NP-Completeness
Some graph theoretic examples

Samuel Grahn

Examensarbete i matematik, 15 hp
Handledare: Vera Koponen
Examinator: Jörgen Östensson
Augusti 2016
NP-Completeness
Some graph theoretic examples

Samuel Grahn

August 11, 2016
Introduction

Since the dawn of computing, there has been plenty of discussion and research dedicated to deciding what can be calculated or solved, and what cannot. But as time passed, we got more interested in what could be computed within a reasonable time. Even as computers got faster, some problems grew in complexity in ways our computational evolution could not catch up to.

We started to divide the computable problems into subcategories; problems that have a known polynomial time solution belong to the class $\mathcal{P}$, and problems that - given a solution - can be verified within polynomial time belong to the class $\mathcal{NP}$. We will define $\mathcal{P}$ and $\mathcal{NP}$ formally later. Since any problem solvable in polynomial time can be verified (by simply finding the solution) in polynomial time, we know that $\mathcal{P} \subseteq \mathcal{NP}$, but it is still an open question in mathematics, and one of the millennium problems, to decide wether or not $\mathcal{P} = \mathcal{NP}$.

Most people believe that $\mathcal{P} \neq \mathcal{NP}$, due to the vast amount of empirical evidence that such is the case. In order to try to prove this, we created a subgroup of $\mathcal{NP}$, called $\mathcal{NP}$-complete problems. If an algorithm solves a problem that is $\mathcal{NP}$-complete in polynomial time, then all problems in $\mathcal{NP}$ are solvable within polynomial time.

The problem wether $\mathcal{P} = \mathcal{NP}$ has a lot of practical implications, since a lot of problems today, like protein folding and RSA encryption are $\mathcal{NP}$-complete. If the equality $\mathcal{P} = \mathcal{NP}$ proves to be true, it would have both nice and potentially harmful consequences. For instance, our internet security would be at risk - provided the exponent of the polynomial time bound is reasonably small - and more efficient protein folding could potentially save many lives.

---

Turing Machines

In order to provide a reasonable explanation of algorithmic complexity we are required to understand the famous Turing Machine\(^2\) a very simple construction, yet incredibly powerful. A turing machine is a mathematical model of a computer, and every problem solvable by a computer can also be solved by a Turing Machine.

**Definition 1** (Turing Machine).

A Turing machine is defined as a quadruple \(M = (K, \Sigma, \delta, s)\) where

- \(K\) is a finite set of states.
- \(s \in K\) is the initial state.
- \(\Sigma\) is a finite set of symbols, we call it the *alphabet* of \(M\). We assume \(K, \Sigma\) are disjoint sets, and that \(\sqcup, \triangleright \in \Sigma\), where \(\sqcup\) and \(\triangleright\) represent blank and start (i.e. the symbol representing empty space in a given tape and the first symbol of said tape, respectively).
- \(\delta\) is a function \(\delta : K \times \Sigma \to K \cup \{h, yes, no\} \times \Sigma \times \{\rightarrow, \leftarrow, -\}\). We assume \(h\) (the halting state), \(yes\) (the accepting state), \(no\) (the rejecting state), \(\leftarrow, \rightarrow, -\) (the cursor directions left, right and stay) are not in \(K \cup \Sigma\).

\(\delta\) is the *program* of the machine. For each combination of \(k \in K\) and \(a \in \Sigma\), it defines a step in the machine’s process \(\delta(k, a) = (k_0, a_0, D)\) where \(k_0\) is the next state, \(a_0\) is the symbol to write and \(D \in \{\rightarrow, \leftarrow, -\}\) is the direction in which the cursor will move.

The process of computing \(M(tape)\), i.e. running the machine \(M\) with input string \(tape\), can most easily be described as follows

\[
\begin{align*}
tape & \leftarrow \ldots & \triangleright \text{Gets set to input tape} \\
currentState & \leftarrow s & \triangleright \text{Set current state to initial state} \\
cursor & \leftarrow 0 & \triangleright \text{Start reading first symbol of the tape} \\
\textbf{while } currentState \neq yes, no \text{ or } h \textbf{ do} & \triangleright \text{While machine is not finished} \\
\text{cursymb} & = tape(cursor) & \triangleright \text{Read symbol at cursor location} \\
(next, symb, dir) & = \delta(currentState, cursymb) & \triangleright \text{Get info from } \delta \text{ function} \\
tape(cursor) & = symb & \triangleright \text{Write to tape} \\
move cursor in direction dir & & \triangleright \text{Change to next state} \\
current = next & & \\
\textbf{if } currentState = h \textbf{ then} & answer = tape & \triangleright \text{Return tape if halted} \\
\textbf{else } & answer = currentState & \triangleright \text{Otherwise accept or reject} \\
\end{align*}
\]

Note that a Turing Machine can run into infinite loops, and never reach any of the states \(yes, no\) or \(h\). If this occurs while computing \(M(s)\), we simply say \(M(s) = \infty\).

Example 1.
Let $M = (K, \Sigma, \delta, s)$ be a Turing machine and let $K = \{s\}$ and $\Sigma = \{\triangleright, \sqcup, 0, 1\}$. Define $\delta$ as follows

<table>
<thead>
<tr>
<th>$k \in K$</th>
<th>$\sigma \in \Sigma$</th>
<th>$\delta(k, \sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$\triangleright$</td>
<td>$(s, \triangleright, \rightarrow)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$0$</td>
<td>$(s, 1, \rightarrow)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$1$</td>
<td>$(s, 0, \rightarrow)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$\sqcup$</td>
<td>$(h, \sqcup, -)$</td>
</tr>
</tbody>
</table>

And compute $M(\triangleright 0110)$ as follows.

$(s, \triangleright 0110) \rightarrow (s, \triangleright \sqcup 110) \rightarrow (s, \triangleright 1010) \rightarrow (s, \triangleright 1000) \rightarrow (s, \triangleright 1001\sqcup) \rightarrow (h, \triangleright 1001\sqcup)$

$M$ has now reached a halting state, and we get the result $\triangleright 1001$.

Definition 2.
Let $\Sigma$ be a set of symbols. $\Sigma^*$ is the set of strings of arbitrary length consisting of symbols from $\Sigma$.

Definition 3 (Languages).
Let $L \subseteq (\Sigma \setminus \{\sqcup\})^*$ be a language, i.e. a subset of strings of the symbols in $\Sigma \setminus \{\sqcup\}$ and $M$ be a Turing Machine.
If $M(x) = yes$ $\iff$ $x \in L$ and $M(x) = no$ $\iff$ $x \notin L$, we say that $M$ decides $L$. If $L$ is decided by some Turing Machine $M$, then $L$ is called a recursive language.

Example 2.
Let $L \subseteq \{0, 1\}^*$ be the set of all strings on the form $\triangleright 0^*1^*$, i.e. any number of zeroes followed by any number of ones. **Claim:** $L$ is recursive.

**Proof.** Let $M = (K, \Sigma, \delta, s)$ be a turing machine with $K = \{s, t\}$, $\Sigma = \{0, 1\}$ and $\delta$ defined as follows

<table>
<thead>
<tr>
<th>$k \in K$</th>
<th>$\sigma \in \Sigma$</th>
<th>$\delta(k, \sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$\triangleright$</td>
<td>$(s, \triangleright, \rightarrow)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$0$</td>
<td>$(s, 0, \rightarrow)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$1$</td>
<td>$(t, 1, -)$</td>
</tr>
<tr>
<td>$s$</td>
<td>$\sqcup$</td>
<td>$(yes, \sqcup, -)$</td>
</tr>
<tr>
<td>$t$</td>
<td>$0$</td>
<td>$(no, 0, -)$</td>
</tr>
<tr>
<td>$t$</td>
<td>$1$</td>
<td>$(t, 1, \rightarrow)$</td>
</tr>
<tr>
<td>$t$</td>
<td>$\sqcup$</td>
<td>$(yes, \sqcup, -)$</td>
</tr>
</tbody>
</table>
We see that $M(s) = yes$ if $s \in L$ and $M(s) = no$ otherwise, thus $M$ decides $L$.  

**Complexity Classes**

In order to use Turing machines as a way to define complexity classes, we need to find a way to express the time it takes for a machine to compute given a string as input. Most, if not all problems can be reduced to a *decision problem*, i.e. deciding whether or not a string $s$ is in a language $L$.

**Definition 4 (Time Bound).**

Let $M = (K, \Sigma, \delta, s)$ be a Turing Machine deciding a language $L$, and $x$ be a string. Let $|x|$ denote the length of the string, i.e. the number of symbols occurring in it. Let $f : \mathbb{N} \rightarrow \mathbb{R}$. If, for an arbitrary $x \in (\Sigma \setminus \{\|\})^*$, no more than $f(|x|)$ steps are required for $M$ to decide if $x$ belongs to $L$, then we say that $f(n)$ is a *time bound* for $M$, and that $L \in \text{TIME}(f(n))$.

**Example 3.**

Take $M$ and $L$ from the proof in Example 2. Consider the time it takes for $M$ to decide if a string is in $L$ or not. Since the machine checks every piece of the string exactly once for a worst case (it aborts earlier if the string is not in $L$), we see that for a string of length $n$, $M$ operates within time $n$. Thus, $f(n) = n$ is a time bound for $M$, and $L \in \text{TIME}(n)$.

**Definition 5 ($\mathcal{P}$).**

$\mathcal{P}$ is the set of all languages that can be decided in polynomial time, i.e. a language $L$ is in $\mathcal{P}$ if and only if there exists a polynomial $p(n)$ such that $L \in \text{TIME}(p(n))$.

**Definition 6 ($\mathcal{NP}$).**

We define $\mathcal{NP}$ as the set of all decision problems that can be verified in polynomial time. Formally, this can be expressed as there existing a Turing Machine $V$ such that

- $V$ operates within polynomial time.
- $\forall x \in L(\exists c : V(\langle x, c \rangle) = yes)$, where $\langle x, y \rangle$ is the string concatenation of $x$ and $y$. We call $c$ a solution to the problem instance $x$.
- $\forall x \notin L(\forall c(V(\langle x, c \rangle) = no))$.

This can be easily explained by stating that $V$ simply checks whether or not $c$ is a valid solution to the problem.

**Definition 7 (Reduction).**

Let $a, b$ be languages and $M_a, M_b$ Turing Machines that decide $a$ and $b$ respectively. If there exists a Turing Machine $R$ that, given an input string $x$ returns another string
y, such that \( \forall x : M_a(x) = \text{yes} \iff M_b(R(x)) = \text{yes} \), we say that \( R \) is a reduction from \( a \) to \( b \). Further, if \( R \) runs within polynomial time, we call \( R \) a polynomial reduction. The existence of a polynomial reduction grants us the property of \( a \) being polynomially reducible to \( b \), which we denote by \( a \propto b \).

**Theorem 1** (Reductions are transitive).
If \( a \propto b \) and \( b \propto c \), then \( a \propto c \).

**Proof.** Let \( a, b, c \) be languages s.t. \( a \propto b \) and \( b \propto c \) with the reduction machines \( R_{ab}, R_{bc} \) respectively. The Turing Machine \( R_{ac}(x) = R_{bc}(R_{ab}(x)) \) is a reduction from \( a \) to \( c \), thus \( a \propto c \).

Informally, the \( \mathcal{NP} \)-complete problems are problems in \( \mathcal{NP} \) which, if there exists a Turing Machine solving one of them in polynomial time, then all problems in \( \mathcal{NP} \) can be solved within polynomial time, i.e. \( \mathcal{P} = \mathcal{NP} \).

**Definition 8** (\( \mathcal{NP} \)-complete).
A language \( L \) is \( \mathcal{NP} \)-complete if and only if

- \( L \in \mathcal{NP} \)
- For all \( \pi \in \mathcal{NP}, \pi \propto L \)

Note that in order to show the second criteria, we need only to show that a \( \mathcal{NP} \)-complete language is polynomially reducible to \( L \), using Theorem 1.

The first problem proven to be \( \mathcal{NP} \)-complete is the boolean satisfiability problem (\( SAT \)).

**Definition 9** (Boolean Satisfiability Problem(SAT)).
Let \( U = \{x_1, \ldots, x_n\} \) be a set of boolean variables (i.e. having two possible values; \( true \) or \( false \)) and \( \overline{U} = \{\neg x_1, \ldots, \neg x_n\} \) their negations, meaning whenever \( x_i \) is true, \( \neg x_i \) is false, and vice versa. Let \( B = C_1 \land \cdots \land C_m \), where \( C_i \) is a disjunction of any number of variables, or literals, in \( U \cup \overline{U} \). We say that \( B \) is satisfiable if there exists a truth assignment \( T : U \rightarrow \{true, false\} \) such that, when replacing each occurrence of \( x_i \) or \( \neg x_i \) in \( B \) by \( T(x_i) \) or \( \neg T(x_i) \) respectively, the expression evaluates to \( true \).

**Theorem 2.**
(SAT is \( \mathcal{NP} \)-complete) The language of all satisfiable boolean expressions is \( \mathcal{NP} \)-complete.

---

3The proof can be found in *Computational Complexity, Papadimitriou*, on page 171
Example 4.
Given a boolean expression \( B = (\neg x_1 \lor x_2) \land (\neg x_2 \lor x_1) \), we construct a truth table.

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( (\neg x_1 \lor x_2) )</th>
<th>( (\neg x_2 \lor x_1) )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Since \( x_1 = true \), \( x_2 = true \) makes \( B \) true, \( B \) is satisfiable.

Definition 10 (3-SAT).
Let \( U = \{x_1, \ldots, x_n\} \) be a set of boolean variables and \( \overline{U} = \{\neg x_1, \ldots, \neg x_n\} \) their negations. Let \( B = C_1 \land \cdots \land C_m \) be a boolean expression where each clause \( C_i \) is a disjunction of exactly three variables (i.e. \( C_i = (\alpha \lor \beta \lor \gamma) \) for some \( \alpha, \beta, \gamma \in U \cup \overline{U} \)). 3-SAT is the language of all satisfiable boolean expressions on this form.

Since our boolean expressions can be infinite, but the alphabet of a Turing machine is finite, the languages are encodings of boolean expressions and graphs. For instance, one can let each variable \( x_i \) be encoded as the binary representation of \( i \), with \( \Sigma = \{0, 1, \neg, \land, \lor, <, >\} \). This way, the expression \( (x_1 \lor x_2) \land (\neg x_1 \lor x_3) \) can be encoded as \(<1 \lor 10> \land <1 \lor 11>\).

In the following proofs, we will not construct Turing machines to represent our algorithms, but rather simply state that the algorithms we use can be reconstructed into Turing machines, using the encoding mentioned above.
Theorem 3 (3-SAT is $\mathcal{NP}$-complete).
In order to prove that SAT $\propto$ 3-SAT, we will take an arbitrary instance of SAT and convert it into an instance of 3-SAT.

Proof. First, we observe that the algorithm that verifies an instance of SAT also verifies an instance of 3-SAT, thus 3-SAT $\in \mathcal{NP}$.
Start with any boolean expression $B = C_1 \land \cdots \land C_m$ over a set of variables $U = \{x_1, \ldots, x_n\}$ and their negations $\overline{U} = \{-x_1, \ldots, -x_n\}$. Then, for each $C_i$ of size $|C_i|$, we create a new set of clauses whose conjunction is logically equivalent to it. We get four cases.

Case 1: $|C_i| = 1$, so $C_i = (x_i)$ for some $x_i$ in $U \cup \overline{U}$. Create two new variables $\alpha$ and $\beta$, occurring nowhere else, and create the clauses $D_i = (x_i \lor \alpha \lor \beta) \land (x_i \lor -\alpha \lor -\beta) \land (x_i \lor -\alpha \lor -\beta)$. Note that the only way for this conjunction to be true is if $x_i$ is true.

Case 2: $|C_i| = 2$, so $C_i = (x_i \lor x_j)$ for $x_i, x_j \in U \cup \overline{U}$. Create a new variable $\alpha$ occurring nowhere else, and let $D_i = (x_i \lor x_j \lor \alpha) \land (x_i \lor x_j \lor -\alpha)$. This conjunction is true if and only if $(x_i \lor x_j)$ is true.

Case 3: $|C_i| = 3$, let $D_i = C_i$, i.e. make no changes.

Case 4: $|C_i| > 3$, i.e. $C_i = (z_1 \lor \cdots \lor z_k)$. Create new variables $\{L_1, \ldots, L_{k-2}\}$ occurring nowhere else, and let

$$D_i = (z_1 \lor z_2 \lor L_1) \land (\neg L_1 \lor z_3 \lor L_2) \land \cdots \land (\neg L_{k-2} \lor z_{k-1} \lor z_k)$$

Note that $D_i$ is satisfiable if and only if $C_i$ is satisfiable.

Our construction creates a logical expression $B' = D_1 \land \cdots \land D_l$ that is satisfiable if and only if $B$ is satisfiable. Each clause containing $k$ occurrences of literals will be replaced by at most $3k$ clauses. The construction of each clause thusly runs in at most $\text{TIME}(3k^2)$, i.e. polynomially. There are $m$ clauses, so the construction is polynomial.

Definition 11 (NAE-3-SAT).
NAE-3-SAT (Not All Equal 3-SAT) is a variant of 3-SAT, where a clause is false whenever all three literals have the same value, and true otherwise, e.g., as an instance of NAE-3-SAT, $(\alpha \lor \beta \lor \gamma)$ is true if and only if $(\alpha, \beta, \gamma) \notin \{(true, true, true), (false, false, false)\}$.

Theorem 4 (NAE-3-SAT is $\mathcal{NP}$-complete).

Proof. An algorithm that verifies a solution to 3-SAT can be slightly altered to verify an instance of NAE-3-SAT in polynomial time, thus NAE-3-SAT $\in \mathcal{NP}$.

We reduce from 3-SAT. Given a set of variables $U = \{x_1, \ldots, x_n\}$, their negations
\[ \mathcal{U} = \{-x_1, \ldots, -x_n\} \] and a boolean expression \( B = C_1 \land \cdots \land C_m \), where \( C_i \) is a disjunction of exactly three variables in \( \mathcal{U} \cup \mathcal{U}' \). Create new variables \( x_T, x_F \) and \( c_1, \ldots, c_m \). Replace each clause \( C_i = (\alpha \lor \beta \lor \gamma) \) with \((\alpha \lor \beta \lor c_i) \land (\gamma \lor \neg c_i \lor x_F)\). Finally add a clause \((x_T \lor x_T \lor x_F)\). Call this new expression \( B' \), and the new variables \( \mathcal{U}' \) and \( \mathcal{U}' \) respectively.

Given a truth assignment \( T \) that satisfies \( B \), we can create a truth assignment \( T' \) that satisfies \( B' \) as follows

- Let \( x_F \) be false and \( x_T \) be true.
- If \( \alpha \) or \( \beta \) is true, then let \( c_i \) be false, and its negation \( \neg c_i \) makes the second clause true.
- If \( \gamma \) is true, we can let \( c_i \) be true, satisfying the first clause.
- If all three variables \( \alpha, \beta, \gamma \) are true, let \( c_i = false \).

Thus, given a satisfying thruth assignment to \( B \), we can create one in \( B' \). Next, start with a truth assignment \( T' \) that satisfies \( B' \). Since if \( X \) is a truth assignment satisfying a NAE-3-SAT expression \( E \), we know that \( X'(u) = \neg X(u) \) for \( u \in \mathcal{U}' \cup \mathcal{U}' \) (the inverse of \( X \)) also satisfies \( E \), we can assume without loss of generality that \( x_F = false \). Simply take \( T = T' \), ignoring the assignments of \( c_i, x_T, x_F \), and you will have a satisfying truth assignment to \( B \). Thus, 3-SAT \( \propto \) NAE-3-SAT, and NAE-3-SAT is \( \text{NP-complete} \). \( \square \)
Graph Theoretic Problems

Graph theoretic problems prove a useful tool in expanding the class of $\mathcal{NP}$-complete problems. They are intuitive to reduce from, and can relate to many different problems across many fields of research.

Definition 12 (Graph).
Let $V = \{v_1, \ldots, v_n\}$ be a vertex set, and $E \subset \{\{u, v\} : u, v \in V \land u \neq v\}$ be an edge set. The edge $\{u, v\}$ is commonly referred to as $uv$. A graph $G = (V, E)$ is an ordered pair, consisting of a vertex set, and an edge set.

In order to decide the complexity of graph theoretic problems, we must define the size of a graph. We can simply let the size of a graph be $n = |V|$, i.e. the size of its vertex set. We know it can at most contain $n(n - 1)/2$ edges. Thus, the maximal number of vertices and edges combined is $n + n(n - 1)/2$, a polynomial. This means that given an alphabet $\Sigma$ that encodes the vertex $v_i$ as the binary representation of $i$ (which is done in $K \log i$ for some constant $k$), we can encode any graph with $n$ vertices in polynomial time.

Definition 13 (Vertex Cover).
Let $G = (V, E)$ be a graph. A vertex cover of $G$ is a set $V' \subseteq V$ such that $uv \in E \Rightarrow u \in V' \lor v \in V'$. That is, $V'$ is a set of vertices where every edge in $E$ has at least one vertex in $V'$. As a decision problem, we ask whether a graph has a vertex cover of size at most $k$, where $k$ is an integer.

Example 5.
The graph $G = (V, E)$ where $V = \{a, b, c, d\}$ and $E = \{\{a, c\}, \{c, d\}, \{a, d\}, \{b, d\}\}$ has a vertex cover $C = \{a, d\}$ of size $|C| = 2$.

Figure 1: The graph $G = (V, E)$ and a vertex cover $C = \{a, d\}$

In order to prove the $\mathcal{NP}$-completeness of graph theoretic problems, we must define the alphabet with which we write the strings that are part of the different languages. A graph $G = (V, E)$ can for instance be encoded with the alphabet $\Sigma = \{0, 1, <, >, -\}$ by encoding the vertices $v_i \in V$ as the binary representation of $i$, each separated by the symbol $-$, and then representing each edge $\{v_i, v_j\}$ as the binary representation of $i$ and $j$ separated by $-$, enclosed between $<$ and $>$. The edge $\{v_1, v_2\}$ would then be represented as $< 1 - 10 >$. 

10
Most of the following problems also require a second parameter; an integer $k$. This can be made easy by encoding the pair $(G, k)$ by sequentially encoding $G$, followed by a new symbol $\ast$, followed by the binary representation of $k$.

**Theorem 5** (Vertex Cover(VC) is $\mathcal{NP}$-complete).
The set of all pairs $(G, k)$ such that $G$ is a graph with a vertex cover of size at most $k$ is $\mathcal{NP}$-complete.

*Proof.* For a graph $G = (V, E)$, verifying if a subset $V' \subset V$ is a vertex cover of size $k$ is done by first counting the vertices of $V'$, then checking for each edge $e \in E$ if one of its vertices is in $V'$. Checking every edge once runs in $\text{TIME}(|V||E|)$, i.e. polynomially, thus $\text{VC} \in \mathcal{NP}$.

In order to prove that $3\text{-SAT} \propto \text{VC}$, we start with a problem instance of $3\text{-SAT}$ consisting of a set of variables $U = \{x_1, \ldots, x_n\}$, their negations $U = \{\neg x_1, \ldots, \neg x_n\}$ and a boolean expression $B = C_1 \land \cdots \land C_m$ where each clause $C_i$ consists of exactly three variables, i.e. $C_i = (\alpha \lor \beta \lor \gamma)$, where $\alpha, \beta, \gamma \in U \cup U$.

Let $V' = U \cup U$ and $E' = \{x_1\neg x_1, \ldots, x_n\neg x_n\}$. For each clause $C_i = (\alpha_i \lor \beta_i \lor \gamma_i)$, create a graph $G_i = (V_i, E_i)$, where $V_i = V' \cup \{a_i, b_i, c_i\}$ and $E_i = E' \cup \{a_i, b_i, \beta_i, \gamma_i c_i\} \cup \{a_i, b_i, c_i, a_i\}$.

![Figure 2: The subgraph $G_i = (V_i, E_i)$ for $C_i = (\alpha_i \lor \neg \beta_i \lor \gamma_i)$](image)

Finally, let $V = \bigcup_{i=1}^m V_i$ and $E = \bigcup_{i=1}^m E_i$. In order to find a vertex cover for our finished graph $G = (V, E)$, we require at least two vertices in each triangle $a_i, b_i, c_i$, and at least one of $x_i, \neg x_i$ for all $i$. Thus, the minimum size possible for a vertex cover of $G$ is $n + 2m$.

We create a truth assignment from a vertex cover $V_c$ of size $n + 2m$ by letting every $x_i$ get $\text{true}$ if $x_i \in V_c$, and $\text{false}$ if $\neg x_i \in V_c$. Note that $x_i$ and $\neg x_i$ cannot both be in a vertex cover of size $n + 2m$.

Since every triangle generated by a $C_i$ has exactly two vertices in the cover, we know that the vertex not in the cover has an edge to a $x_i$ (or $\neg x_i$) in the cover. That is, setting that $x_i$ to $\text{true}$ (or $\text{false}$ if $\neg x_i$ is in the cover) makes the clause satisfied. This argument holds for all triangles, giving us the property that if the graph has a vertex
cover of size \( n + 2m \), the boolean expression it was generated from is satisfiable.

Conversely, if we have a truth assignment \( T \) that satisfies the expression, we can simply let \( V_T \) be the set of all vertices \( x_i \) that \( T \) sets to \( true \), and two vertices from every triangle, such that the first variable in the clause that is assigned \( true \) is the node not in the cover. This covers all the edges in \( G \) and has size \( n + 2m \). Thus we have proved that \( G \) has a vertex cover of size \( n + 2m \) if and only if \( B \) is satitsfiable.

This construction creates \( 3m + 2n \) vertices and \( 6m + n \) edges, i.e. running polynomially. Thus \( 3\text{-SAT} \propto \text{VC} \) and \( \text{VC} \) is \( \mathcal{NP} \)-complete.

\[ \square \]

**Definition 14** (Independent Set).
An independent set on a graph \( G = (V,E) \) is a subset \( I \subset V \) such that no vertices in \( I \) are adjacent, i.e. \( \forall u,v \in I : \{u,v\} \notin E \).

**Example 6.**
Let \( G = (V,E) \) be a graph where \( V = \{a,b,c,d\} \) and \( E = \{\{a,c\}, \{c,d\}, \{a,d\}, \{b,d\}\} \).

The set \( I = \{a,b\} \) is an independent set of \( G \).

![Figure 3: The graph \( G = (V,E) \) and independent set \( I = \{a,b\} \)](image)

**Theorem 6** (Independent Set\((IS)\) is \( \mathcal{NP} \)-complete.).
The set of all pairs \((G,k)\) such that \( G \) is a graph with an independent set of size at least \( k \) is \( \mathcal{NP} \)-complete.

**Proof.** Verifying that a set \( I \) of size \( k \) is independent in \( G \) is done by iterating over the edges in \( i \), and checking that none of them contain two vertices in \( I \). This is done in \( \text{TIME}(k^2|E|) \), i.e. polynomially. Thus, \( \text{IS} \in \mathcal{NP} \).

We reduce from VC. Start with a graph \( G = (V,E) \) where \(|V| = n\). If \( C \subset V \) is a vertex cover of \( G \), we know that \( \overline{C} = V \setminus C \) is an independent set in \( G \). If \(|C| = m\), we have an independent set of size \( k = n - m \), thus \( \text{VC} \propto \text{IS} \), and \( \text{IS} \) is \( \mathcal{NP} \)-complete. \[ \square \]

**Definition 15** (Clique).
Let \( G = (V,E) \) be a graph. A clique of \( G \) is a set \( C \subset V \) such that \( \forall v_i, v_j \in C(\{v_i, v_j\} \in E) \). In other words, any pair of vertices in \( C \) are adjacent.

**Example 7.**
Let \( G = (V,E) \) be a graph where \( V = \{a,b,c,d\} \) and \( E = \{\{a,c\}, \{c,d\}, \{a,d\}, \{b,d\}\} \).

The set \( C = \{a,c,d\} \) is a clique in \( G \).
Theorem 7 (Clique is \( \mathcal{NP} \)-complete).
Deciding whether a graph \( G \) has a clique of size \( k \) is \( \mathcal{NP} \)-complete.

Proof. We reduce from IS. Given a graph \( G = (V,E) \) with \( |V| = n \), if \( I \subset V \) is an independent set of size \( |I| = k \), we know that \( I \) is a clique in \( \overline{G} = (V,\overline{E}) \), where \( \overline{E} = \{\{u,v\} : u,v \in V \land \{u,v\} \not\in E\} \). Thus \( IC \preceq \text{Clique} \), and Clique is \( \mathcal{NP} \)-complete.

Definition 16 (Hamiltonian Circuit).
Given a graph \( G = (V,E) \), where \( |V| = n \), a hamiltonian circuit is an ordering of the vertex set \( h : \{1, \ldots, n\} \to V \) such that

- \( h(i) = h(j) \Rightarrow i = j \)
- \( \forall i \{h(i), h(i+1)\} \in E \)
- \( \{h(n), h(1)\} \in E \)

In other words, a circuit on \( G \), passing through every vertex exactly once.

Example 8.
Let \( G = (V,E) \) be a graph with vertices \( V = \{a, b, c, d\} \) and edges \( E = \{\{a,c\}, \{c,d\}, \{a,d\}, \{b,d\}, \{a,b\}\} \).

\[ \begin{array}{c}
\text{Figure 5: The graph } G = (V, E) \text{ where } (a,b,d,c) \text{ is a hamiltonian circuit.}
\end{array} \]

Definition 17 (Directed Graph).
A directed graph \( G = (V,E) \) is an ordered pair consisting of \( V \), the vertex set (defined the same way as for a regular graph) and the edge set \( E \). In a directed graph the edges are ordered pairs, going from \( x \) to \( y \), but not from \( y \) to \( x \), i.e. \( E \subset \{(x,y) : x, y \in V\} \).
**Example 9.**
Let $G = (V, E)$ be a directed graph where $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, d)(b, d), (c, a), (c, d)\}$

![Image](image_url)

**Figure 6: G = (V, E)**

**Definition 18 (Directed Hamiltonian Circuit).**
Given a directed graph $G = (V, E)$, where $|V| = n$, a hamiltonian circuit is an ordering of the vertex set $h : \{1, \ldots, n\} \rightarrow V$ such that

- $h(i) = h(j) \Rightarrow i = j$
- $\forall i \ (h(i), h(i + 1)) \in E$
- $(h(n), h(1)) \in E$

**Theorem 8 (Directed Hamiltonian Circuit (DHC) is $NP$-complete).**
The set of all directed graphs $G$ with a directed hamiltonian cycle is $NP$-complete.

**Proof.** If given a solution, it is easily verified by simply counting the vertices, and then checking the edges if the circuit is valid, thus DHC is in $NP$.

We reduce from 3-SAT. Start with a set of variables $U = \{x_1, \ldots, x_n\}$, their negations $\overline{U} = \{\neg x_1, \ldots, \neg x_n\}$ and a boolean expression $B = C_1 \land \cdots \land C_m$, where $C_i = (\alpha \lor \beta \lor \gamma)$, $\alpha, \beta, \gamma \in U \cup \overline{U}$. For each variable $x_i \in U$, create a row of vertices $c_i, 1, \ldots, c_i, 3(m+1)$. Create two vertices, $s$ and $t$. Create edges from $s$ to $c_1, 1$ and $c_1, 3(m+1)$, and from $c_n, 1$ and $c_n, 3(m+1)$ to $t$. for each $c_{i,j}$, create edges to and from $c_{i,j+1}$. Create edges from each $c_{i,1}$ and $c_{i,3(m+1)}$ to $c_{i+1,1}$ and $c_{i+1,3(m+1)}$. Call this intermediate graph $G_1$. 

14
For each clause $C_i$, create a vertex $C_i$. If $x_j$ appears in $C_i$, create edges $(c_j, 3i, C_i)$ and $(C_i, c_j, 3i + 1)$. If $\neg x_i$ appears in $C_i$, create edges $(c_j, 3i+1, C_i)$ and $(C_i, c_j, 3i)$.

This graph consists of $2 + m + 3(m + 1)n$ vertices and $2mn + 3m + 5$ edges, and is constructed in polynomial time. Now we will prove that $B$ has a satisfying truth assignment $\Leftrightarrow G$ has a Hamiltonian circuit.

Let $T$ be a truth assignment that satisfies $B$. Start at $s$, and then go to $c_{1,1}$ or $c_{1,3(m+1)}$ if $x_1$ is true or false respectively. Traverse the row (left to right if $x_1$ is true and right to left otherwise) and take any detour through $C_i$ available (only when $C_i$ is not already traversed). Then go to $c_{2,1}$ or $c_{2,3(m+1)}$ if $x_2$ is true or false respectively, and so on. Since $T$ satisfies $B$, we know that there will be at least one opportunity to traverse each $C_i$. 
and our chosen path visits every vertex of \( G \). Thus, given a satisfying truth assignment, we can find a hamiltonian circuit of \( G \).

Let \( H \) be a Hamiltonian circuit on \( G \). Since it is a circuit, we can assume without loss of generality that it starts at \( s \). A hamiltonian circuit must then go either \( c_1,1 \) or \( c_1,3(m+1) \). Let the variable \( x_i \) assume \( true \) if \( H \) proceeds to \( c_1,1 \), and \( false \) otherwise, continue this way for \( x_2, \ldots, x_n \), and then finally go to \( t \) and finish the circuit. Since \( H \) is a Hamiltonian circuit, we know that each \( C_i \) is traversed, this means each \( C_i \) contains a \( x_i \) or a \( \neg x_i \), where \( x_i \) is set to \( true \) or \( false \) respectively. Given a Hamiltonian Circuit of \( G \), we can find a satisfying truth assignment, and thus 3-SAT \( \propto \) DHC and DHC is \( \mathcal{NP} \)-complete.

**Theorem 9** (Hamiltonian Cycle (HC) is \( \mathcal{NP} \)-complete).
The language of graphs with a hamiltonian circuit is \( \mathcal{NP} \)-complete.

**Proof.** Checking if an ordering \( H \) of the vertices is a Hamiltonian circuit of a graph \( G \) runs in polynomial time, thus HC \( \in \mathcal{NP} \).

We reduce from DHC. Start with a directed graph \( G = (V, E) \). Create a new, undirected graph \( G' = (V', E') \) where \( V' \) has 3 vertices for each \( v \in V \), call them \( v_{in}, v, v_{out} \) and create edges \( \{v_{in}, v\} \) and \( \{v, v_{out}\} \). For each edge \( (a, b) \in E \), create an edge \( \{a_{out}, b_{in}\} \in E' \).

![Diagram](image_url)

Figure 9: \( G' \) generated from \( G \) as in example 8.

For \( G = (V, E) \), \(|V| = n, |E| = m\), the reduction creates \( 3n \) vertices and \( m + 2n \) edges. This is done in polynomial time.

Given a Hamiltonian circuit of \( G \), one can simply traverse the same way in \( G' \), replacing the occurrence of \( v \) with \( v_{in}, v, v_{out} \) (traversing all three). It is easy to see that this is a Hamiltonian circuit, since if it were not, there would need to be a \( v \) untraversed in \( G \). Conversely, given a Hamiltonian circuit on \( G' \), simply replace the occurrence of \( v_{in}, v, v_{out} \) with \( v \), and you have a Hamiltonian circuit in \( G \). Thus DHC \( \propto \) HC, and HC is \( \mathcal{NP} \)-complete.
**Definition 19** (Travelling Salesman Problem(TSP)).
Given a complete graph $G = (V,E)$, i.e. each pair of vertices are adjacent, a function $f : E \to \mathbb{N}$ called the *weight function*, and an integer $k$, called the limit. The *travelling salesman problem* asks if there exists a circuit of $G$ such that each vertex is visited exactly once, and when adding $f(e)$ for each traversed edge $e$, we get a maximum sum of $k$. We call this the *cost* of the circuit.

**Theorem 10** (TSP is $\mathcal{NP}$-complete).
The set of all pairs $(G,k)$ where $G = (V,E)$ is a complete, weighted graph and $G$ has a circuit with cost at most $k$ is $\mathcal{NP}$-complete.

*Proof.* We reduce from HC. Given a graph $G = (V,E)$ with $|V| = n$, create a graph $G' = (V,E')$, where $E' = \{\{u,v\} : u,v \in V\}$ (i.e. the complete edge set), and a function $f : E \to \mathbb{N}$ such that
\[
f(e) = \begin{cases} 
1 & \text{if } e \in E \\
2 & \text{if } e \notin E
\end{cases}
\]
If this graph has a solution to TSP with limit $n$, we know it only passes the edges that are in $G$, and we know $G$ has a hamiltonian circuit. Conversely, given a hamiltonian circuit, the same route will also satisfy the TSP with limit $n$. Thus $\text{HC} \preceq \text{TSP}$ and TSP is $\mathcal{NP}$-complete.

**Definition 20** ($k$-Colouring).
A $k$-colouring of a graph $G = (V,E)$ is a function $f : V \to \{1, \ldots, k\}$ such that
- $\{u,v\} \in E \Rightarrow f(u) \neq f(v)$

We say that a graph $G$ is *$k$-colourable* if there exists a $k$-colouring of $G$.

**Theorem 11** ($3$-Colouring($3C$) is $\mathcal{NP}$-complete).
The language of all graphs which have a 3-coloring is $\mathcal{NP}$-complete.

*Proof.* We reduce from NAE-3-SAT. Start with a set of boolean variables $U = \{u_1, \ldots, u_n\}$, their negations $\overline{U} = \{\neg u_1, \ldots, \neg u_n\}$ and an expression $B = C_1 \land \cdots \land C_m$, where each clause has exactly three variables. Create a graph $G = (V,E)$ with vertices $V = \{A, B, x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n, u_1, \ldots, u_n, \neg u_1, \ldots, \neg u_n, c_1, \ldots, c_m, t_1, \ldots, t_{3m}\}$. Let $E$ be a set such that
- $\forall u_i : \{u_i, x_i\}, \{u_i, \neg u_i\} \in E$
- $\forall \neg u_i : \{\neg u_i, \neg x_i\} \in E$
- $u_i \in U \cup \overline{U}$ appears in $C_i \Rightarrow \{x_i, c_i\} \in E$
- $\forall i \in \{1, \ldots, m\} : \{t_{3i}, t_{3i+1}\}, \{t_{3i+1}, t_{3i+2}\}, \{t_{3i+2}, t_{3i}\} \in E$
For $C_i = (x_j, x_k, x_l), \{x_j, t_{3m}\}, \{x_k, t_{3m+1}\}, \{x_l, t_{3m+2}\} \in E$

![Graph Diagram](image)

Figure 10: The graph from $B = C_1 \land C_2$ for $C_1 = (u_2 \lor u_1 \lor \neg u_3), C_2 = (u_2, \neg u_1, \neg u_3)$ with colouring from truth assignment $(u_1, u_2, u_3) = (true, false, false)$

Suppose that $G$ has a 3-colouring. Without loss of generality, we will assign colour 1 to $A$ and colour 2 to $B$, thus all vertices $c_i$ are assigned colour 3. Each triangle $t_{3i}, t_{3i+1}, t_{3i+2}$ must cover all three colours, thus one of the vertices is coloured 1. This vertex is adjacent to a $x_j$ or $\neg x_j$ coloured 2 (since it is also adjacent to $c_i$). The adjacent $u_j$ or $\neg u_j$ can then only be assigned colour 3. Only one of $u_j$ and $\neg u_j$ can have colour 3. Each clause $C_i$ will now contain a $u_j$ or $\neg u_j$ coloured 3. Take these variables and let a truth assignment $T$ assign these to true, and we have $T$ that satisfies $B$.

Conversely, given a truth assignment $T$ that satisfies $B$, simply assign every vertex $u_i$ or $\neg u_i$ set to true colour 3, and the other colour 2. If $u_j$ is coloured 3, assign the corresponding $x_j$ colour 2, otherwise colour it 1. Colour the corner of each triangle $t_{3m}, t_{3m+1}, t_{3m+2}$ connected to a colour 1 $x_j$ with colour 2. Since a truth assignment satisfying $B$ cannot have all variables in a clause true, we know that each triangle will have at least one such corner. Of the other corners of the triangle, at least one have a connection to a $x_j$ coloured 2. Colour this corner 1. The other corner have a connection to either a colour 1 or 2 $x_j$. Colour this corner 3. The result is a 3-colouring of $G$. Thus NAE-3-SAT $\propto$ 3C, and 3C is $\mathcal{NP}$-complete.

\[\square\]
**Theorem 12** (\(k\)-Colouring is \(\mathcal{NP}\)-complete for \(k > 3\)).
The language of pairs \((G, k)\) such that \(G\) is a graph with a \(k\)-coloring is \(\mathcal{NP}\)-complete.

**Proof.** We reduce from 3-colourability. Let \(G = (V, E)\) be a graph. Add vertex \(v\) to \(G\), and connect edges from each vertex of \(G\) to \(v\). Call the resulting graph \(G'\). It follows that \(G\) is 3-colourable if and only if \(G'\) is 4-colourable. One can repeat this argument to construct \(G''\), proving the reduction from 4-colouring to 5-colouring. Inductively, \(3C \propto k\)-colouring (for \(k > 3\)), and \(k\)-colouring is \(\mathcal{NP}\) complete for \(k > 3\). \(\square\)

**Definition 21** (Max Cut(MC)).
Given a graph \(G = (V, E)\) and an integer \(k\), we want to know if there is a partition of \(V\) into \(V_1 \subset V, V_2 = V \setminus V_1\) such that the number of edges from \(V_1\) to \(V_2\) is \(k\) or more.

**Example 10.**
Let \(G = (V, E)\) be a graph where \(V = \{a, b, c, d\}\) and \(E = \{\{a, c\}, \{c, d\}, \{a, d\}, \{b, d\}, \{a, b\}\}\).

![Figure 11: The graph \(G = (V, E)\) where \(V_1 = \{a, c\}, V_2 = \{b, d\}\) is a max cut of size 3.](image)

**Theorem 13** (Max Cut is \(\mathcal{NP}\)-complete).
The language of pairs \((G, k)\) such that \(G\) has a max cut of size at least \(k\) is \(\mathcal{NP}\)-complete.

**Proof.** We reduce from NAE-3-SAT. Start with a set of variables \(U = \{x_1, \ldots, x_n\}\), their negations \(\overline{U} = \{\neg x_1, \ldots, \neg x_n\}\) and a boolean expression \(B = C_1 \wedge \cdots \wedge C_m\), where each clause \(C_i\) contains exactly three variables. We will extend our definition of a graph to allow multiple edges between the same two vertices. This can be formalized by replacing the edge set with a set \(E = \{(u, v, i) : u, v \in V, i \in \mathbb{N}\}\), where \(i\) is a counter representing it being the \(i^{th}\) edge between \(u\) and \(v\).

Let \(V = U \cup \overline{U}\) and, for each clause \(C_i = (\alpha \lor \beta \lor \gamma)\), create edges \(\{\alpha, \beta, i\}, \{\beta, \gamma, j\}, \{\gamma, \alpha, k\}\) with \(i, j, k\) being the lowest possible integer not already in an edge. Finally, add \(n_i\) copies of the edge \(\{x_i, \neg x_i, j\}\), where \(n_i\) is the number of times \(x_i\) or \(\neg x_i\) appears in the clauses.
We will now prove that $G$ has a cut of size $5m$ if and only if $B$ is satisfiable.

Suppose that there exists a cut $S$ of size $5m$ or more. We can assume without loss of generality that all variables are separated from their negations, since if both $x_i$ and $\neg x_i$ are on the same side of the cut, they contribute together at most $2n_i$ edges to the cut, and moving the vertex contributing the lesser contribution number of the two will not decrease the size of the cut. We can think of the vertices in $S$ as true, and the ones in $V \setminus S$ as false. The total number of edges in the cut that join opposite literals is $3m$, as many as there are occurrences of variables in $B$. The remaining $2m$ edges must be obtained from the triangles that represent the clauses, and since each triangle can contribute at most 2 to the size of the cut, all triangles must be split. A triangle being split means at least one of its variables are true, and at least one is false, satisfying $B$.

Conversely, if we have a truth assignment $T$ satisfying $B$, we can simply let $x_i$ be in $S$ if $T(x_i) = true$, and otherwise let $\neg x_i$ be in $S$. Since each clause is satisfied by $T$, this will generate a cut of size $5m$. Thus, NAE-3-SAT $\propto$ MC, and MC is $\mathcal{NP}$-complete.

\[\Box\]

\textbf{References}

- Papadimitriou, Computational Complexity, Addison-Wesley Publishing Company, 1994
- William Kocay and Donald L. Kreher, Graphs, Algorithms and Optimization, Chapman & Hall/CRC, 2005
- https://courses.engr.illinois.edu/cs473/sp2011/lectures/23_notes.pdf