both.
- either $u \notin S$, or $v \notin S$, or both.
- Thus, $S$ is an independent set.
**SET-COVER**

- Given a set $U$ of elements, a collection $S$ of subsets of $U$, and an integer $k$, are there $\leq k$ of these subsets whose union is equal to $U$?

- Sample application.
  - $m$ available pieces of software.
  - Set $U$ of $n$ capabilities that we would like our system to have.
  - The $i^{th}$ piece of software provides the set $S_i \subseteq U$ of capabilities.
  - Goal: achieve all $n$ capabilities using fewest pieces of software.

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$

$S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$

$S_d = \{5\}$

$S_e = \{1\}$

$S_f = \{1, 2, 6, 7\}$

$k = 2$
Given the following instance: Which size is the minimum SET-COVER?

1. 1
2. 2
3. 3
4. None of the above.

\[ U = \{1, 2, 3, 4, 5, 6, 7\} \]
\[ S_a = \{1, 4, 6\} \quad S_b = \{1, 6, 7\} \]
\[ S_c = \{1, 2, 3, 6\} \quad S_d = \{1, 3, 5, 7\} \]
\[ S_e = \{2, 6, 7\} \quad S_f = \{3, 4, 5\} \]
SET-COVER is NP-Complete

- Obviously, it is in NP.
- Given a VERTEX-COVER instance $G = (V, E)$ and $k$, we construct a SET-COVER instance $(U, S, k)$ that has a set cover of size $k$ iff $G$ has a vertex cover of size $k$.
- Universe $U = E$.
- Include one subset for each node $v \in V : S_v = \{e \in E : e$ incident to $v\}$.
- This is a polynomial construction.

```
U = \{1, 2, 3, 4, 5, 6, 7\}
S_a = \{3, 7\}
S_b = \{2, 4\}
S_c = \{3, 4, 5, 6\}
S_d = \{5\}
S_e = \{1\}
S_f = \{1, 2, 6, 7\}
k = 2
```
SET-COVER is NP-Complete

- Obviously, it is in NP.
- Given a VERTEX-COVER instance $G = (V, E)$ and $k$, we construct a SET-COVER instance $(U, S, k)$ that has a set cover of size $k$ iff $G$ has a vertex cover of size $k$.
- $G = (V, E)$ contains a vertex cover of size $k$ iff $(U, S, k)$ contains a set cover of size $k$.
- $\Rightarrow$ Let $X \subseteq V$ be a vertex cover of size $k$ in $G$.
- Then $Y = \{S_v : v \in X\}$ is a set cover of size $k$.

\[ U = \{1, 2, 3, 4, 5, 6, 7\} \]
\[ S_a = \{3, 7\} \]
\[ S_b = \{2, 4\} \]
\[ S_c = \{3, 4, 5, 6\} \]
\[ S_d = \{5\} \]
\[ S_e = \{1\} \]
\[ S_f = \{1, 2, 6, 7\} \]
\[ k = 2 \]
SET-COVER is NP-Complete

- Obviously, it is in NP.
- Given a VERTEX-COVER instance $G = (V, E)$ and $k$, we construct a SET-COVER instance $(U, S, k)$ that has a set cover of size $k$ iff $G$ has a vertex cover of size $k$.
- $G = (V, E)$ contains a vertex cover of size $k$ iff $(U, S, k)$ contains a set cover of size $k$.
- $\Leftarrow$ Let $Y \subseteq S$ be a set cover of size $k$ in $(U, S, k)$.
- Then $X = \{v : S_v \in Y\}$ is a vertex cover of size $k$ in $G$.

Diagram:

- $U = \{1, 2, 3, 4, 5, 6, 7\}$
- $S_a = \{3, 7\}$
- $S_c = \{3, 4, 5, 6\}$
- $S_e = \{1\}$
- $S_f = \{1, 2, 6, 7\}$
- $S_b = \{2, 4\}$
- $S_d = \{5\}$
- $k = 2$
Basic reduction strategies.

- Simple equivalence: 
  \[\textsc{Independent} - \text{Set} \equiv_P \textsc{Vertex} - \text{Cover}.\]
- Special case to general case: 
  \[\textsc{Vertex} - \text{Cover} \leq_P \textsc{Set} - \text{Cover}.\]
- Encoding with gadgets: \[3 - \textsc{Sat} \leq_P \textsc{Independent} - \text{Set}.\]

Transitivity. If \(X \leq_P Y\) and \(Y \leq_P Z\), then \(X \leq P Z\).

- Pf idea. Compose the two algorithms.
- Ex. \[3 - \textsc{Sat} \leq_P \textsc{Independent} - \text{Set} \leq_P \textsc{Vertex} - \text{Cover} \leq_P \textsc{Set} - \text{Cover}.\]
A Hamiltonian cycle in a graph $G$ is a simple cycle that visits each vertex.

There are two variants of this problem, depending on whether the graph is directed or undirected.

Both problems are NP-complete.

In what follows we deviate a bit from the proof in the text and prove each one separately.
Theorem

DIRECTED HAMILTONIAN CYCLE is NP-complete.

Proof.

1. DIRECTED HAMILTONIAN CYCLE is in NP. Clearly it’s polynomial-time checkable.

2. DIRECTED HAMILTONIAN CYCLE is NP-hard.

3. We will prove this by reducing 3-SAT to it: 3-SAT \( \leq_P \) DIRECTED HAMILTONIAN CYCLE

   Start with a 3-SAT instance that has \( n \) variables \( \{v_1, v_2, \ldots, v_n\} \) and \( k \) clauses \( \{c_1, c_2, \ldots, c_k\} \), where each clause is of the form \( z_1 \vee z_2 \vee z_3 \), each \( z_j \) being a literal.

4. We will show produce from it a graph \( G = (V, E) \) such that
   - The construction is polynomial in \( n + k \).
   - \( G \) has a Hamiltonian cycle iff the 3-SAT instance is satisfiable.
We assume that each clause in our 3-SAT instance involves 3 distinct variables.

If a clause is of the form \( v_1 \lor \overline{v}_1 \lor v_2 \) then it is automatically true, and we can just eliminate it from the instance.

A clause such as \( v_1 \lor v_1 \lor v_2 \) is really just \( v_1 \lor v_2 \), and we have already seen how to turn this in to a pair of clauses (with a new variable), each clause containing 3 literals.

So let us assume our 3-SAT instance contains literals corresponding to 3 different variables.

For each variable \( v_i \) we create a set of vertices in \( G \) and hook them together in a “doubly linked list”.

We have $3(k + 1)$ nodes here.

We take the list corresponding to each node and connect it to some auxiliary nodes to form a oval-like structure, and we then hook up these oval structures vertically.

We also add $k$ other nodes, each one corresponding to one of the clauses in the 3-SAT expression.
HAMILTONIAN-CYCLE – Example

v_1: 

v_2: 

v_3: 

v_n: 

\[ c_1 \]
\[ c_2 \]
\[ c_3 \]
\[ \vdots \]
\[ c_k \]
To form a Hamiltonian cycle, each row will either be traversed left-to-right or right-to-left, the choice for each row being independent of the choice for every other row.

A traversal of row $i$ left-to-right encodes the value True for the variable $v_i$, a traversal of row $i$ right-to-left encodes the value False for $v_i$.

There are $2^n$ possible Hamiltonian cycles of the graph and these different cycles correspond exactly to the $2^n$ different ways of assigning either True or False to the $n$ different variables $\{v_1, v_2, \ldots, v_n\}$.

Next we hook up the nodes $\{c_1, c_2, \ldots, c_k\}$ to the rest of the graph in such a way that the clause information is encoded.
- We divide each row (corresponding to each variable $v_i$ as follows:
- An initial node (i.e., the left-most one).
- A “separator node”.
- $k$ sets of 3 nodes each. The $j^{th}$ set corresponds to the clause $c_j$. Actually, the first two nodes in the set correspond to $c_j$ and the third node in each set is another “separator node”. We will call the first two nodes in each set the “$c_j$ group in row $i$”.
- A final node (i.e., the right-most one).
Each clause $c_j$ contains three literals ($c_j = z_1^{(j)} \lor z_2^{(j)} \lor z_3^{(j)}$).

For each of those literals, we add two edges involving $c_j$.

A literal $z$ corresponds to $v_i$ or $\bar{v}_i$.

The two edges we insert will connect $c_j$ with the two nodes in the $c_j$ group in row $i$, as follows:

- If $z = v_i$, we insert an edge from the left node in the $c_j$ group $\rightarrow c_j$ and an edge from $c_j \rightarrow$ the right node in the $c_j$ group.

- If $v_i$ is True, then (since row $i$ is traversed left-to-right), we can use these two edges to make a side trip to $c_j$, including $c_j$ in the cycle.
If \( z = \bar{v}_i \), we do things "the other way": we insert an edge from the right node in the \( c_j \) group \( \rightarrow c_j \) and an edge from \( c_j \rightarrow \) the left node in the \( c_j \) group.

The reason for doing this is that if \( v_i \) has the value False (so \( \bar{v}_i \) is True), then (since row \( i \) will be traversed right-to-left) we can use these two edges to make a side trip to \( c_j \), thus including \( c_j \) in the cycle.
HAMILTONIAN-CYCLE

- Each clause $c_j$ has three pairs of edges that are inserted for it.
- These edges will never “step on each other”: the edges from clause $c_j$ will only attach to “column $j$” of the main part of the graph, so the edges from two different clauses will never coincide.
- We assumed that no clause contains the same variable twice, so each of the three pairs of edges introduced for each clause goes to a different row, so they can’t coincide either.
- If the original expression is satisfiable, then $G$ has a Hamiltonian cycle: traverse each edge in the appropriate way (i.e., left-to-right if $v_i$ is True and right-to-left if $v_i$ is False).
- For each clause $c_j$ at least one of the literals in $c_j$ will be True.
- For the row corresponding to variable $v_i$ in that literal, a trip can be made to $c_j$ since the two edges to and from $c_j$ were set up that way. Thus each $c_j$ can be included in the cycle, and so we have a Hamiltonian cycle for $G$. 
Suppose $G$ has a Hamiltonian cycle, we must show that the original 3-SAT instance is satisfiable.

This is immediately true if we know that each $c_j$ is reached by a path to and from the same row.

Then the variable in that row corresponds to a True literal in $c_j$, and so each $c_j$ is satisfied.

All we have to prove is that if $G$ has a Hamiltonian cycle, then each $c_j$ is reached by a path to and from the same row.

Suppose it were not and we had something like the situation in the following figure.
HAMILTONIAN-CYCLE – This Can’t Happen!

$v_p$: $a_1 \ a_2 \ a_3$

$v_q$: $c_j$

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Suppose that $a_1$ is a node in some row that is reached from the left, and that the edge from $a_1$ to $c_j$ is not followed by an edge (in the path) from $c_j$ to $a_2$.

We know that either $a_2$ or $a_3$ must be a separator nodes. Consider these two possibilities separately:

**Case I:** $a_2$ is a separator node. It must be attached to the nodes on either side of it, but $a_2$ cannot be attached to $a_1$ by an edge in the Hamiltonian cycle, since $a_1$ already has two Hamiltonian cycle edges attached to it, so it’s impossible.

**Case II:** $a_3$ is a separator node. $a_1$ and $a_2$ must both correspond to the same clause ($c_j$). $a_2$ must be attached either to $c_j$ or to $a_1$ by an edge in the Hamiltonian cycle, but neither one is possible, since both those nodes already have two Hamiltonian cycle edges attached to them.
If \( a_1 \) is approached from the left the above is impossible. If it were approached from the right, then a similar argument (directions switched) would show the same.

Therefore we showed that a Hamiltonian cycle of \( G \) corresponds to an assignment of truth values to the variables \( \{v_1, v_2, \ldots, v_n\} \) that satisfies the original 3-SAT instance.

Finally, we note that the construction of \( G \) was polynomial, and that concludes the proof.
Theorem

**UNDIRECTED HAMILTONIAN CYCLE** is NP-complete.

Proof.

1. **UNDIRECTED HAMILTONIAN CYCLE** is in NP. Clearly it’s polynomial-time checkable.

2. **UNDIRECTED HAMILTONIAN CYCLE** is NP-hard. We will prove this showing: DIRECTED HAMILTONIAN CYCLE $\leq_P$ UNDIRECTED HAMILTONIAN CYCLE.

3. We start with an instance of DIRECTED HAMILTONIAN CYCLE – a directed graph $G$ – and we will construct an undirected graph $H$ which has a Hamiltonian cycle iff $G$ does.
Each vertex \( v \) in \( G \) corresponds to three vertices \( v^{\text{in}}, v^{\text{mid}}, \) and \( v^{\text{out}} \) in \( H \).

They are connected by two (undirected) edges: one between \( v^{\text{in}} \) and \( v^{\text{mid}} \), and the other between \( v^{\text{mid}} \) and \( v^{\text{out}} \).

The rest of the edges in \( H \) mirror the edges in \( G \): If \((u, v)\) is a (directed) edge in \( G \), we create an edge in \( H \) from \( u^{\text{out}} \) to \( v^{\text{in}} \).

Clearly, this is a polynomial time construction.
Lemma

If $G$ has a Hamiltonian cycle, then $H$ does.

Proof.

If $u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n \rightarrow u_1$ is a (directed) Hamiltonian cycle in $G$, then

$$u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow$$

$$u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}} \cdots \leftrightarrow$$

$$u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}$$

is an undirected Hamiltonian cycle in $H$. 

Nurit Haspel  
CS310 - Advanced Data Structures and Algorithms
Lemma

If \( H \) has a Hamiltonian cycle, then \( G \) does.

Proof.

Since each “mid” node is connected by one edge to an “in” node and one edge to an “out” node, the only way that each “mid” node can be in a cycle is for all three (“in”, “mid”, “out”) nodes to be in that cycle.

Therefore, a Hamiltonian cycle of \( H \) must be of the form

\[
\begin{align*}
&u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}} \\
&\cdots \leftrightarrow u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}
\end{align*}
\]

But this corresponds exactly to the Hamiltonian cycle

\( u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n \rightarrow u_1 \) in \( G \).
SUBSET-SUM

- Also called INTEGER PARTITION.
- An instance of the problem is a set $S$ of integers and a “target” integer $t$.
- The question is, “Is there a subset of $S$ whose sum is $t$?”
- For instance, if

$$S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$$

and $t = 3754$, then the answer is “yes”, because

$$1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = t$$
Theorem

*SUBSET SUM* is NP-complete.

**Proof.**

1. **SUBSET SUM is in NP.** This is obvious: checking that a *particular* subset adds up to $t$ can certainly be done in linear time.

2. **SUBSET SUM is NP-hard.** We will prove this by reducing VERTEX COVER to SUBSET SUM: $\text{VC} \leq_P \text{SUBSET SUM}$
   - We need to start with a graph in which we are trying to find a vertex cover of size $N$, and turn this VC instance into an instance of SUBSET SUM.
   - We take our graph, and we construct its *incidence matrix*. 
SUBSET-SUM – Example

<table>
<thead>
<tr>
<th></th>
<th>$e_0$</th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$e_4$</th>
<th>$e_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$v_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$v_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
There are exactly two 1’s in each column. That will be a key point.

We will call this matrix \( b \), and in the example above \( b[2, 1] = 0 \).

Each row can be thought of as a base-4 representation of an integer, only with the low-order digits on the left so that the row for \( v_2 \) corresponds to \( 4^2 + 4^4 + 4^5 \)

For example – the row corresponding to the vertex \( v_i \)

corresponds to the number \[ \sum_{j=0}^{\left| E \right|-1} b[i, j] \cdot 4^j \]
We extend the matrix by adding a new row for each edge, and we will put a 1 in the column that corresponds to that edge:

<table>
<thead>
<tr>
<th></th>
<th>e₀</th>
<th>e₁</th>
<th>e₂</th>
<th>e₃</th>
<th>e₄</th>
<th>e₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₀</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v₁</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v₂</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>v₃</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v₄</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e₀</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e₁</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e₂</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e₃</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e₄</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e₅</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Each column has three 1’s in it: two from vertex rows and one from an edge row. The top rows of this matrix are just the original matrix $b$.

For each vertex row we construct the number (which is just the number above, but with a high-order term added).

$$V_i = 4|E| + \sum_{j=0}^{|E|-1} b[i,j]4^j.$$ We will call these the “vertex numbers”.

For each edge row we construct the number (this time without the high-order term added) $E_k = 4^k$. We will call these the “edge numbers”.

The subset sum instance is this: the numbers in our set $S$ are just the numbers $V_i$ and $E_k$ we just constructed.

This is obviously a polynomial construction.
The target number is $t = N \cdot 4^{|E|} + 2 \cdot \sum_{j=0}^{|E|-1} 4^j$

We will show that the graph we started with has a vertex cover of size $N$ iff the subset sum problem we have just constructed is solvable.

Notice the following facts:

- If we add up any subset of numbers in $S$ (even if we add up all the numbers in $S$), there will be no “carries” from one column to the next in the base-4 addition. The reason is that each column can contain at most three 1’s, and it would take four 1’s to produce a carry.
- It follows from this that for a sum of numbers in $S$ to equal $t$ it must contain exactly $N$ vertex numbers, since that is how many vertex numbers we will need to get the high term $N \cdot 4^{|E|}$ in $t$. 

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
Lemma

If the VC instance is solvable, then the derived SUBSET SUM instance is solvable.

Proof.

- If we have a vertex cover of the graph with $N$ vertices, and if we take the sum of the corresponding vertex numbers, we have a high-order term of $N \cdot 4^{|E|}$.
- As for the other terms, since each edge in the graph is “covered”, we will have at least a contribution of $1 \cdot 4^j$ for each edge $e_j$.
- If we only have $1 \cdot 4^j$ and not $2 \cdot 4^j$, then we can add the edge number $E_j$.
- This way we have a solution to the SUBSET SUM problem.
Lemma

If the derived SUBSET SUM instance is solvable, then the VC instance is solvable.

Proof.

- Take the vertex numbers in the solution of the SUBSET SUM instance, there are exactly $N$ of them.
- The rest of the numbers in the solution (edge numbers) can only contribute at most a 1 in each remaining column.
- The vertex numbers have to contribute either 1 or 2 in each column, so each edge is covered by either 1 or 2 vertices in the subset of vertices that corresponds to the vertex numbers in the solution to the derived SUBSET SUM instance.
- Those vertices constitute a vertex cover of size $N$. 
Dynamic Programming Algorithm for Subset-Sum

Given a set $S$ of $N$ numbers with a target number $t$, we recursively decide whether to add the number $S_i$ or not:

- If we add $S_i$, solve recursively for $S - S_i$ and $t - S_i$
- Else, solve recursively for $S - S_i$ and $t$
- Boundary conditions – either $S$ is empty or $t < 0$ (when $S_i > t$)

It can be done using DP as follows: Allocate a boolean table $Sub$ of size $N + 1 \times t + 1$

$$Sub[i][j] = T \text{ if there is a subset of } S[0..i - 1] \text{ with sum equal to } j, \text{ otherwise false.}$$

DP Formulation: $Sub[i][j] = Sub[i - 1][j] || Sub[i - 1][j - S_i]$

Finally, we return $Sub[t][n]$

Runtime – $O(N \times t)$

How come we have a polynomial runtime?
Dynamic Programming Algorithm for Subset-Sum

Example – $S = \{3, 34, 4, 12, 5, 2\}$, $t = 8$

<table>
<thead>
<tr>
<th>sum $S_i$</th>
<th>0</th>
<th>3</th>
<th>34</th>
<th>4</th>
<th>12</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Nurit Haspel
CS310 - Advanced Data Structures and Algorithms
The Traveling Salesperson Problem (TSP)

- We have a set of cities, represented as vertices in a graph.
- A salesperson needs to visit each city as cheaply as possible.
- Assume that the cost is the total distance of the trip.
- Between each two vertices there is an edge with an associated weight and we want to find the shortest path that visits each node.
- To make things simple, we may also assume that the path should be a cycle.
- The associated decision problem is “Does this graph have a Hamiltonian cycle of weight \( \leq W \)?”
The Traveling Salesperson Problem (TSP)

- We can reduce UNDIRECTED HAMILTONIAN CYCLE to this problem as follows:
- Let $G$ be any undirected graph. This is the of UNDIRECTED HAMILTONIAN CYCLE.
- We will construct a graph $H$ with edge weights that will be an instance of TSP as follows:
  - The vertices of $H$ are just the vertices of $G$.
  - Every two vertices of $H$ are connected by an edge. ($H$ is a complete graph.)
  - The weight of an edge in $H$ is 0 if that edge is also an edge in $G$, and is 1 otherwise.
- Then the question “Does $H$ have a Hamiltonian cycle of weight $\leq 0$?” has a positive answer iff $G$ has a Hamiltonian cycle. Thus the TSP problem is NP-complete.
Subgraph isomorphism

An isomorphism is a bijection between the vertices of two graphs $f : V(G_1) \rightarrow V(G_2)$ such that any two vertices $u$ and $v$ of $G_1$ are adjacent in $G_1$ iff $f(u)$ and $f(v)$ are adjacent in $G_2$.
Given two graphs $G$ and $H$, is $H$ isomorphic to some subgraph of $G$?

Again, the problem is clearly in NP.

It’s NP-hard because we can reduce CLIQUE to it.

To ask the question “Does $G$ have a clique of size $k$?” is to ask the question “Does $G$ have a subgraph that is isomorphic to the complete graph on $k$ vertices?” So CLIQUE $\leq_P$ SUBGRAPH ISOMORPHISM, and so SUBGRAPH ISOMORPHISM is NP-complete.
• Sometimes a small change in the problem definition changes the complexity significantly.
• Graph isomorphism – in NP but not known whether the problem is NP-complete.
• Polynomial time solutions exist for:
  • Eulerian path/cycle – A cycle that goes through every edge once (vertices can be repeated).
  • DNF-SAT.
  • Linear programming (variables are not restricted to integers).
  • etc...