Extra Notes: Polytime Reductions I

There are many interesting $\mathcal{NP}$-complete problems that we can find across a wide range of study areas. We'll give a few examples here as basis for you to work with, and then follow them up in couple of weeks with a few more specialized examples.

Contents

1 About Running Times and Space Requirements

As I've said in class, we'll sometimes ignore a logarithmic factor in our running time/space analysis when said logarithmic factor doesn't have any bearing on whether an algorithm is polytime or not. See the week 6 slides for more details.

Specifically, we'll often ignore the logarithmic factors that show up when we're using pointers or indices. Because of this, we'll often say, for example, that $\langle G = (V, E)\rangle$ has a size of $\mathcal{O}(|V| + |E|)$ or of $\mathcal{O}(|V|^2)$ rather than $\mathcal{O}([|V| + |E|] \log |V|)$ or of $\mathcal{O}(|V|^2 \log |V|)$.

There are a couple of reasons for doing this:

1. It simplifies our analysis.

2. It brings our running times in line with what you may be used to from, say, CSCB63.

   In B63 we say that a BFS on a graph $G = (V, E)$, for example, takes $\mathcal{O}(|V| + |E|)$ time. But we actually need $\mathcal{O}(\log |V|)$ space and time just to read the indices for the vertices in G! So it's really closer to $\mathcal{O}([|V| + |E|] \log |V|)$. There's a semi-hidden bound on input sizes in B63 that makes this reasonable. But we'll continue to call it $\mathcal{O}(|V| + |E|)$ here for continuity's sake.

3. Finally, In the current section of the course we're primarily interested in whether a program runs in polynomial time. The logarithmic factors that we're ignoring never change the polytime/non-polytime status of a program.

While the choice to ignore these factors makes the overall program analysis simpler, you should be aware that we're making it, and you should be prepared to include logarithmic factors when they are necessary for a proper analysis:

- If a logarithm is important to the question as to whether an algorithm runs in polynomial time, you shoukd include it.

  In particular, if a number n is included as part of an input, it will usually be written in binary or decimal notation. So the size of this input will be $\mathcal{O}(\log n)$. So you have to be careful about looping over all i from 1 to n.

- If you are running a logspace analysis, you won't be asking whether something uses polynomial amount of space, but rather a logarithmic amouont of space. You can't ignore logarithmic factors in this situation.

- If you are working to give a very tight complexity bound, or need to use a counting argument to show things are working the way you expect, you'll need to account for logarithmic factors.

- If you are working to give an optimized algorithm, you'll care about more than just making it run in polytime, and so logarithmic factors will matter. So a course like CSCC73 may well require you to be more exact about your running times. Be prepared for this.

2 The Cook-Levin Theorem

The Cook-Levin theorem will prove the fundamental fact that we'll use in all of our $\mathcal{NP}$-completeness reductions: that 3SAT (and SAT) are $\mathcal{NP}$-complete.

This will give us a basis for doing any other reductions, since it is easier to give one reduction (from 3SAT) than to show that $\forall A \in \mathcal{NP}$, $A \leqslant_p L$.

But how can we possibly prove this?

A Natural $\mathcal{NP}$-complete Problem

Idea: We don't actually need to reduce from every possible A in $\mathcal{NP}$ – there's one language that "naturally" solves all of them!

The following problem is $\mathcal{NP}$-complete:

$$\mathrm{A_{B\text{-}NTM}} = \{\langle M, w, \#^r \rangle | M \text{ is an NTM that accepts } w \text{ within } r \text{ steps}\}$$

This problem is in $\mathcal{NP}$: the certificate is an accepting computation path of length $\leq r$.

To see why this has to be $\mathcal{NP}$-hard, consider the example:

- HAM-PATH $\in \mathcal{NP}$, so we can build an NTM M that decides whether a $\langle G, s, t\rangle$ is in HAM-PATH using $\mathcal{O}(n^b)$ steps for some $b \in \mathbb{N}$.
- We can set $w = \langle G, s, t\rangle$ and r to be the worst-case time M can take for an input of size $|w|$.
- If we do so, then $\langle M, w, \#^r\rangle \in \mathrm{A_{B\text{-}NTM}}$ iff $\langle G, s, t\rangle \in$ HAM-PATH.

A similar argument will clearly work for any $A \in \mathcal{NP}$.

So we really only need to show that $\mathrm{A_{B\text{-}NTM}} \leqslant_p$ 3SAT – that is, we need to find a way to embed a TM operation into a boolean formula.

This is going to be a bit like our PCP proof, in that we're going to build a ϕ that emulates the important behaviour of our original problem.

Building a Boolean Formula from a TM

So we'd like to build a boolean formula ϕ that encodes the operation of a TM M.

Let's start by looking at a deterministic TM, and then we can generalize to the non-deterministic type.

If M is deterministic, we can describe its operations by listing its configurations: e.g.,

$$C_0 = q_0 w_0 w_1 \dots w_{n-1}, C_1 = t_{0,1} q_1 w_1 \dots w_{n-1}, \dots.$$

We can also stack its configurations like so:

$$\begin{aligned} C_0 = &\quad q_0 w_0 w_1 \dots w_{n-1} \\ C_1 = &\quad t_{0,1} q_1 w_1 \dots w_{n-1} \\ &\quad \vdots \end{aligned}$$

We'll turn this stack into a table. Firstly, we note that there is some $k \in \mathbb{N}$ such that M cannot take more that n^k steps, since we're considering an M that halts in polynomial time (n^k is the r from the previous page).

We also note that, since M cannot move its head more than one square per step, it cannot look at more than n^k memory spaces. So we can buffer our stack with $\sqcup$ characters like so:

TM operation:

#	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	#
#	$t_{1,0}$	q_1	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	#
#										#
#										#
$\vdots$	$\vdots$	$\vdots$				$\vdots$				$\vdots$
#	$t_{n^k-1,0}$	$t_{n^k-1,1}$		$\dots$					t_{n^k-1,n^k-1}	#

Notice that we've placed # symbols on the ends of the tape. These just act as place markers for us.

So we have an $n^k \times (n^k + 2)$ table that completely describes the operation of M on w. We'll have to build a ϕ that encodes this M.

> **Question**: *What are the properties of the table we need to encode into ϕ if we want it to emulate the TM operation?*

There are four major properties we care about:

1. Every cell contains exactly one character.
2. The characters on the first row match C_0 (it contains q_0 and the input).
3. There is a q_{accept} character somewhere on the table.
4. Every row in the table follows from the one above given the operation of M.

There are other properties we might want to encode, e.g., there is only one head character per row. But we can derive these properties from the four listed (e.g., from properties 2. and 4.).

So if we can build a ϕ encoding these four properties, we're done! (for deterministic TMs)

Since ϕ is already a conjunction of lots of smaller clauses, we can compose it of four different clause sets:

- ϕ_{cell} is a boolean formula that is true iff property 1. holds.
- ϕ_{start} is a boolean formula that is true iff property 2. holds.
- ϕ_{accept} is a boolean formula that is true iff property 3. holds.
- ϕ_{move} is a boolean formula that is true iff property 4. holds.

If we can build these formulas, then $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$.

To start building these formulas, let's fix the variables we use: We say the variable $x_{i,j,s}$ is true if cell $[i, j]$ in the above table holds the character s.

Encoding ϕ_{cell}:

To encode this property, we need to ensure that every cell contains at least one character that is true, but not more than one.

To ensure that at least one character is in cell [i, j], we need to state that one of the character variables in that cell is true. So we write

$$\bigvee_{s} x_{i,j,s}.$$

To ensure that only one character is in the cell, we state that for any two characters s and s', at least one of their character variables is false:

$$\bigwedge_{s,s',s\neq s'} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,s'}}\right).$$

We can see that the second formula is already in 2CNF, while the first is a disjunction of many variables.

We can turn a 2CNF clause into a 3CNF clause by repeating one of the literals (or adding a dummy literal that we force elsewhere to be false).

We can turn a disjunction of many variables into a 3CNF clause like so:

We start with a clause such as $a \vee b \vee c \vee d$, and we separate it by adding new variables to get $(a \vee b \vee y_1) \wedge (\overline{y_1} \vee c \vee d)$.

So we can encode property 1. for a single cell at [i, j] into a 3CNF formula. If we call this formula $\phi_{cell:i,j}$, then clearly the ϕ_{cell} we want is just

$$\phi_{cell} = \bigwedge_{i,j} \phi_{cell:i,j}.$$

So we have ϕ_{cell}.

Encoding ϕ_{start}:

To encode ϕ_{start}, we just manually set the characters on the first row to be C_0:

$$x_{0,0,\#} \wedge x_{0,1,q_0} \wedge x_{0,2,w_0} \wedge \ldots \wedge x_{0,n^k,\sqcup} \wedge x_{0,n^k+1,\#}$$

Every variable above is already in a 1CNF clause, so we can turn it into a 3CNF clause using the same trick as we used above.
So we have ϕ_{start}.

Encoding ϕ_{accept}:

To encode ϕ_{accept}, we need to specify that the q_{accept} character can be found somewhere on the table. So we just write that one of its variables is true:

$$\bigvee_{i,j} x_{i,j,q_{accept}}.$$

We can turn this into a 3CNF using the trick described in the ϕ_{cell} section.

So we have ϕ_{accept}.

Encoding ϕ_{move}:

This is the hard one. The trick is to remember that two neighbouring configurations can differ by only three characters. So if we look at any two adjacent rows in the table, we need to only worry about windows three characters long, e.g., in the table above,

$\#$	q_0	w_0
$\#$	$t_{1,1}$	q_1

q_0	w_0	w_1
$t_{1,1}$	q_1	w_1

$\sqcup$	$\sqcup$	$\sqcup$
$\sqcup$	$\sqcup$	$\sqcup$

Now, there may be many possible allowable windows of this sort – they just need to be consistent with the TM operation. So, for example, the

q_0	w_0	w_1
$t_{1,1}$	q_1	w_1

and

$\sqcup$	$\sqcup$	$\sqcup$
$\sqcup$	$\sqcup$	$\sqcup$

might occur in many places in the table. We say a window is legal if, like the windows above, it is consistent with the TM operation given the local knowledge we have in the window. So, for example, a window like

q_0	w_0	w_1
$t_{1,1}$	q_1	q_1

would not be legal, not just because it doesn't occur, but because it can't – the TM can't have two heads at once.

In order to ensure that the steps of the TM are encoded into ϕ, ϕ_{move} will have to check that every window is legal:

$$\phi_{move} = \bigwedge_{i,j} \big(\text{The (i, j) window is legal}\big).$$

Now, suppose that we write a window A at $[i, j]$ as

$A_{i,j}$	$A_{i,j+1}$	$A_{i,j+2}$
$A_{i+1,j}$	$A_{i+1,j+1}$	$A_{i+1,j+2}$

.

Then the property that the (i, j) window is legal can be encoded as:

$$\bigvee_{A:A\text{ is legal}} \big(x_{i,j,A_{i,j}} \wedge x_{i,j+1,A_{i,j+1}} \wedge x_{i,j+2,A_{i,j+2}} \wedge x_{i+1,j,A_{i+1,j}} \wedge x_{i+1,j+1,A_{i+1,j+1}} \wedge x_{i+1,j+2,A_{i+1,j+2}}\big).$$

We note that because the A are just the 2×3 windows that could be consistent with the operation of M, the number of A is finite, and so the above formula exists – and is even bounded in size for all i and j.

We leave it as an exercise to show that this formula can be written in 3CNF, but we note that when we do, the size of the formula depends only on M, not on the input size. So we can get a 3CNF formula $\phi_{move:i,j}$.

Given $\phi_{move:i,j}$, we can just write $\phi_{move} = \bigwedge_{i,j} \phi_{move:i,j}$. So we have ϕ_{move}.

Putting it all Together

Since we've defined ϕ_{cell}, ϕ_{start}, ϕ_{accept}, and ϕ_{move}, we also have $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, as we stated above.

The only problem that remains is to ensure that the reduction is polynomial in size.

But we note that ϕ_{cell} is just the same formula for all cells. It is $\mathcal{O}(n^{2k})$ in size. The same holds for ϕ_{accept} and ϕ_{move} (note that the individual $\phi_{move:i,j}$ may be very large, but it depends on M, not on the size n of w).

We can also see that ϕ_{start} has a clause for every cell in the first row of the table: it has size $\mathcal{O}(n^k)$.

So our entire reduction is of size $\mathcal{O}(n^{2k})$, and so the reduction is polynomial in time.

To finish off, we remember that we were assuming that M was deterministic. To make it non-deterministic:

- Note that there's nothing in our construction that breaks if there's more than choice that we can mke in ny one configuration. So we can add legal windows when defining ϕ_{move} to cover the possibility of non-determinism during the program run. The proof still works!
- We'll return to this proof later, though, and when we do it'll be useful to mke use of a very prticulr type of NTM.

  So suppose that we take any language $L \in \mathcal{NP}$. Then there is a polytime verifier V for L such that L can be characterized as

  $$L = \left\{ w \mid \exists \text{ a polysized certificte } c_w \text{ such that } V \text{ accepts } \langle w, c_w \rangle \right\}.$$

  Note that the statement that c_w is polysized in w means that there are $a, b \in \mathbb{N}$ such that for all w, $|c_w| \leqslant a|w|^b$.

  Since our construction works with general NTMs, it'll work in particular for the NTM

```
$M$ = "On input $w$:
  1. Non-deterministically choose certificate $c_w$ of sie t most $a|w|^b$.
     We can do this by walking left-to-right thorough a size-$a|w|^b$ block of cells on our tape.
     In every cell we either choose a character for that cell, write that character, and move right,
     or we simply end the certificate.
     The choice is non-deterministic, and there is one choice per certificate.
  2. Run $V$ on $\langle w, c \rangle$.
  3. If it accepts, accept, otherwise reject.
```

Our construction works just fine for this prticulr type of M, and we can find such an M for every $L \in \mathcal{NP}$.

So our reduction gives us a ϕ in polynomial time, and so $\mathrm{A}_{\text{B-NTM}} \leqslant_p$ 3SAT.

So 3SAT *(and* SAT*) are* $\mathcal{NP}$*-complete, as desired.*

3 A Useful 3SAT Lemma

In the following examples we'll often reduce from 3SAT as a basis, and when we do it will be useful to argue that each clause of a given 3CNF formula ϕ has three distinct variables as its literals. We formalize this idea here.

Lemma 3:

Suppose that we are given a 3SAT formula ϕ. Then we can efficiently find an equivalent 3SAT formula ϕ' such that:

i) ϕ' uses the same variables as ϕ, with the possibility of a small number of extra variables,

ii) For each satisfying truth assignment τ of ϕ there is exactly one equivalent truth assignment τ' of ϕ', and

iii) This τ' differs from τ only in its added variables.

iv) ϕ' has three distinct variables in each clause.

Proof:

Suppose we are given an arbitrary 3SAT formula ϕ. Now, if ϕ has three distinct variables in each clause, then we are done: just set $\phi' = \phi$. So suppose that ϕ has clauses with repeated variables.

Then it either has clauses with a variable and its negation ($(x \vee \neg x \vee y)$) or it has a clause with a literal that occurs twice.

We will firstly introduce the three variables a, b, and c, where a, b, and c are distinct from the old variables. We then add the clauses

$$(\neg a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c)$$

to the formula. You can see that the only way to assign values to a, b, and c to make these clauses T is $a = b = c = F$.

Now, a clause of the form $(x \vee \neg x \vee y)$ will always be satisfied, and so if we replace it with the clause $(x \vee \neg a \vee y)$, we will have another clause that will always be satisfied (we could also remove the clause altogether, but that would risk removing the variable x from the formula). If we repeat this process until all such clauses (i.e., clauses that contain both some variable x_i and its negation $\neg x_i$), we will have a new ϕ'. If any clause of ϕ' stiull contains repeated variables, it must contain a repeated literal.

If there are clauses of this type, we can replace it as follows: Whenever we see a clause of the form $(x \vee x \vee y)$, we can replace it with $(x \vee a \vee y)$. Similarly, we replace clauses of the form $(x \vee x \vee x)$ with the clause $(x \vee a \vee b)$.

If ϕ' is the formula we obtain in this fashion, we can see:

i) ϕ' uses the same variables as ϕ, with the possibility of new added variables (a, b, and c).

ii) For each satisfying truth assignment τ of ϕ there is exactly one equivalent truth assignment τ' of ϕ'. The truth assignment τ' is ust τ with the added assignment $a = b = c = F$.

iii) As we just said in point *ii)*, τ' differs from τ only in its added variables.

iv) Finally, ϕ' contains no clause with repeated variables.

So if I preprocess any ϕ I see with this strategy, I'll always get three distinct variables per clause.

In fact, we can also see that ϕ' contains at most seven more clauses then ϕ, and that all of its clauses can be efficiently found using ϕ. So this entire process can be run in polynomial time.

■

4 Examples

4.1 Boolean Logic and Satisfiability

4.1.1 3SAT

Definition of 3SAT:

Instance: A Boolean formula ϕ in 3CNF.

Question: Is there a truth assignment τ to the variables of ϕ that sets ϕ to true?

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is simply a truth assignment τ to the variables in ϕ.
3) The size of this certificate is $\mathcal{O}(k)$, where k is the number of variables in ϕ. Since every variable occurs at least one in the clauses, this certificate is polynomial in the size of ϕ.
4) A verifier for this certificate would be:

```
Let $V$ =“On $\langle\phi, \tau\rangle$:”
  1. For $i = 1$ to $n$ (where $n$ is the number of clauses in $\phi$):
     $\mathcal{O}(n)$ iterations.
  2.     Use $\tau$ to assign truth values to the literals in the clauses of the clause $c_i$.
     $\mathcal{O}(n)$ time per iteration.
  3.     If $\tau$ does not set at least one literal in $c_i$ to true, reject.
     $\mathcal{O}(1)$ time per iteration.
  4. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of ϕ.

So 3SAT is in $\mathcal{NP}$.

Proof that 3SAT is $\mathcal{NP}$-hard:

See the section on the Cook-Levin theorem.

■

4.1.2 SAT

Definition of SAT *(Boolean Satisfiability)*:

> **Instance:** A Boolean formula ϕ.
>
> **Question:** Is there a truth assignment τ to the variables of ϕ that sets ϕ to true?

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

The proof that this is in $\mathcal{NP}$ is essentially the same as for 3SAT – we'll just have to parse the Boolean formula and apply τ from the leaves to the root.

Proof that SAT is $\mathcal{NP}$-hard:

This is also an immediate concequence of the Cook-Levin Theorem.

■

4.1.3 X1-3SAT

Definition of X1-3SAT (*Exactly-One-3SAT*):

> **INPUT:** A 3CNF ϕ.
>
> **QUESTION:** Does ϕ have a satisfying assignment in which exactly one literal in each clause is set to true?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is simply a truth assignment τ to the variables in ϕ.
3) The size of this certificate is $\mathcal{O}(k)$, where k is the number of variables in ϕ. Since every variable occurs at least one in the clauses, this certificate is polynomial in the size of ϕ.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle \phi, \tau \rangle$:"
   1. For $i = 1$ to $n$ (where $n$ is the number of clauses in $\phi$):
      $\mathcal{O}(n)$ iterations.
   2.    Use $\tau$ to assign truth values to the literals in the clauses of the clause
$c_i$.
      $\mathcal{O}(n)$ time per iteration.
   3.    If $\tau$ does not set exactly one literal in $c_i$ to true, reject.
      $\mathcal{O}(1)$ time per iteration.
   4. Accept.
      $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of ϕ.

So X1-3SAT is in $\mathcal{NP}$.

The reduction 3SAT $\leqslant_p$ X1-3SAT**:**

Let an input 3CNF ϕ be given. Note that we can assume using 3 that the clauses in ϕ each have three distinct variables.
We will build a new ϕ' as follows: For each clause $C_i = (\ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3})$, we introduce the six new variables a_i, b_i, c_i, d_i, and e_i. We replace the clause C_i with the new clauses

- $(a_i \vee b_i \vee c_i)$,
- $(a_i \vee \neg\ell_{i_1} \vee d_i)$,
- $(b_i \vee \neg\ell_{i_2} \vee e_i)$,
- $(c_i \vee \neg\ell_{i_3} \vee f_i)$,

The idea behind these clauses is as follows: in the X1-3SAT problem, we have to be able to find one true literal per clause, while the other two literals must be false. We'll use this property to extend any satisfying truth assignment τ for ϕ into a satisfying assignment (satisfying in the X1-3SAT sense) τ' for ϕ'. If τ is not satisfying, no such τ' will be possible. Given a satisfying τ, then, τ' will encode a guess as to which of the three literals in C_i is true τ, and then check that guess (more than one can be true, of course).

The first of these clauses, the $(a_i \vee b_i \vee c_i)$, acts as a choice for which of the three literals ℓ_{i_1}, ℓ_{i_2}, and ℓ_{i_3}, we expect to be true. So if we set $a_i = T$, we're asserting that ℓ_{i_1} is also T under τ, while ℓ_{i_2} and ℓ_{i_3} may take either value.

Clearly, if we make a good guess (if we can choose a true literal in C_i under τ), then C_i is satisfied by τ. And if C_i is satisfied by τ, we can make a good guess.

> If we set, say, $a_i = T$ and $\ell_{i_1} = T$ under τ as well, then the first clause, then our second new clause $(a_i \vee \neg\ell_{i_1} \vee d_i)$ evaluates to $(T \vee F \vee d_i)$, where d_i can take either value. Setting d_i to F will allow us to set exactly one literal in this clause to T.
>
> Similarly, the clauses $(b_i \vee \neg\ell_{i_2} \vee e_i)$ and $(c_i \vee \neg\ell_{i_3} \vee f_i)$ will set to $(F \vee \neg\ell_{i_2} \vee e_i)$ and $(F \vee \neg\ell_{i_3} \vee f_i)$, respectively. Setting $e_i = \ell_{i_2}$ and $f_i = \ell_{i_3}$ lets us assign exactly one literal per clause to true for all of our clauses.
>
> The analysis is similar for the choices $b_i = T$ and $c_i = T$.

So if τ is a satisfying assignment (and so C_i is satisfied by τ), we can extend τ to a certificate to ϕ' that satisfies the clause C_i.

On the other hand, if we make a bad guess the truth assignment should not extend (in particular, if there is no true literal in C_i then we cannot make a good guess.)

If we set, say, $a_i = T$ and $\ell_{i_1} = F$, then the clause $(a_i \vee \neg \ell_{i_1} \vee d_i)$ evaluates to $(T \vee T \vee d_i)$. This truth assignment will not satisfy the X1-3SAT criteria, regardless of the other variable assignments.

The analysis for the choices $b_i = T$ and $c_i = T$ in this case, as well.

So if we repeat this procedure for every clause, we find:

$\Rightarrow$) If ϕ has a satisfying truth assignment τ, then we can extend τ' to the new variables as described above to get a certificate that $\langle \phi' \rangle \in$ X1-3SAT.

$\Leftarrow$) If ϕ' has a certificate τ' showing it is in X1-3SAT, then we can restrict τ' to the variables in ϕ to get a truth assignment τ. By our arguments above, τ satisfies ϕ, so that $\langle \phi \rangle \in$ 3SAT.

Finally, we can see that this reduction replaces each of the n clauses of ϕ with four new clauses, so that the result is a 3CNF of size $4n$. Each of these clauses can be found in polytime, and so the entire construction is polytime, as required.

Therefore, X1-3SAT is $\mathcal{NP}$-complete.

■

4.1.4 NAE-3SAT

Definition of NAE-3SAT (*Not-All-Equal-3SAT*):

INPUT: A 3CNF ϕ.

QUESTION: Does ϕ have a satisfying assignment in which at least one literal in each clause is set to true at least one literal in each clause is set to false?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is simply a truth assignment τ to the variables in ϕ.
3) The size of this certificate is $\mathcal{O}(k)$, where k is the number of variables in ϕ. Since every variable occurs at least one in the clauses, this certificate is polynomial in the size of ϕ.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle \phi, \tau \rangle$:"
  1. For $i = 1$ to $n$ (where $n$ is the number of clauses in $\phi$):
     *$\mathcal{O}(n)$ iterations.*
  2.    Use $\tau$ to assign truth values to the literals in the clauses of the clause $c_i$.
     *$\mathcal{O}(n)$ time per iteration.*
  3.    If $\tau$ does not set either exactly one or exactly two literals to true in $c_i$ to true, *reject.*
     *$\mathcal{O}(1)$ time per iteration.*
  4. *Accept.*
     *$\mathcal{O}(1)$ time.*
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of ϕ.

So NAE-3SAT is in $\mathcal{NP}$.

The reduction 3SAT $\leqslant_p$ NAE-3SAT**:**

Let an input 3CNF ϕ be given. We will build the formula ϕ' as follows:

- The variables in ϕ' will be the variables x_i in ϕ, plus an extra variable x_T not in ϕ. We will also add three variables per clause C_j: a_j, b_j, and c_j.
- For each clause $C_j = (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ in ϕ we will add the following clauses to ϕ':

  $(\neg x_T \vee \ell_{j,1} \vee a_j)$,
  $(\neg a_j \vee \ell_{j,2} \vee b_j)$,
  $(\neg b_j \vee \ell_{j,3} \vee c_j)$, and
  $(c_j \vee x_T \vee x_T)$.

  Note that we do not include the clause $(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ in ϕ'.

  Once we've done this for every clause, we return the resulting ϕ'.

We argue that $\langle \phi \rangle \in$ 3SAT iff $\langle \phi' \rangle \in$ NAE-3SAT.

Proof that $\langle \phi \rangle \in$ 3SAT iff $\langle \phi' \rangle \in$ NAE-3SAT:

Suppose that $\langle \phi \rangle \in$ 3SAT:

Then it has a satisfying truth assignment τ.

We create the new truth assignment τ' by extending τ to the new variables. After copying the variable assignments from τ to τ', we set $x_T = T$.

Given any clause C_j, we set the values of a_j, b_j, and c_j as follows:

- If $\ell_{j,1} = T$, we set $a_j = F$. Otherwise we set $a_j = T$.
- If $\ell_{j,2} = T$ or $a_j = F$, we set $b_j = F$. Otherwise we set $b_j = T$.
- If $\ell_{j,3} = T$ or $b_j = F$, we set $c_j = F$. Otherwise we set $c_j = T$ (note that we never get to this point if one of the three literals from ϕ is true).

We can see that under this assignment:

- If $\ell_{j,1} = T$, then the four clauses corresponding to C_j evaluate to

$$(F \vee T \vee F) \wedge (T \vee \ell_{j,2} \vee F) \wedge (T \vee \ell_{j,3} \vee F) \wedge (F \vee T \vee T).$$

- If $\ell_{j,1} = F$ and $\ell_{j,2} = T$, then the four clauses corresponding to C_j evaluate to

$$(F \vee F \vee T) \wedge (F \vee T \vee F) \wedge (T \vee \ell_{j,3} \vee F) \wedge (F \vee T \vee T).$$

- If $\ell_{j,1} = \ell_{j,2} = F$ and $\ell_{j,3} = T$, then the four clauses corresponding to C_j evaluate to

$$(F \vee F \vee T) \wedge (F \vee F \vee T) \wedge (F \vee T \vee F) \wedge (F \vee T \vee T).$$

So all four claues have at least one true and one false literal.

If we do this for every clause C_j, we will have a truth assignment τ' such that for every clause, at least one literal is true and one literal is false.

So $\langle \phi' \rangle \in$ NAE-3SAT.

- **Suppose that $\langle \phi' \rangle \in$ NAE-3SAT:**

Then it has a truth assignment τ' such that for every clause, at least one literal is true and one literal is false.

Now, we can see that if τ' is a certificate for ϕ', then so is $\neg\tau'$ (that is, the truth assignment obtained by negating every variable assignment in τ'). So we can assume wlog. that in τ', $x_T = T$.

Let τ be the truth assignment for ϕ that we get by taking the variable assignments from τ' for the variables x_i in ϕ.

We argue that τ is a satisfying truth assignment for ϕ: that is, for every C_j, at least one literal evaluates to true.

Suppose otherwise. Then, there is some $C_j = (\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3})$ such that $\ell_{j,1} = \ell_{j,2} = \ell_{j,3} = F$ under τ (and so under τ', as well).

Then, we can see that the four C_j clauses in ϕ' would be forced to evaluate as follows:

$$\begin{aligned}
&(\neg x_T \vee \ell_{j,1} \vee a_j) \wedge (\neg a_j \vee \ell_{j,2} \vee b_j) \wedge (\neg b_j \vee \ell_{j,3} \vee c_j) \wedge (c_j \vee x_T \vee x_T) \\
=&(F \vee F \vee a_j) \wedge (\neg a_j \vee F \vee b_j) \wedge (\neg b_j \vee F \vee c_j) \wedge (c_j \vee T \vee T) \\
=&(F \vee F \vee T) \wedge (F \vee F \vee b_j) \wedge (\neg b_j \vee F \vee c_j) \wedge (c_j \vee T \vee T) \\
=&(F \vee F \vee T) \wedge (F \vee F \vee T) \wedge (F \vee F \vee c_j) \wedge (c_j \vee T \vee T)
\end{aligned}$$

After which there would be no way to assign a value to c_j without getting either an $(F \vee F \vee F)$ or a $(T \vee T \vee T)$ clause.

This would mean that τ' was not a certificate for ϕ', a contradiction.

So τ is a satisfying truth assignment for ϕ, and so $\langle\phi\rangle \in$ 3SAT.

Finally, we can see that this reduction replaces each of the n clauses of ϕ with four new clauses, so that the result is a 3CNF of size $4n$. Each of these clauses can be found in polytime, and so the entire construction is polytime, as required.

Therefore, NAE-3SAT is $\mathcal{NP}$-complete.

■

4.2 Graph Theory

4.2.1 CLIQUE

Definition of CLIQUE:

Instance: A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.

Question: Does G contain a clique of size k (i.e., a set C of k vertices in V such that, for every $u, v \in C$, where $u \neq v$, $\{u, v\} \in E$)?

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the size-k clique C (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of variables in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle G = (V, E), C\rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C$ has at least $k$ and at most $n$ elements.
     $\mathcal{O}(n)$ time.
  3. For $i = 1$ to $k - 1$:
     $\mathcal{O}(n)$ iterations.
  4.   For $j = i + 1$ to $k$:
     $\mathcal{O}(n)$ iterations per value of $i$.
  5.     Check that $\{C[i], C[j]\} \in E$
         $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time per iteration, depending on your implementation.
  6. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^3)$ time, and so is polytime in the size of G.

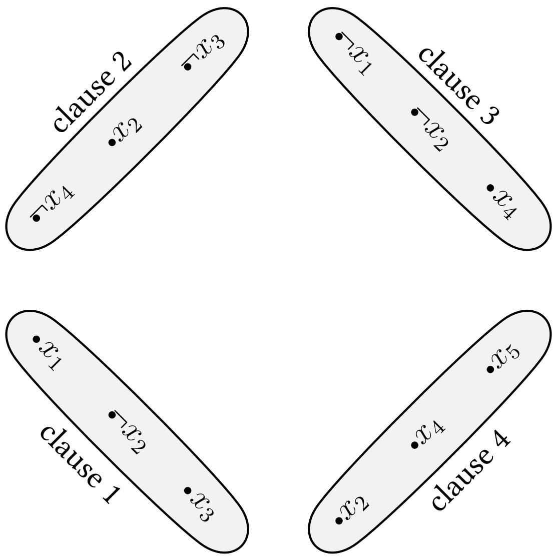
So CLIQUE is in $\mathcal{NP}$.

Reduction from 3SAT (adapted from (Sipser, 2013)):

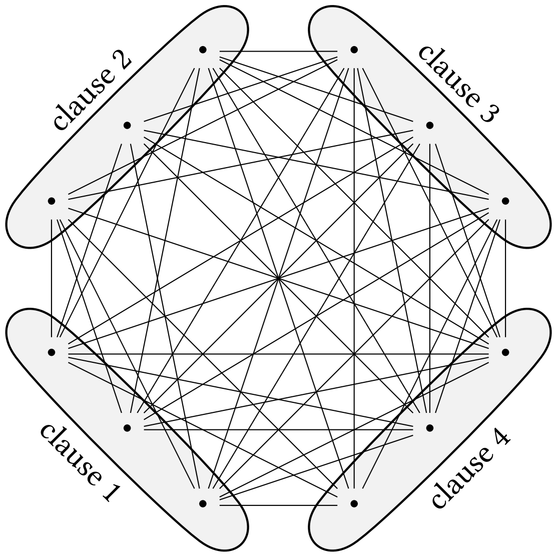
Suppose we are given a 3SAT instance ϕ with ℓ variables and k clauses.

e.g., $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_4 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee x_4 \vee x_5)$

Idea: Build a G so that each clause maps to a subset of vertices. Each vertex corresponds to one of the three literals in the clause:

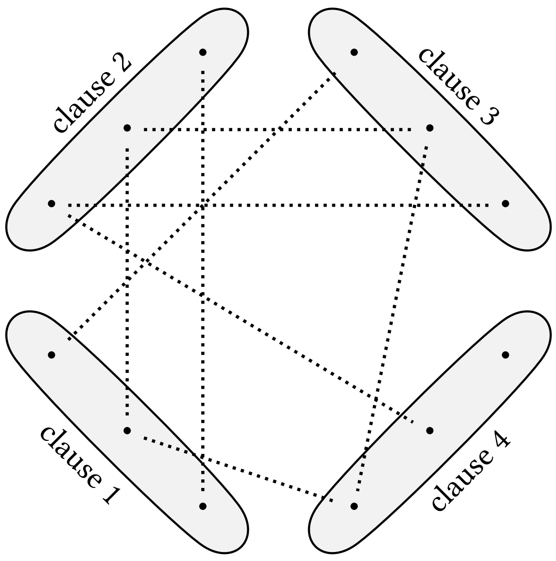


Now, add edges between <u>different</u> subsets whenever those edges could correspond to some consistent truth formula:



Sorry about how messy it is — that's part of how the reduction works. But look for the edges that are missing.

The missing edges:



This is the G we return (the first one, not the one with the missing edges). To finish, we simply set k to be the number of clauses in ϕ.

We also note that G has a k-clique iff $\phi \in$ 3SAT:

(⇐): Suppose $\phi \in$ 3SAT:

- It has a satisfying assignment.
- Let S be a set of vertices in G that takes one vertex from each clause group that matches the truth assignment.
- These variables are connected in G.

- $\Rightarrow S$ is a k-clique in G.

(⇒): Suppose G has a k-clique C:

- C must have exactly one vertex in each clause group.
- Every variable node in C must be chosen consistently (i.e., you can't choose x_1 in one clause group and $\overline{x_1}$ in another, since there would be no edge between them).
- We can read the truth values of these variables to get a truth assignment. (if a variable is not set by any node in C, we can set it however we want).
- Every vertex in C will correspond to a literal that makes its clause true, and so every clause in ϕ will be set to true.
- So this is a satisfying assignment for ϕ.

We note that this construction can be made in polynomial time:

Given a ϕ with a clauses and b variables, the number of vertices in G will be $3a$, and the maximum number of edges will be $9a/2$. We can determine whether each of these edges is in E with at worst $\mathcal{O}(n)$ time, and so the entire reduction is polytime.

So 3SAT $\leqslant_p$ CLIQUE, and so CLIQUE is $\mathcal{NP}$-complete.

■

4.2.2 INDEPENDENT-SET

Definition of INDEPENDENT-SET (IS):

> **Instance:** A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.
>
> **Question:** Does G contain an independent set of size k (i.e., a set I of k vertices in V such that, for every $u, v \in I$, where $u \neq v$, $\{u, v\} \notin E$)?

Reduced from: CLIQUE.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the size-k independent set I (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of variables in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle G = (V, E), I \rangle$:"
 1. Reject if any of the following checks fails:
    $\mathcal{O}(1)$ *time.*
 2. Check that $I$ has at least $k$ and at most $n$ elements.
    $\mathcal{O}(n)$ *time.*
 3. For $i = 1$ to $k - 1$:
    $\mathcal{O}(n)$ *iterations.*
 4.   For $j = i + 1$ to $k$:
    $\mathcal{O}(n)$ *iterations per value of* $i$.
 5.     Check that $\{I[i], I[j]\} \notin E$
        $\mathcal{O}(n)$ *or* $\mathcal{O}(1)$ *time per iteration, depending on your implementation.*
 6.*Accept.*
    $\mathcal{O}(1)$ *time.*
```

5) We can see that this verifier runs in $\mathcal{O}(n^3)$ time, and so is polytime in the size of G.

So INDEPENDENT-SET is in $\mathcal{NP}$.

Reduction from CLIQUE:

Suppose we are given a CLIQUE instance $\langle G = (V, E), k \rangle$ where G has n vertices and m edges.

We define the **complement** $\overline{G}$ of G to be the graph $(V, \overline{E})$: That is, $\overline{G}$ has the same vertices as G, and an edge $\{u, v\}$ is in $\overline{E}$ (is an edge of $\overline{G}$) iff $\{u, v\} \notin E$ (is not an edge of G).
We return the pair $\langle \overline{G}, k \rangle$.

We also note that $\overline{G}$ has a size-k independent set iff G has a k-clique:

($\Leftarrow$): Suppose G has a k-clique C:

- Then for any $u \neq v \in C$, $\{u, v\} \in E$

- $\Rightarrow \{u, v\} \notin \overline{E}$
- $\Rightarrow C$ is a size-k independent set in $\overline{G}$.

  ($\Rightarrow$): Suppose $\overline{G}$ has a size-k independent set I: Then for any $u \neq v \in I$, $\{u, v\} \notin \overline{E}$
- $\Rightarrow u \neq v \in E$
- $\Rightarrow I$ is a k-clique in G.

So CLIQUE $\leqslant_p$ INDEPENDENT-SET, and so INDEPENDENT-SET is $\mathcal{NP}$-complete.

■

4.2.3 VERTEX-COVER

Definition of VERTEX-COVER (VC) :

> **Instance:** A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.
>
> **Question:** Does G contain a vertex cover of size k or less (i.e., a set V' of k vertices in V such that, for every edge $\{u, v\} \in E$, at least one of u or v is in V')?

Reduced from: INDEPENDENT-SET.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the size-k vertex cover V' (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle G = (V, E), V' \rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $V'$ has at most $\min(k, n)$ elements.
     $\mathcal{O}(n)$ time.
  3. For all edges $\{u, v\} \in E$:
     $\mathcal{O}(n^2)$ iterations.
  4.     Check that at least one of $u$ or $v$ is in $V'$.
     $\mathcal{O}(n)$ time per iteration.
  5. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^3)$ time, and so is polytime in the size of G.

So VERTEX-COVER is in $\mathcal{NP}$.

We see that for any $u \neq v \in V$, there is a path from u to v only if $\pi(v) > \pi(u)$ (note that the converse may not be true). But this means that if there is a path from u to v, there cannot be a path from v to u. So G_π is acyclic.

So if C is a cycle in G, then C cannot be a subgraph of G_π, and therefore it must have an edge in $S\pi$. Since this is true for any cycle C in G, we see that S_π is a feedback arc set for G. Thi proves the first part of the assertion.

For the second part of the assertion, let S be any feedback arc set of G, and let G_S be the subgraph of G with every edge in S removed. Then G_S is acyclic.

Now, we know that we can topologically order any acyclic graph, and so this gives us an ordering π_S of V. Since a topological order for a DAG such as G_S must extend the initial ordering, we can see that every edge in G_S is a forward edge in π_S. But then, every backward edge must be in S, since those are the only remaining edges. So π_S is an ordering that satisfies the second part of the assertion.

Finally, we note that if $S \neq S_{\pi_S}$ (that is, if S is not exactly the set of backward edges in the ordering π_S), then we can take the set of backward edges to obtain a smaller feedback arc set. So the problem of finding the smallest feedback arc set exactly coincides with the problem of finding the π that minimizes the number of backwards edges, as required.

■

This observation is not needed for this reduction itself, but it is necessary to understand the reductions from FEEDBACK-ARC-SET to a number of permutaion-related problems, such as GROUPING-BY-SWAPPING.

Reduced from: VERTEX-COVER.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set S.
3) The size of this certificate is $\mathcal{O}(m)$, where m is the number of edges in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle G = (V, E), k, S\rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $S \subseteq E$.
     $\mathcal{O}(m^2)$ time.
  3. Check that $|S| \leqslant k$.
     $\mathcal{O}(m)$ time.
  4. Remove every edge in $S$ from $G$ to get a new graph $G'$.
     $\mathcal{O}(n^2)$ time.
  5. Run a DFS on $G'$ to determine if it has any cycles.
     $\mathcal{O}(n + m)$ time.
  6. If there are cycles,reject, otherwise accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2 + m^2) = \mathcal{O}(n^4)$ time, and so is polytime in the size of G.

So FEEDBACK-ARC-SET is in $\mathcal{NP}$.

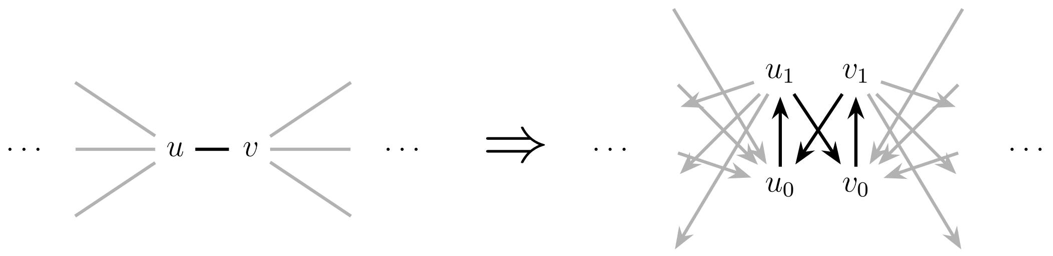
Reduction from VERTEX-COVER:

Suppose we are given an instance $\langle G = (V, E), k\rangle$ of VERTEX-COVER.

We construct $\langle G' = (V'E'), k'\rangle$ as follows:

- $V' = V_0 \cup V_1$, where each V_i is a copy of V.
- For every $v \in V$, we add the edge (v_0, v_1) to G'.
- For every edge $\{u, v\}$, we add the edges (u_1, v_0) and (v_1, u_0) to E.

  So we end up with something like this:



- We let $k' = k$.

We argue that $\langle G', k'\rangle \in$ FEEDBACK-ARC-SET iff $\langle G, k\rangle \in$ VERTEX-COVER:

Suppose $\langle G, k\rangle \in$VERTEX-COVER.

Then it has a size-k vertex cover V'.

Let $S = \{(v_0, v_1) | v \in V'\}$. Clearly, $|S| = |V| = k = k'$.

Now, let C be any cycle in G'.

Then C contains either some vertex u_0 or some vertex u_1, where $u \in V$. Since the only edge out of u_0 is (u_0, u_1), and since the only edge into u_1 is also (u_0, u_1), C must contain the edge (u_0, u_1) (and the vertex u_1).

Now, the edge in C out of u_1 must be of the form (u_1, v_0), for some $v \in V$. Since the only edge out of v_0 is (v_0, v_1), the edge (v_0, v_1) is in C.

Since $\{u, v\}$ is an edge in G, we know that at least one of u or v is in V'.

So at least one of the edges (u_0, u_1) or (v_0, v_1) is in S.

So S is a size-k' feedback arc set for G'.

$\Rightarrow \langle G', k'\rangle \in$ FEEDBACK-ARC-SET.

Suppose $\langle G', k' \rangle \in$ FEEDBACK-ARC-SET.

Then it has a size-k' feedback arc set S. We argue that we can assume that S contains only edges of the form $(u_0, u_1), u \in V$ (i.e., no edges of the form $(u_1, v_0), u, v \in V$).

> Suppose otherwise – that is, that S has an edge of the form $(u_1, v_0), u, v \in V$.
>
> Since the only edge out of v_0 is (v_0, v_1), any cycle of G' that contains (u_1, v_0) must also contain (v_0, v_1). So we can replace (u_1, v_0) with (v_0, v_1) to get another size-k' feedback arc set.
>
> *(If (v_0, v_1) is already in S we can remove (u_1, v_0) to get a smaller feedback arc set.)*
>
> After doing this to all edges of the form (v_0, v_1) in S, we'll have an S that satisfies our assumption.

Let V' be the set of $v \in V$ such that there is an edge $(v_0, v_1) \in S$. Clearly, $|V'| = |S| \leqslant k' = k$ (using our assumption on S).

Now, we can see that for any $u, v \in V$ such that $\{u, v\} \in E$, the graph G' contains the cycle $[u_0, u_1, v_0, v_1]$.

This cycle has four edges: (u_0, u_1), (u_1, v_0), v_0, v_1 and v_1, u_0.

By our assumption on S, we know that S must contain at least one of the edges (u_0, u_1) or (v_0, v_1). So at least one of the vertices u or v is in V'.

So V' is a size-k vertex cover of G.

$\Rightarrow \langle G, k \rangle \in$ VERTEX-COVER.

Finally, we can see that $|V'| = 2|V|$ and $|E'| = |V| + 2|E|$. Furthermore, $k' = k$. All of these vertices and edges (and the value for k') can be found in polynomial time, and so this entire reduction is polytime.

So FEEDBACK-ARC-SET is $\mathcal{NP}$-complete, as required.

■

4.2.5 HAM-PATH

Definition of HAM-PATH *(Hamiltonian Path)*:

> **Instance:** A directed graph $G = (V, E)$ with specified vertices s and t.
>
> **Question:** Does there exist a Hamiltonian path in G from s to t (i.e., a simple path from s to t that passes through every $v \in V$ exactly once)?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.

2) Its certificate is a Hamiltonian path P from s to t (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

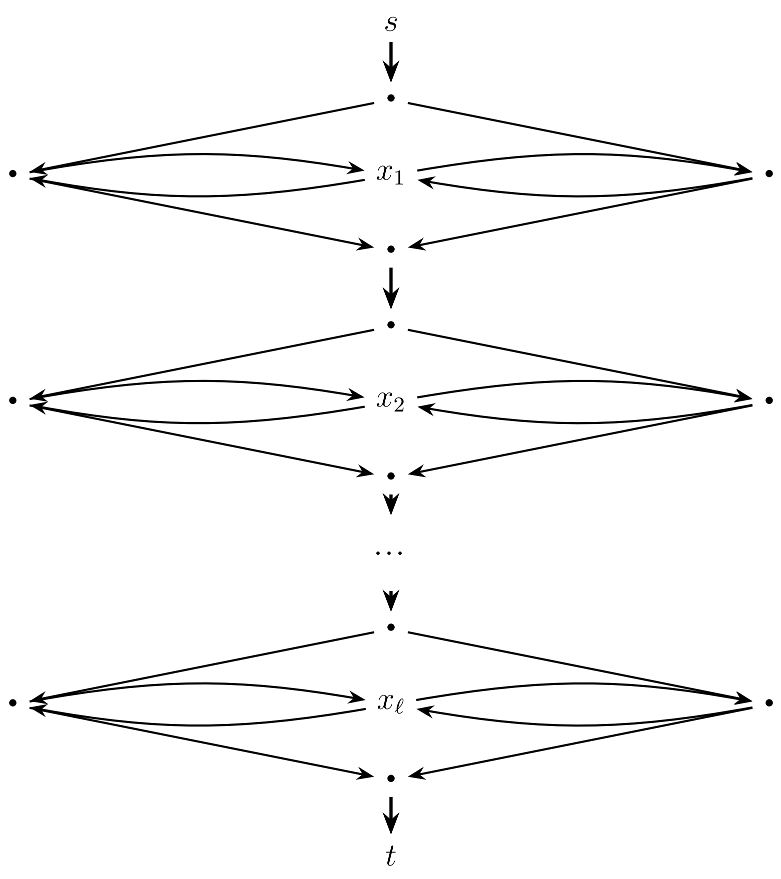
```
Let $\mathcal{V}$ ="On $\langle G = (V, E), s, t, P\rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $P$ has exactly $n = |V|$ distinct elements.
     $\mathcal{O}(n^2)$ time.
  3. For $i = 1$ to $n - 1$:
     $\mathcal{O}(n)$ iterations.
  4.     Check that $(P[i], P[i + 1]) \in E$.
     $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time per iteration, depending on your implementation.
  5. Check that $P[1] = s$ and $P[n] = t$.
     $\mathcal{O}(1)$ time.
  6. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of G.

So HAM-PATH is in $\mathcal{NP}$.

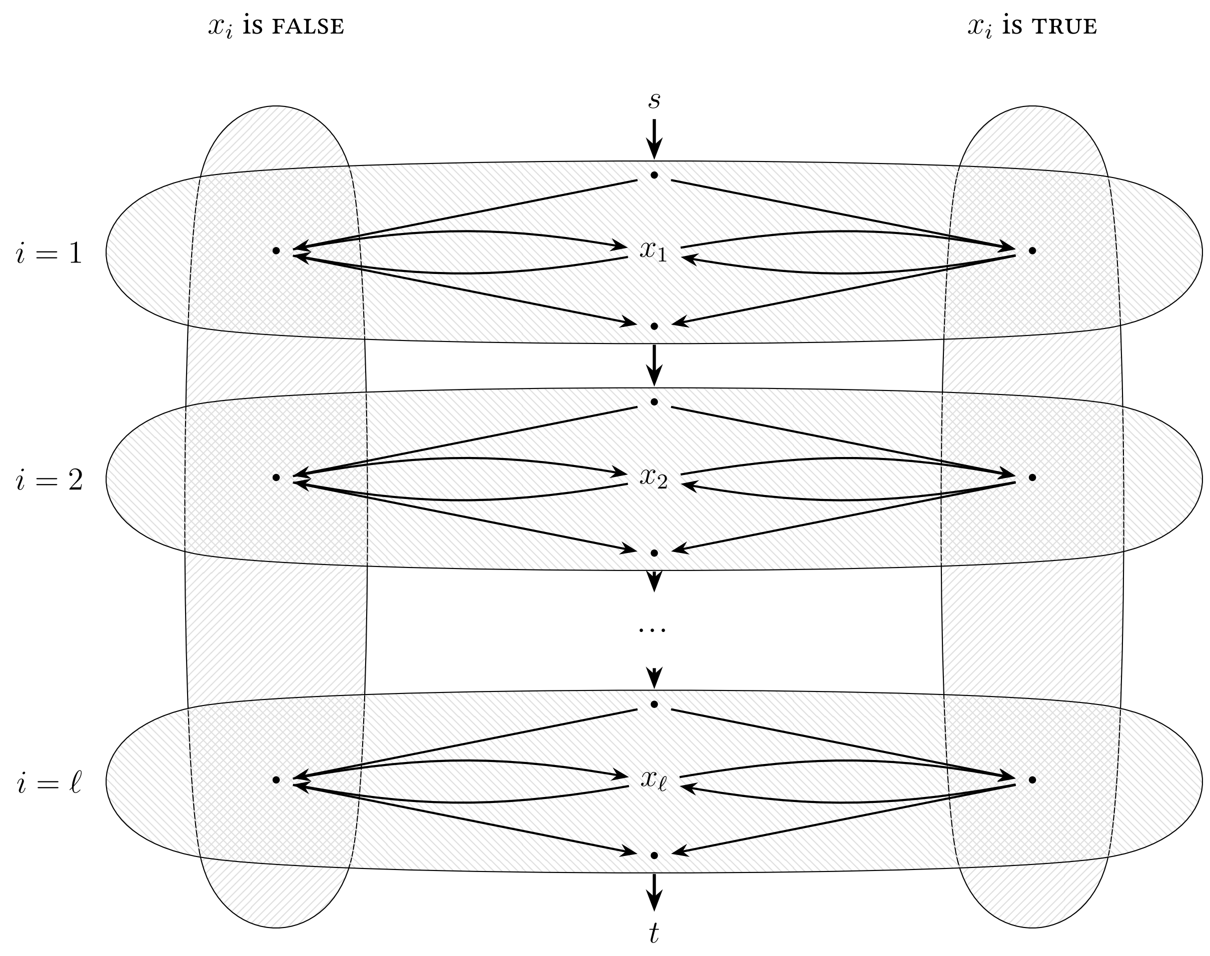
Reduction from 3SAT (adapted from (Sipser, 2013)):

Suppose we are given a 3SAT instance ϕ with ℓ variables and k clauses. Note that we can assume using 3 that the clauses in ϕ each have three distinct variables.
We'll start by encoding a choice of truth assignment in ϕ into a HAM-PATH instance.
We'll do this with the following graph:

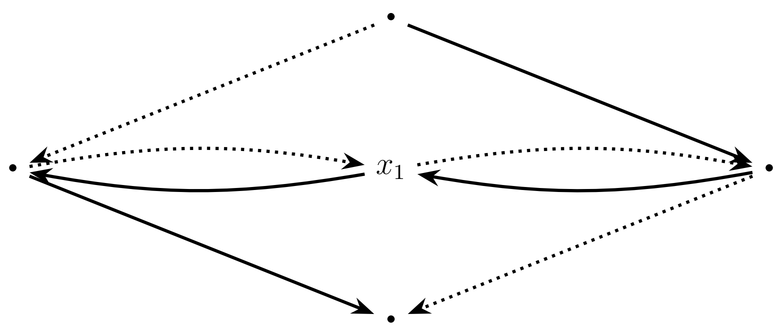


OK, so what does this do?

Well, each "level" of the graph represents a variable, and the choice of which direction you move on that variable represents whether it is set to true or false:



That is, if we expand out the top level, we can see that there are exactly two possible ways that a Hamiltonian path can pass through it:



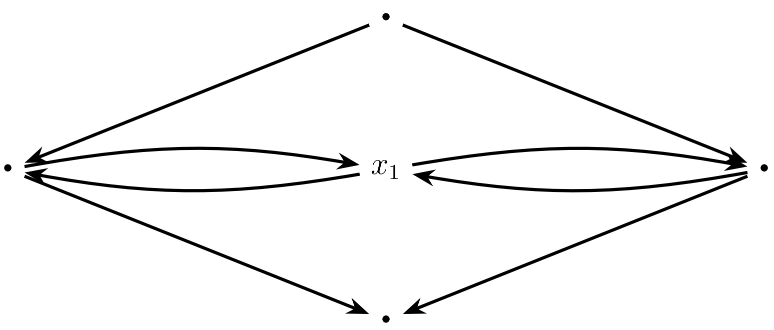
We'll treat the dotted line as an assignment of FALSE to x_1, and the solid line as an assignment of TRUE.

The same idea applies to every variable: every Hamiltonian path through our full graph corresponds to a truth assignment to our initial 3SAT fomula ϕ.

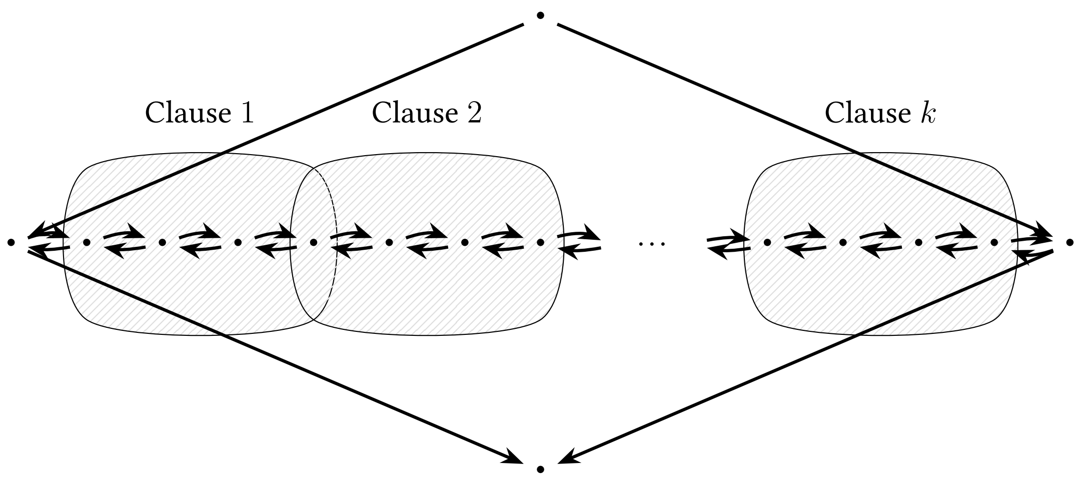
Now, we need to fill in the clauses. We'll do that by expanding the $x_1, x_2, \dots, x_\ell$ vertices in the center of the graph.

The first thing we'll do is break the x_i nodes into a series of widgets, one per clause.

For example, when we have k clauses, the first level of our graph will change from



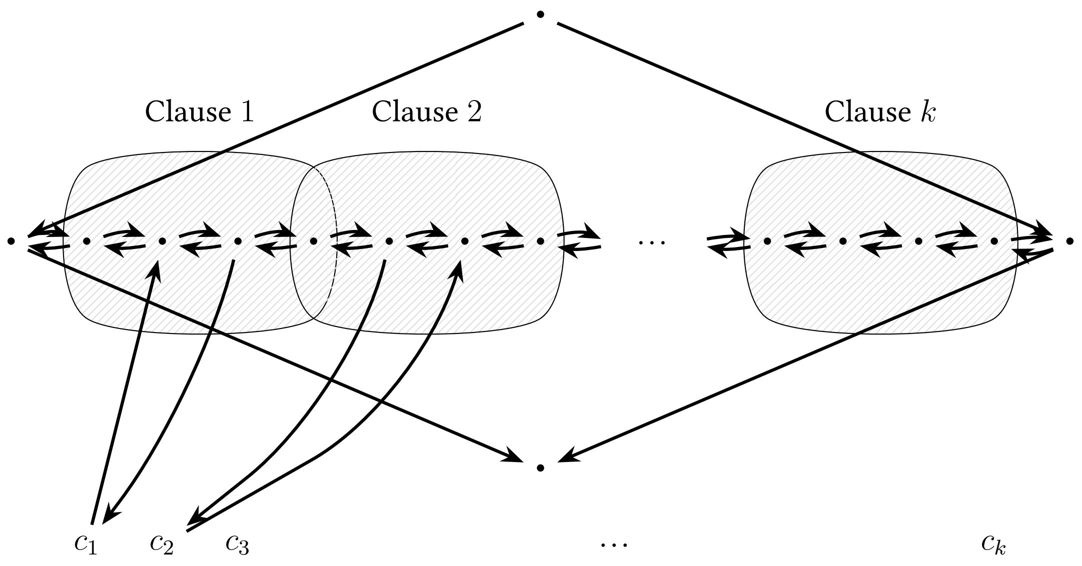
to



The shared node between clause 1 and clause 2 is the exit from one widget and the entrance to the next.

Remember that the direction of travel across the entire diamond is based on whether I have set x_1 to TRUE or FALSE: a left-to-right movement across the clauses indicates x_1 is FALSE, a right-to-left movement indicates a TRUE.

With that in mind, we'll repeat this process with every variable, and then we'll add a new set of nodes to the graph: one per clause:



There is one of each vertex $c_1, c_2, \ldots, c_k$ in the graph - they are not repeated between variables. You'll see that the clause 1 and clause 2 widgets are attached to their respective clauses. The leftward movement of the clause 1 arrows again indicates that x_1 is TRUE - in this case, we put these arrows in if and only if x_1 is in clause 1. Similarly, the rightward movement of the clause 2 arrows indicates that x_2 is FALSE - in this case, we put these arrows in if and only if $\neg x_1$ is in clause 2.

If a clause has both x_i and $\neg x_i$ in it, it will always be satisfied, so we cn safely ignore it during the reduction.

Once we've done this for every variable and every clause, we're done the reduction. As a check, you'll have thee in arrows and three out arrows from each clause vertex.

$\Rightarrow$) Now, you can see that if there is any satisfying assignment for our input ϕ, taking the corresponding direction on each of the diamond variable widgets will allow us to visit each clause - so we will be able to form a Hamiltonian path from s to t.

$\Leftarrow$ On the other hand, these are the only possible Hamiltonian paths: consider the node b in the following graph:

$$\begin{tikzcd}
a \arrow[r, bend left] & b \arrow[l, bend left] \arrow[r, bend left] & c \arrow[l, bend left]
\end{tikzcd}$$

In this subgraph, a and c may be connected to other vertices, but b may not.

If we visit the node a but do not continue on to b, there will be no way to visit b - its only other neighbour is c. If we were to use c later in order to get to b, we would be stuck at b: both of its neighbours would have already been visited.

But you can see in the reduction that every clause vertex is hedged on both sides by a subgraph of this form. If we were to try to enter a clause node from one variable and leave using another, or if we were to try to go through the clause vertex backwards, we'd end up with an unusable b, just like in the subgraph above. So in any Hamiltonian cycle, the path passing through any clause node must leave in the same variable and in the same direction as it started in. This means that every Hamiltonian path corresponds to a satisfying truth assignment for ϕ.

Finally, we see that for an ℓ-variable, k-clause ϕ, this graph has $4k + 3$ nodes per variable widget, and ℓ of these widgets. There are an additional k clause nodes, and two more nodes for s and t. So the total number of nodes is $\ell(4k+3)+k+2$. Each variable widget has $8k+4$ edges, again multiplied by ℓ, and each of the k clause nodes is an endpoint 6 additional edges. Finally, we need an additional $\ell + 1$ edges to connect the s, the t, and the variable widgets. So the total number of edges is $\ell(8k + 4) + 6k + \ell + 1$. Altogether, the number of vertices and edges in the graph are both $\mathcal{O}(\ell k)$, and each vertex and edge can be constructed using a lookup in the input ϕ. So the total reduction is polytime, as required.
So 3SAT $\leqslant_p$ HAM-PATH, and so HAM-PATH is $\mathcal{NP}$-complete.

■

4.2.6 HAM-CYCLE

Definition of HAM-CYCLE *(Hamiltonian Cycle)*:

> **Instance:** A directed graph $G = (V, E)$.
>
> **Question:** Does there exist a Hamiltonian cycle in G (i.e., a simple cycle that passes through every $v \in V$ exactly once)?

Reduced from: HAM-PATH.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is a Hamiltonian cycle C (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle G = (V, E), C\rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C$ has exactly $n = |V|$ distinct elements.
     $\mathcal{O}(n^2)$ time.
  3. For $i = 1$ to $n - 1$:
     $\mathcal{O}(n)$ iterations.
  4.     Check that $(C[i], C[i + 1]) \in E$.
     $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time per iteration, depending on your implementation.
  5. Check that $(C[n], C[1]) \in E$.
     $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time, depending on your implementation.
  6. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of G.

So HAM-CYCLE is in $\mathcal{NP}$.

Reduction from HAM-PATH:

Suppose we are given an instance $\langle G = (V, E), s, t\rangle$ of HAM-PATH.

Let $V' = V \cup \{v'\}$, where v' is a new vertex not in V. We’ll set $E' = E \cup \{(t, v'), (v', s)\}$. Let $G' = (V', E')$.

We argue that $\langle G'\rangle \in$ HAM-PATH iff $\langle G, s, t\rangle \in$ HAM-PATH:

Suppose $\langle G, s, t\rangle \in$ HAM-PATH

Then there is a Hamiltonian path $P = (v_1 = s, v_2, v_3, \ldots, v_{n-1}, v_n = t)$ in G.

But then the cycle $P = (v', v_1 = s, v_2, v_3, \ldots, v_{n-1}, v_n = t)$ encodes a Hamiltonian cycle in G'.

$\Rightarrow \langle G'\rangle \in$ HAM-CYCLE.

Suppose $\langle G' \rangle \in$ HAM-CYCLE.

Then there is a Hamiltonian cycle $C = (v', v_1, \ldots, v_n)$ in G'.

Since the only edge out of v' is (v', s), it must hold that $v_1 = s$.

Similarly, the only edge into v' is (t, v'), and so $v_n = t$.

But then $P = (s = v_1, \ldots, t = v_n)$ is a Hamiltonian path from s to t in G.

$\Rightarrow \langle G, s, t \rangle \in$ HAM-PATH.

Finally, we see that our reduction simply copies out G with the addition of one vertex and two edges. We can determine where to place this vertex and these edges in polynomial time, and so the entire reduction is polytime.

So HAM-CYCLE is $\mathcal{NP}$-complete, as required.

■

4.2.7 3COL

Definition of 3COL *(3-Colourability)*:

Instance: A graph $G = (V, E)$.

Question: Does G admit a three (vertex) colouring (i.e., a way of 3-colouring the vertices of G so that no edge has two endpoints of the same colour)?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the 3-colouring C (we can treat this as a dictionary or mapping of the vertices of V to $\{$*red, green, blue*$\}$).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle G = (V, E), C \rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C$ maps every vertex to one of three colours (we'll refer to
them as {red, green, and blue}).
     $\mathcal{O}(n)$ time.
  3. For all edges $\{u, v\} \in E$:
     $\mathcal{O}(n^2)$ iterations.
  4.    Check that $C[u] \neq C[v]$.
     $\mathcal{O}(n)$ time per iteration.
  5. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^3)$ time, and so is polytime in the size of G.

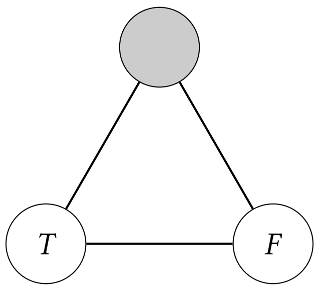
So 3COL is in $\mathcal{NP}$.

Reduction from 3SAT (adapted from an exercise in (Sipser, 2013)):

Suppose we are given a 3SAT instance ϕ with ℓ variables and k clauses.

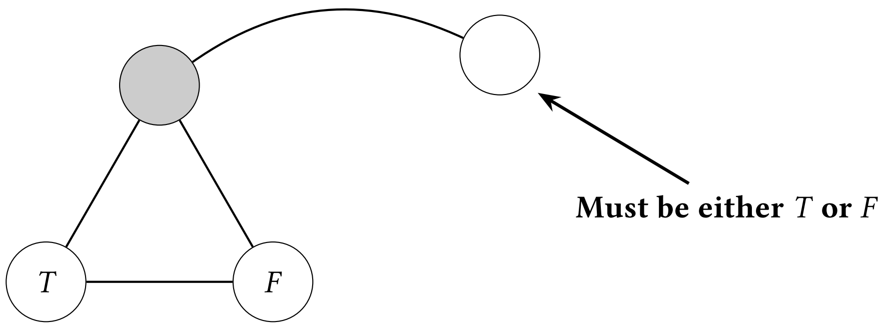
The trick here is to build graph widgets that actually encode the individual *or*s, *not*s, and *and*s from the Boolean formula.

To start, let's build a palette widget for our graph. This is actually a very simple widget: it's just a triangle.

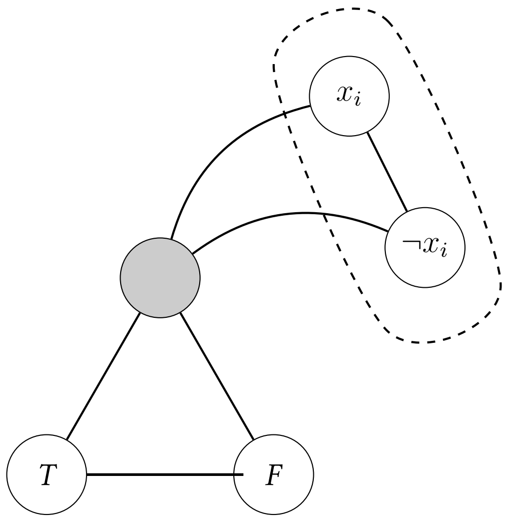


You can see that any 3-colouring of a graph with this widget must give a different one of the three colours to each of these three nodes.
Now, if we're able to 3-colour a graph in one way, we know at least five other ways to colour it as well: we can always permute the colours. So the colours aren't fixed.
But for the moment we'll treat them as if the colour are the ones we see here - we'll label them *true*, *false*, and *red*.

You can see that, by attaching the *red* palette node to any other node, we can force that other node to take one of the two Boolean values.

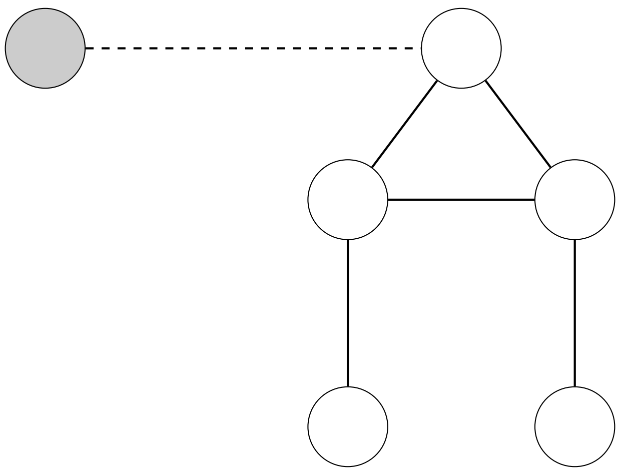


Using this trick, the first thing we want to encode are the variable assignments for ϕ. We can do this by creating a node for every variable x_i, and for its negation $\neg x_i$. If we connect both to *red*, they must take values of *true* or *false*, and if we connect them to each other they must take opposite values.

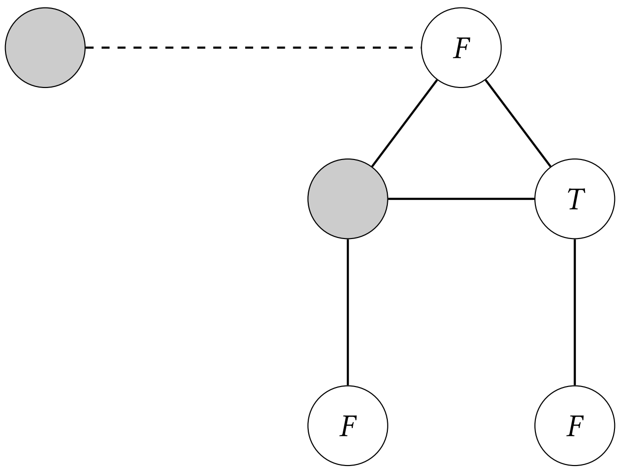


We can repeat this process for every variable.

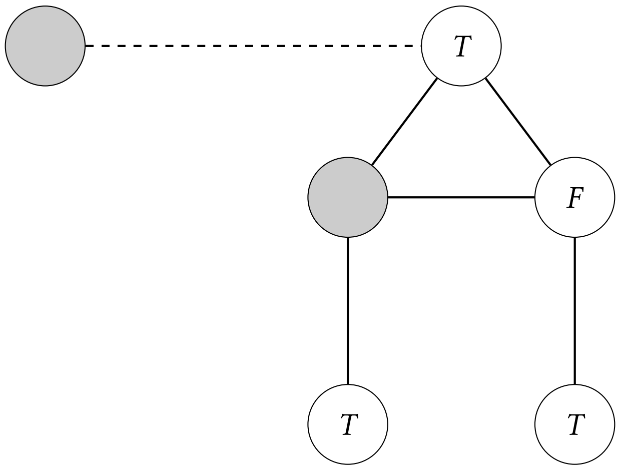
Now, we can build an *or* widget for our graph:



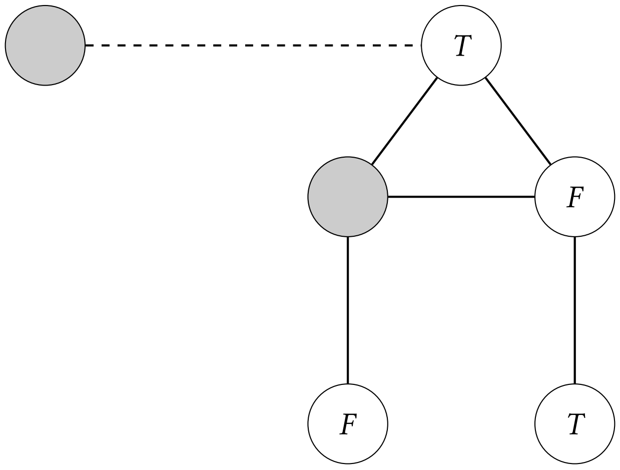
The two bottom nodes will be Boolean valued - they'll have a colour of either *true* or *false*. If we do this, you can see that the top node can take the value *true* iff one of the two bottom nodes also has a value true: if both are *false*, the only way to fill colour the widget is



(Or the mirror image.)
On the other hand, the colourings

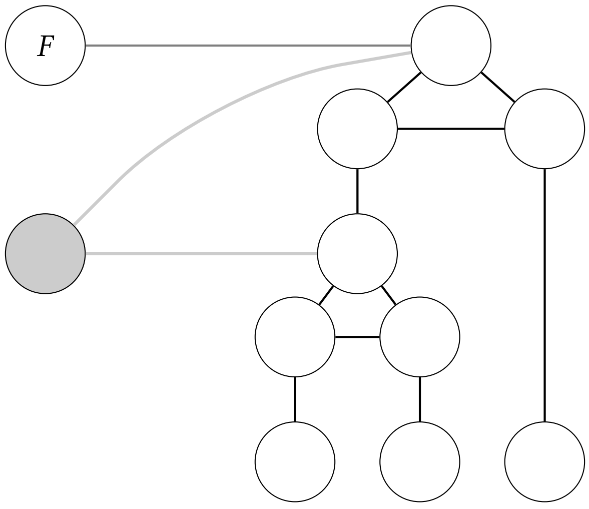


and



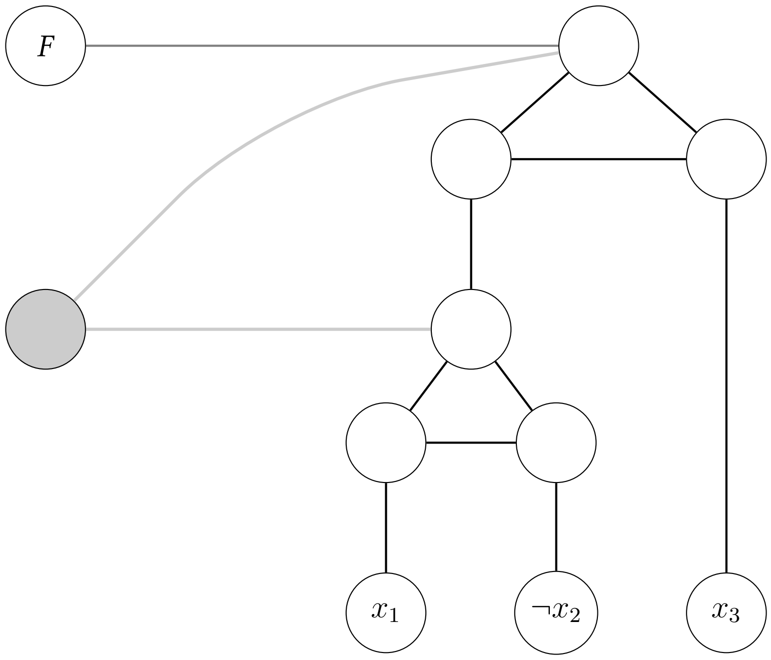
(Again, and their mirror images.)
Are all possible. So it is possible to assign a *true* to the top node if and only if at least one bottom node is *true*, as desired.

Now, for a 3CNF formula, each clause is made up of two *or* operations on some literals. We can simulate this by stacking two of these widgets on top of each other:



From our discussion of the *or*-widget, we know that the top node can be coloured *true* iff at least one of the bottom nodes is also *true*, but the fact that we have connected this node to *red* and *false* means that this is the only colouring that we can accept.

If we identify the bottom nodes with the literals in a clause, then, we will have encoded the whole clause into the 3SAT problem:



If we repeat this process for every clause, we will have our entire graph, and, by our discussion, this graph will be 3-colourable iff the input ϕ was satisfiable.

$\Rightarrow$) If ϕ has a satisfying truth assignment, then we can use the values from that assignment to fill in the inputs to the corresponding *or*-widgets. By our discussion above, these widgets can then be 3-coloured.

$\Leftarrow$) If our output graph G is 3-colourable, then we can argue wlog. that its palette widget has the *red*/T/F colouring we used above (if not, we can simply rename the colours).

Since the inputs to the *or*-widget are connected to the *red* node in the palette, they must be either T of F. We can read these values to get a truth assignment τ for ϕ. Since each *or*-widget is satisfied iff their inputs are not all F, this truth assignment must be a satisfying truth assignment for ϕ.

Finally, we can see that for an ℓ-variable, k-node ϕ, this graph requires 3 nodes for the palette, two nodes for each of the ℓ variables, and six nodes per clause.

Likewise, we need three edges for the palette, three for each variable, and thirteen for each clause.

So all told, the number of vertices and edges in this graph are both $\mathcal{O}(\ell + k)$, and we can construct the edges and vertices using simple lookups to the input ϕ. So the whole construction is polytime, as required.
So 3SAT $\leqslant_p$ 3COL, and so 3COL is $\mathcal{NP}$-complete.

■

4.2.8 PARTITION-INTO-TRIANGLES

Definition of PARTITION-INTO-TRIANGLES (Problem and reduction are from (Garey and Johnson, 1979)):

> **Instance:** A graph $G = (V, E)$.
>
> **Question:** Can G be partitioned into disjoint triangles (i.e., can the vertices in V be split into pairwise disjoint sets of three such that each set is a trinagle in G)?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the partitioning P (we can treat this as a collection of triples of vertices in V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle G = (V, E), P\rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $P$ lists each $v \in V$ exactly once in its triples, and that there are no extra vertices in $P$).
     $\mathcal{O}(n^2)$ time.
  3. For all triples $\{a, b, c\} \in P$:
     $\mathcal{O}(n)$ iterations.
  4.     Check that $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ are all edges in $E$.
     $\mathcal{O}(n)$ time per iteration.
  6.Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of G.

So PARTITION-INTO-TRIANGLES is in $\mathcal{NP}$.

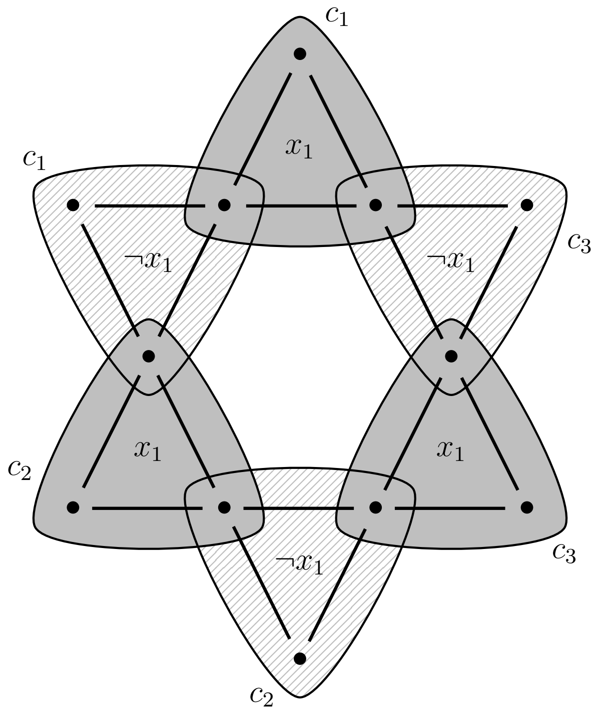
Reduction from 3SAT:

Suppose we are given a 3SAT instance ϕ with ℓ variables and k clauses. Note that we can assume using 3 that the clauses in ϕ each have three distinct variables.

Firstly, here's a widget for truth assignments. Suppose (as an example) we have a formula

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

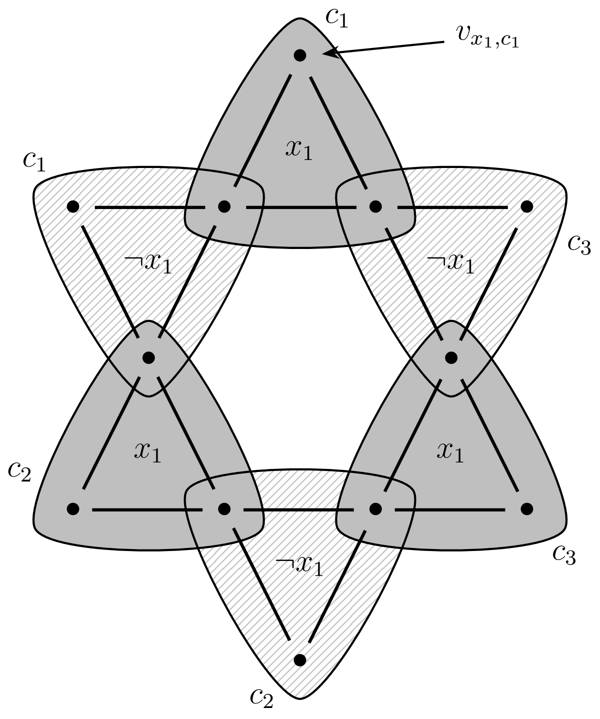
We'll create the following widget for all of the variables (this one is for x_1):



So we have two triangles for each clause in ϕ. The inner nodes don't connect to anything else, and the outer nodes can be connected. The same construction can be used for different ϕ, with the understanding that the star will have two leaves per clause in ϕ

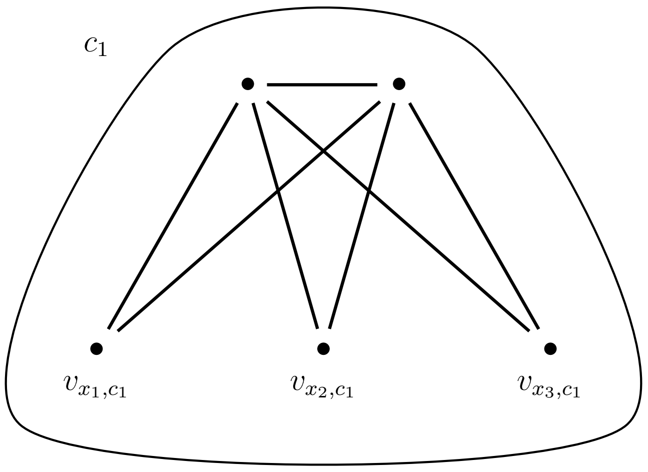
You can see that if we try to break this graph into triangles, there are two ways of doing it — we can choose the ones labeled x_1 (the solid grey), or we can choose the ones labeled $\neg x_1$ (the striped ones).

Using this widget, the star widget for x_1 will have a solid-coloured leaf for x_1 and the clause c_1. We will refer to this node as v_{x_1,c_1}:



We can similarly find the nodes $v_{\neg x_2,c_1}$ and v_{x_3,c_1} for variables x_2 and x_3.

With that in mind, our widget for the clause c_1 would be



You can see that the only way to build a triangle using the top two nodes is to connect it to one of the three nodes in the variable assignment widgets.

Now, if we have a ϕ with n clauses and k variables, our truth assignment widgets will have $2nk$ outer nodes, half of which will be covered by setting a truth assignment to the variables. We can assign n mode of these outer nodes to triangles if we can set every clause to true, leaving $(k-1)n$ of these outer nodes remaining.

We deal with these nodes by building $k(n-1)$ extra copies of the clause widgets, with the change that they connect to every outer node in our variable widget.

Once we've connected all of these widgets together we have our output graph G.
Proof that $\phi \in$ 3SAT **iff** $\langle G \rangle \in$ PARTITION-INTO-TRIANGLES:

$\Rightarrow$) Suppose that $\langle \phi \rangle \in$ 3SAT:

Then there exists some truth assignment that assigns at least one literal per clause to true.

If we use that truth assignment to choose an assignment to triangles for the variable widgets, these literals will be free to be assigned with the n clause widgets.

This will assign all of the inner nodes from the variable widgets to triangles, and will also assign all of the nodes from clause widgets to triangles. But it will leave $(k-1)n$ of the outer nodes from the variable widgets to be assigned.

By assigning these nodes to the extra widgets from end of the construction.

This will give us a decomposition of G into triangles.

$\Rightarrow \langle G \rangle \in$ PARTITION-INTO-TRIANGLES.

$\Leftarrow$) Suppose that $\langle G \rangle \in$ PARTITION-INTO-TRIANGLES:

Then There is some way of breaking G into disjoint triangles.

In this partitioning, each of the clause widgets must contribute to exactly one triangle.

The extra vertex in each of these triangles must correspond to some literal on the outside of the variable widgets.

Since the variable widget enforces consistency in the variable assignments, the literals chosen by the clause widgets are consistent.

$\Rightarrow$ These literal assignments can be extended into a consistent truth assignment.

Since this truth assignment sets every clause to true, it is a satisfying assignment for ϕ.

$\Rightarrow \langle\phi\rangle \in$ 3SAT.

Finally, we can see that the number of nodes in G is $3kn$, and that we can tell in polytime whether any pair of vertices is an edge. So the entire construction takes polynomial time.
So 3SAT $\leqslant_p$ PARTITION-INTO-TRIANGLES, and so PARTITION-INTO-TRIANGLES is $\mathcal{NP}$-complete.

■

4.2.9 CUBIC-SUBGRAPH

Definition of CUBIC-SUBGRAPH (Problem from (Garey and Johnson, 1979). Reduction inspired by (Stewart, 1994)):

INPUT: A graph $G = (V, E)$.

QUESTION: Does G have a cubic subgraph (i.e., a set $V' \subseteq V$ such that the subgraph of G containing all of the vertices in V' and all of the edges between these vertices will be cubic)?

A graph G is cubic, or 3-regular, if every vertex has a degree of exactly 3. Also note that we don't accept empty graphs.

Reduced from: 3COL.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set V' inducing the cubic subgraph (we can treat this as a list of vertices in V).
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle G = (V, E), V' \rangle$:"
  1. Reject if any of the following checks fails:
     *$\mathcal{O}(1)$ time.*
  2. Check that $V' \subseteq V$, and that no vertex ocurs more than once in it.
     *$\mathcal{O}(n^2)$ time.*
  3. For all vertices $u \in V'$:
     *$\mathcal{O}(n)$ iterations.*
  4.     *count* := 0.
     *$\mathcal{O}(1)$ time per iteration.*
```

```
5.    For $v \in V'$:
        $\mathcal{O}(n)$ iterations.
6.        If $\{u, v\} \in E$, count++.
            $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time per iteration, depending on your implementation.
7.    Check that count == 3.
    $\mathcal{O}(1)$ time per iteration.
8. Accept.
    $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ time, and so is polytime in the size of G .

So CUBIC-SUBGRAPH is in $\mathcal{NP}$.

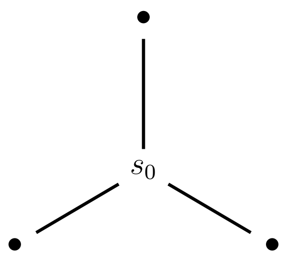
Reduction from 3COL:

Let an input graph $G = (V, E)$ be given, where G has n nodes and m edges.

We'll break our construction into two basic steps:

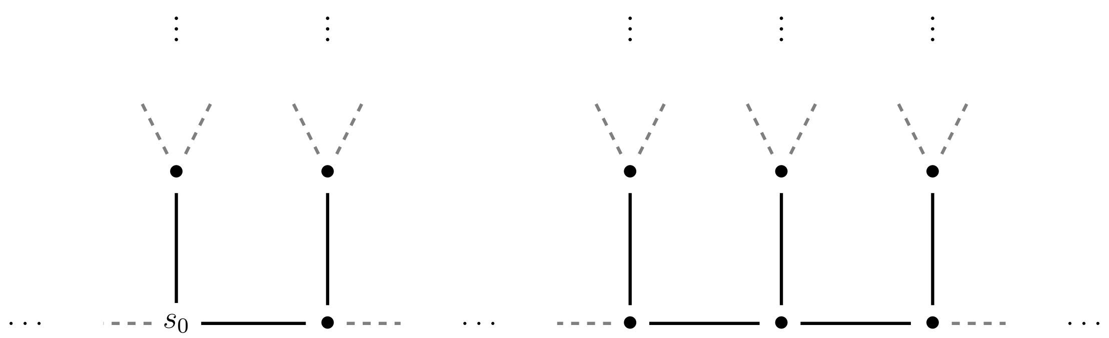
1. We'll use G to build a graph G' with a specified node s_0 such that G' contains a cubic subgraph that contains s_0 iff G has a 3-colouring.
2. We'll argue that every cubic subgraph of G' must contain this node s_0.

So to start off, let's consider a graph G' with a node s_0:



Note that s_0 has exactly three neighbours. Now, suppose that we have some cubic subgraph H' of G' that contains the node s_0. Since H' is cubic, each of its vertices – and so, in particular, the vertex s_0 – must have three neighbours in H'. Each of these neighbours must also be a neighbour in G', and since s_0 only has three such neighbours to begin with we see that each neighbour of s_0 in G' must also be in H'.

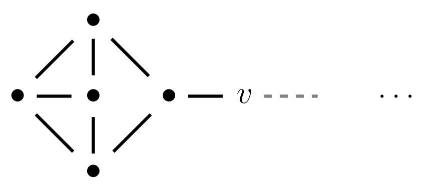
We can extend this idea. Suppose we build a whole chain of vertices from s_0 such that each vertex in the chain has three neighbours:



If we have a cubic subgraph H' of this graph that contains the vertex s_0, then H' must also contain the entire chain.

> *In general, if G' contains this pattern, then any cubic subgraph of G' either contains all or none of the chain. We can't have just part of it.*

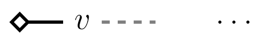
Of course, we can't really extend this sort of chain forever. We'll need to end it somewhere. There are a couple of ways of doing this – we can loop it back onto iteslf, for example. But here's a little widget that'll also let us cap off a chain (or any other patterns):



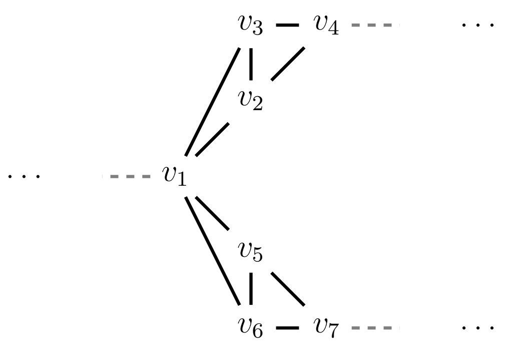
All of the vertices other than v in this subgraph have degree three – so if we had a subgraph of G' that was almost cubic, but which had one degree two vertex v, we could attach this widget to v to get a cubic subgraph.

Since every vertex other than v in this widget has degree three, and since these vertices connect only to each other (or to v), we cannot remove any one of them from a cubic subgraph without removing the entire widget. You'll see, too, that if v is not included in our subgraph, then none of the other vertices in this widget can be included, either. So this widget is all-or-nothing in terms of whether it can be included into a cubic subgraph, just like the chains we've already seen.

We'll refer to this widget as a *tag*, and we'll denote it graphically as



And these are *almost* enough for us to do our reduction! To get a reduction from 3COL we'll need just two more widgets – firstly, we'll need a widget that encodes a choice. So consider the following mechanism:



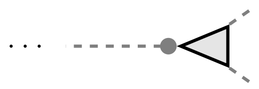
If we try to build a cubic subgraph using this widget, we see that if v_2 is included, then so are v_1, v_3, and v_4. Similarly, if v_5 is included, then so are v_1, v_6, and v_7 (we can make similar arguments for v_3 and v_6).

But this means that we cannot include both v_2 and v_5 – otherwise, v_1 would have a degree of at least four.

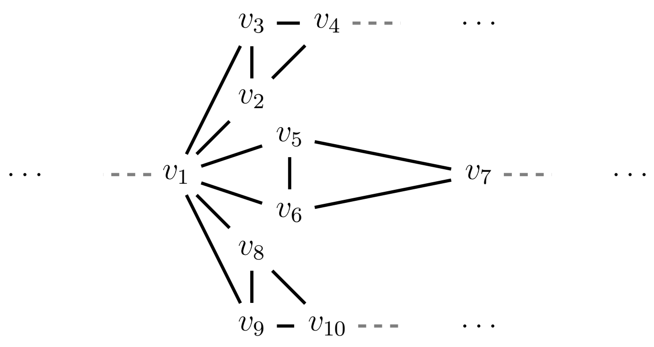
On the other hand, if v_1 is included, then it must have three neighbours. If these neighbours include the one on the left-hand boundary of the widget, we see that our cubic subgraph either contains v_2, v_3, and v_4, or it contains v_5, v_6, and v_7.

So either v_1 is not contained in our cubic subgraph (and in fact, none of these vertices is included), or v_1 is included, and this widget encodes a choice as to whether to extend our cubic subgraph through the v_4 or through the v_7 vertex.

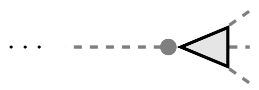
We'll denote this widget graphically as



In fact, we have as many branches as we want coming out of the widget:

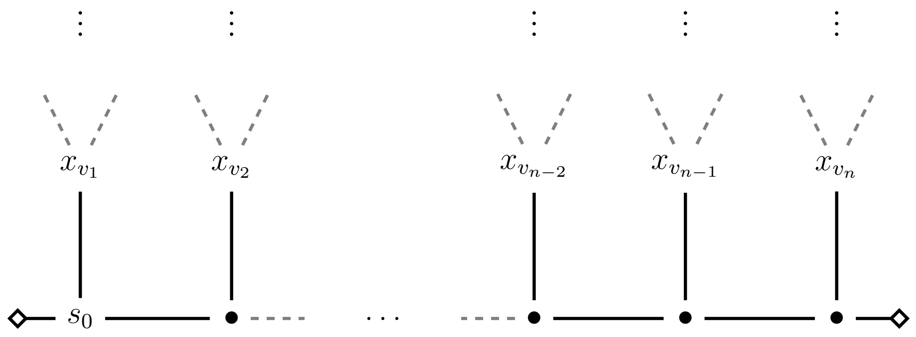


We'll denote multiple branch copies of this widget like so:



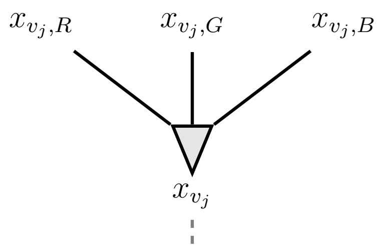
With these widgets, we are now ready to do our reduction.

If our input G has n nodes, we'll start by setting up the following widget, which we'll denote as X_G:



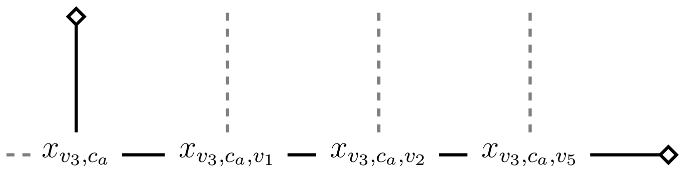
Note that X_G contains the vertex s_0, and so we'll start by looking only at cubic subgraphs that contain this vertex. From our previous discussion, we know that if G' contains X_G then any cubic subgraph of G' must either contain all of X_G or none of it. Since we insist that our cubic subgraph must include s_0, we'll have subgraphs that contain all of X_G. Given how we set up the rest of the graph, it will turn out that every cubic subgraph will be of this form.

You can see that X_G has n nodes named x_{v_j}. There's one per node in G, and we'll extend G' at each x_{v_j} to encode the choice of colour that we make for the vertex v_j. Each such extension, which we'll refer to as a vertex widget, will be written as X_{v_j}. These widgets will each have a base that is made up of a 3-choice branch, one per colour:



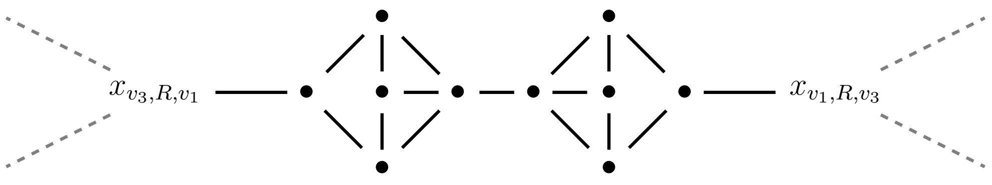
From our discussion about the choice widgets, we know that if any node in these widgets in included in a cubic subgraph of G', then the node x_{v_j} must be in that subgraph. Furthermore, if x_{v_j} is included, then exactly one of the three $x_{v_j,R}$, $x_{v_j,G}$, or $x_{v_j,B}$ branches in included. Since the x_{v_j} vertices are common to both the X_G subgraph and the X_{v_j} widgets, any cubic subgraph of G' either does not contain X_G or any of the X_{v_j} vertices, or it contains all of these widgets, and all of the X_{v_j} widgets encode a colour choice as described above.

For each vertex $v_j \in G$ we will connect the nodes $x_{v_j,c_a}, a \in \{R, G, B\}$, to chains whose length matches the number of neighbours of v_j in G. For example, if v_3 were adjacent to v_1, v_2, and v_5 then each x_{v_3,c_a} would be connected to chain



Note that there is a separate chain for each colour.

Now, we only have to finish the connections for the vertices x_{v_j,c_a,v_i}. The way we'll do this is to attach each of these vertices to a modified tag – for every edge $\{v_i, v_j\}$ in G we'll add a tag to each of the two nodes x_{v_j,c_a,v_i} and x_{v_i,c_a,v_j}, but we'll connect the tops of the two tags. So, for example, if v_3 and v_1 are connected in G we'd add the nodes



We can see that any cubic cubgraph must contain either one of these tags, or the other, or neither, but not both.

> *In particular, note that we still cannot have part of either of these tags in a cubic subgraph without including the entire tag.*

This completes our construction of the graph G'.

Claim: There is no cubic subgraph of G' that does not contain the node s_0.

Proof:

Suppose that we had a cubic subgraph H' of G' that did not contain the node s_0.

Then H' could not contain any node in X_G.

In particular, H' could not contain any of the nodes x_{v_j}.

But then H' would also not contain any node from the X_{v_j} widgets: we would be unble to use any of the nodes in the choice widgets at the start of the X_{v_j}, and so we would also not be able to include any of the connected chains or tags.

But there are no other nodes in G' that we could include in H', and so we would have a subgraph H' that contained no vertices, which we do not allow.

So we cannot have a cubic subgraph H' of G' unless said subgraph contains the node s_0.

We argue that $\langle G \rangle \in$ 3COL iff $\langle G' \rangle \in$ CUBIC-SUBGRAPH.

Proof that $\langle G \rangle \in$ 3COL iff $\langle G' \rangle \in$ CUBIC-SUBGRAPH:

Suppose that $\langle G \rangle \in$ 3COL:

Then there is some valid 3-colouring $c : V \mapsto \{R, G, B\}$ of G.

We'll build a cubic subgraph H' of G' by starting with the base widget X_G.

For every vertex $v_j \in G$, we extend H' at the node x_{v_j} by including the branch of the associated choice widget corresponding to $x_{v_j,c(v_j)}$, as well as the chain that connects to this branch.

Now, by the construction of this H', we only choose one branch per X_{v_j} widget.

So by our above discussion of the chain, tag, and choice widgets, we see that our H' cannot have any vertex of degree other than three unless it is in a point which two widgets X_{v_j} and X_{v_i} connect.

For such a connection to occur, we would need to have two vertices in H' of the form $x_{v_j,c(v_j),v_i}$ and $x_{v_i,c(v_i),v_j}$, where $\{v_i, v_j\}$ is an edge of G and $c(v_i) = c(v_j)$.

However, since c is a valid 3-colouring of G, we cannot have any connection in H', and so H' is a cubic subgraph, as required.

So $G' \in$ CUBIC-SUBGRAPH.

Suppose that $\langle G' \rangle \in$ CUBIC-SUBGRAPH:

Then it has a cubic subgraph G'.

As we have noted in our claim, G' must contain s_0, and indeed all of X_G.

Since the entire X_G subgraph is in H', each of the x_{v_j} vertices is included, too.

But then, each X_{v_j} widget is included, as well (up to some choice of branches). So for each v_j, exctly one of the x_{v_j,c_a} nodes is in H'.

We can read off these c_a to find a 3-colouring $c : V \mapsto \{R, G, B\}$ of G.

Now, suppose that we had two vertices v_i nd v_j in G such that $\{v_i, v_j\}$ were an edge of G but such that $c(v_i) = c_{v_j}$.

Since $c(v_i) = c_{v_j}$, let's write $\hat{c} = c(v_i) = c_{v_j}$. Then H' would contain the nodes $x_{v_i,\hat{c}}$ and $x_{v_j,\hat{c}}$.

But since both of these nodes are prt of a chain in G', both of these chains must be in H' as well.

In particular, the nodes $x_{v_i,\hat{c},x_j}$ and $x_{v_j,\hat{c},x_i}$ would have to be in H' (note that these nodes must exist since $\{v_i, v_j\}$ is an edge in G).

But then the tags connected to both $x_{v_i,\hat{c},x_j}$ and $x_{v_j,\hat{c},x_i}$ would hve to be included as well. However, if both of these tags were included then H' would have at least two nodes of degree four, a contradition.

So c cannot map ny two adjacent nodes in G to the same colour, and so $\langle G \rangle \in$ 3COL, as required.

Finally, we see that if G has n nodes and m edges, then:

- G' has one X_G widget, which has $2n + 10$ vertices.
- G' has n vertex widgets, each of which has 9 vertices in its colour choice section nd which have in total $33n + 12m$ vertices in the three connected chains and tags.

So all told, G' will have no more than $44n + 12m + 10$ nodes.

We can tell in polytime whether any pair of these nodes has an edge, and so this whole construction is polytime, as required.
So 3COL $\leqslant_p$ CUBIC-SUBGRAPH, and so CUBIC-SUBGRAPH is $\mathcal{NP}$-complete.

■

4.3 Sets, Partitions, and Orderings

4.3.1 SUBSET-SUM

Definition of SUBSET-SUM:

Input: A multiset S of non-negative integers, an integer t.

Question: Is there a subset $S' \subseteq S$ such that $\sum\limits_{x \in S'} = t$?

Reduced from: 3SAT.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972). In that paper it is under the name KNAPSACK, *but the actual definition is essentially what we have here – the usual version of* KNAPSACK *that we see today is a bit different.*

Also: This problem is still $\mathcal{NP}$-complete even if S is a proper set, as opposed to a multiset (i.e., it has no repititions). Our reduction here doesn't prove this (it outputs a multiset), but our reduction in the parsimonious reductions chapter does, in that it outputs a set.
Proof that SUBSET-SUM is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) A certificate would be the subset S'.
3) Since $S' \subseteq S$, and since S is listed in the input, the certificate is polynomial in the size of the input.
4) The verifier would be:

```
$\mathcal{V}$ = “On input $\langle S, t, S' \rangle$:
    1. Check that $S' \subseteq S$. If it is not, *reject.*
       $\mathcal{O}(|\langle S \rangle|^2)$ *time, where* $|\langle S \rangle|$ *is the space need to write the elements of* $S$.
    2. Let $x = 0$.
       $\mathcal{O}(1)$ *time.*
    3. For $y \in S'$:
       $\mathcal{O}(|\langle S \rangle|)$ *iterations.*
    4.     $x += y$.
       $\mathcal{O}(|\langle S \rangle|)$ *time per iteration.*
    5. If $x == t$, *accept*, otherwise *reject.*
       $\mathcal{O}(|\langle S \rangle| + \log t)$ *time.*
```

5) We can see that this verifier runs in $\mathcal{O}(|\langle S \rangle|^2 + \log t)$ time, and so is polytime in the size of $\langle S, t \rangle$.

Proof that 3SAT $\leqslant_p$ SUBSET-SUM (adapted from (Sipser, 2013)):

Given an input 3CNF ϕ, we'll write the variables of ϕ as $x_1, \ldots, x_\ell$ and its clauses as $c_1, \ldots, c_k$. Note that we can assume using 3 that the clauses in ϕ each have three distinct variables.

Idea: Let's start by setting up a table like so:

	1	2	3	…	ℓ	
x_1	1	0	0	…	0	⟨ something ⟩
$\neg x_1$	1	0	0	…	0	⟨ something ⟩
x_2	0	1	0	…	0	⟨ something ⟩
$\neg x_2$	0	1	0	…	0	⟨ something ⟩
x_3	0	0	1	…	0	⟨ something ⟩
$\neg x_3$	0	0	1	…	0	⟨ something ⟩
⋮				⋮		⋮
x_ℓ	0	0	0	…	1	⟨ something ⟩
$\neg x_\ell$	0	0	0	…	1	⟨ something ⟩
	1	1	1	…	1	⟨ something ⟩

So two rows per variable.

Suppose that we want to choose a set of rows whose columns add up to the numbers on the bottom. Then, for any x_i, we would have to choose either the row headed by x_i or the one headed by $\overline{x_i}$. We wouldn't be able to choose both.

This is a bit like forcing every boolean variable x_i to be either true or false - the rows we'd choose above correspond to truth assignments!

Let's continue with this idea: can we tell if a clause is satisfied by a truth assignment? We've left some columns in the table, so let's use them. Suppose, for example, that c_1 is $(x_1 \vee \neg x_2 \vee x_3)$. We can encode this in the first column on the right like so:

	1	2	3	…	ℓ	c_1	…
x_1	1	0	0	…	0	1	…
$\neg x_1$	1	0	0	…	0	0	…
x_2	0	1	0	…	0	0	…
$\neg x_2$	0	1	0	…	0	1	…
x_3	0	0	1	…	0	1	…
$\neg x_3$	0	0	1	…	0	0	…
⋮				⋮			⋮
x_ℓ	0	0	0	…	1	0	…
$\neg x_\ell$	0	0	0	…	1	0	…
	1	1	1	…	1	?	…

That is, we've added a 1 in this column to every row corresponding to a literal in c_1.

Clearly, the sum of the c_1 column is between 1 and 3 if we choose a truth assignment that satisfies c_1.

What's more, we can repeat the same process for every clause:

	1	2	3	$\dots$	ℓ	c_1	c_2	$\dots$	c_k
x_1	1	0	0	$\dots$	0	1	0	$\dots$	0
$\neg x_1$	1	0	0	$\dots$	0	0	0	$\dots$	0
x_2	0	1	0	$\dots$	0	0	1	$\dots$	0
$\neg x_2$	0	1	0	$\dots$	0	1	0	$\dots$	0
x_3	0	0	1	$\dots$	0	1	0	$\dots$	0
$\neg x_3$	0	0	1	$\dots$	0	0	1	$\dots$	1
$\vdots$				$\vdots$				$\vdots$	
x_ℓ	0	0	0	$\dots$	1	0	0	$\dots$	1
$\neg x_\ell$	0	0	0	$\dots$	1	0	0	$\dots$	0
	1	1	1	$\dots$	1	?	?	$\dots$	?

So we've made a table where any satisfying truth assignment to ϕ gives us a 1 in the first set of columns and something between 1 and 3 in the second set of columns. Moreover, if there is no satisfying assignment we'll never be able to satisfy every clause, so at least one clause from the second column will always add up to 0.

If we add a couple of dummy variables to every clause column, we can make them add up to 3 when there's a satisfying assignment:

	1	2	3	$\dots$	ℓ	c_1	c_2	$\dots$	c_k
x_1	1	0	0	$\dots$	0	1	0	$\dots$	0
$\neg x_1$	1	0	0	$\dots$	0	0	0	$\dots$	0
x_2	0	1	0	$\dots$	0	0	1	$\dots$	0
$\neg x_2$	0	1	0	$\dots$	0	1	0	$\dots$	0
x_3	0	0	1	$\dots$	0	1	0	$\dots$	0
$\neg x_3$	0	0	1	$\dots$	0	0	1	$\dots$	1
$\vdots$				$\vdots$				$\vdots$	
x_ℓ	0	0	0	$\dots$	1	0	0	$\dots$	1
$\neg x_\ell$	0	0	0	$\dots$	1	0	0	$\dots$	0
d_1						1	0	$\dots$	0
d'_1						1	0	$\dots$	0
d_2						0	1	$\dots$	0
d'_2						0	1	$\dots$	0
$\vdots$								$\vdots$	
d_k						0	0	$\dots$	1
d'_k						0	0	$\dots$	1
t	1	1	1	$\dots$	1	3	3	$\dots$	3

So we can choose a set of rows in this table whose columns add up to the bottom row iff ϕ has a satisfying assignment.

To turn this into a SUBSET-SUM instance, just note that the numbers in the columns can never add up to more than 3 – so we can treat the rows as numbers in base 10 and never have to worry about one digit affecting another. So our S will just be the non-bottom rows in the table, and t will be the bottom row.

We can see that this construction will take polynomial time:

> The total table size is $(\ell + k)^2$, which is clearly polynomial in the size of ϕ. Each entry in the table is a 0, a 1, or a 3, and each value can be found with at worst a polynomial

time lookup. The string value of t, $\langle t\rangle$ is $1^{\ell}3^{k}$, which can also be found in polynomial time. So all told, this construction takes polynomial time.

And we've already given an overview of our justification as to why it works.

So 3SAT $\leqslant_p$ SUBSET-SUM, and so SUBSET-SUM is $\mathcal{NP}$-complete.

■

4.3.2 3DM

Definition of 3DM (3-Direct Matching: Problem from (Garey and Johnson, 1979)):

Instance: Three sets A, B, and C, where $|A| = |B| = |C|$, and a set $T \subseteq A \times B \times C$.

Question: Is there a subset $M \subseteq T$ such that:

i) $|M| = |A|$,
ii) For any distinct (a, b, c) and $(a', b', c') \in M, a \neq a', b \neq b'$, and $c \neq c'$?

Reduced from: 3SAT.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set M.
3) Since $M \subseteq T$, and T is part of the input, M is polynomial in the size of the input.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle A, B, C, T, M\rangle$:”
    1. Reject if any of the following checks fails:
       $\mathcal{O}(1)$ *time.*
    2. Check that $M \subseteq T$.
       $\mathcal{O}(|A|^6)$ *time.*
    3. Check that $|M| = |A|$.
       $\mathcal{O}(|A|)$ *time.*
    4. Check that for any distinct $(a, b, c)$ and $(a', b', c') \in M, a \neq a', b \neq b'$,
and $c \neq c'$.
       $\mathcal{O}(|A|^2)$ *iterations.*
    5. *Accept.*
       $\mathcal{O}(1)$ *time.*
```

5) We can see that this verifier runs in $\mathcal{O}(|A|^6)$ time, and so is polytime in the size of $\langle A, B, C, T\rangle$.

So 3DM is in $\mathcal{NP}$.

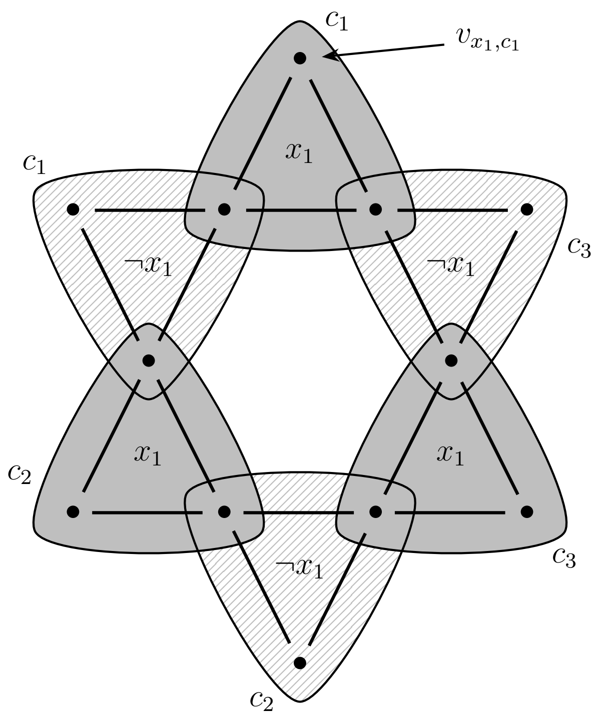
We'll use an interesting trick for this reduction: Note that the 3DM *problem is closely related to the* PARTITION-INTO-TRIANGLES *problem, in that they both relate to partitioning a set into pre-defined triples. In principle we'd like to reduce from one to the other, but that's not so easy to do here – it's not clear how to split the vertices in a graph G into the sets A, B, and C, but that's what we'd need to do for a reduction from* PARTITION-INTO-TRIANGLES *into* 3DM.

On the other hand, we can easily split up the vertices in this way for the specific type of graph we get from the reduction into PARTITION-INTO-TRIANGLES. *This means that we can use essentially the same reduction for each problem. In fact, we build the same graph G in both reductions – the difference is that in the* PARTITION-INTO-TRIANGLES *reduction we directly return the graph G, but for the* 3DM *problem we label the vertices of G, and return the sets of labels for the A, B, and C. The T is just a list of the triangles in G. So the process is just a matter of finding the right labels for the vertices in the* PARTITION-INTO-TRIANGLES *reduction graph G.*

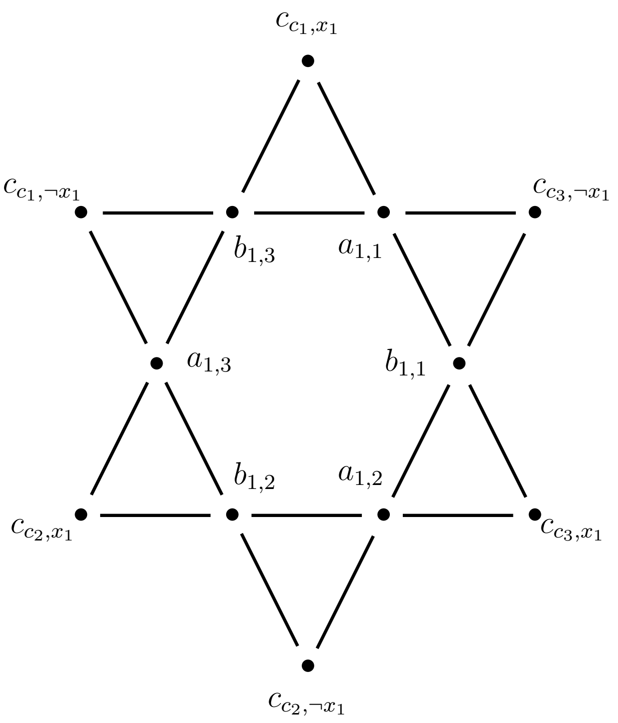
Suppose we have an input 3CNF ϕ. We'll assume using 3 that this ϕ has three distinct variables per clause (i.e., no $(x \lor x \lor y)$ type clauses).

Now, consider the PARTITION-INTO-TRIANGLES reduction from this chapter. Let G be the graph we get from that reduction. Now, G is made out of variable widgets, clause widgets, and garbage collection widgets.

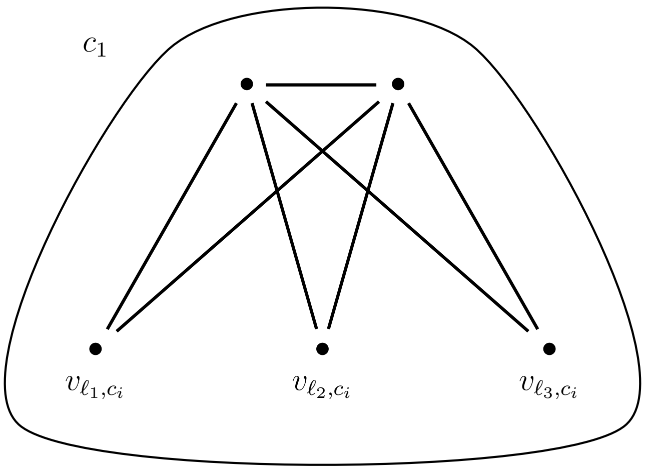
Recall that the variable widgets will be built with one widget per variable, and each widget will be a star with $2n$ points, where n is the number of clauses in ϕ. So for a 3-clause ϕ, the widget for x_1 would be of the form:



What we want to do is label each of the vertices. So we'll give the vertices these labels:

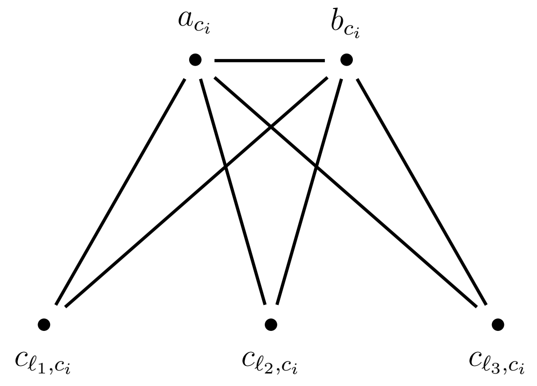


Recall that the variable v_{x_1,c_1} is a variable that remains free if we choose $x_1 = T$, while the other variable is taken. The same applies for the other clauses and variables. With this in mind, recall that in the PARTITION-INTO-TRIANGLES reduction, for the clause $c_i = (\ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3})$, where ℓ_{i_1}, ℓ_{i_2}, and ℓ_{i_3} are the literals in c_i, we produced the following widget:



So now we want to label these nodes. The labels we give are as follows:

Remember that the vertices on the bottom row are also part of the variable widgets, so they're already labelled!



Finally, we have to label the nodes from the garbage collection widgets. But the garbage collection widgets are just $k(n-1)$ copies of the clause widgets that connect to every v_{x_i,c_j} and $v_{\neg x_i,c_j}$ node, and so we label them in the same way.

If we assign all of the a labels to the set A, the b labels to the set B, and the c labels to the set C, then we can see that $|A| = |B| = |C| = 2nk$, where n is the number of clauses in ϕ and k is the number of variables. Furthermore, every triangle in G contains exactly one of each type of label (has a triple of labels in $A \times B \times C$). So we can set T to be a list of all of the triangles in G.

Note that we can find T in polynomial time by looping over all triples of vertices!

Clearly, any partitioning of G into disjoint triangles will induce a subset $M \subseteq T$ that satisfies properties i) and ii) in the 3DM problem defintion, and vice-versa.

So $\langle G \rangle$ is in PARTITION-INTO-TRIANGLES iff $\langle A, B, C, T \rangle$ is in 3DM, and therefore $\langle A, B, C, T \rangle \in$ 3DM iff $\langle \phi \rangle \in$ 3SAT.

We refer the reader to the PARTITION-INTO-TRIANGLES *reduction for the argument that G can be partitioned into triangles iff ϕ has a satisfying truth assignment.*

Finally, we can see that this reduction is polytime: the construction of G is polytime in the size of ϕ, as described in the PARTITION-INTO-TRIANGLES reduction. The label sets A, B, and C are simply labels of the vertices of G, and can be efficiently computed. Finally, the set T can be found in polytime by looping over the triples of vertices in G, and so can also be found in polytime. So the entire construction is polytime, as required.

■

References

Shimon Even, Alon Itai, and Adi Shamir. On the complexity of time table and multi-commodity flow problems. In *16th annual symposium on foundations of computer science (sfcs 1975)*, pages 184–193. IEEE, 1975.

Michael R Garey and David S Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* WH Freeman, 1979.

R KARP. Reducibility among combinatorial problems. *Complexity of Computer Computation*, pages 85–104, 1972.

Jaroslav Opatrny. Total ordering problem. *SIAM Journal on Computing*, 8(1):111–114, 1979.

M Sipser. Introduction to the theory of computation. 3th. *Cengage Learning*, 2013.

Iain A Stewart. Deciding whether a planar graph has a cubic subgraph is np-complete. *Discrete Mathematics*, 126(1-3):349–357, 1994.

Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. *preprint*, 1996.