CSCC63 – Week 8

This week: We'll look a little more at the languages in $\mathcal{NP}$

Polytime Reductions, Continued

Last week we showed that 3COL, CUBIC-SUBGRAPH, and CLIQUE are $\mathcal{NP}$-complete. But we didn't get around to SUBSET-SUM, so let's have a look at that now.

SUBSET-SUM is $\mathcal{NP}$-complete

Idea: We've seen a few different graph-based languages, but there are many different sorts of $\mathcal{NP}$-complete languages. This is an example of a numerical partition based language:

Let's define SUBSET-SUM as:

> **Input:** A multiset S of non-negative integers, an integer t.
> **Question:** Is there a subset $S' \subseteq S$ such that $\sum_{x \in S'} = t$?

1. Proof that Subset-Sum is in 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, $|S'| \leqslant |S| \leqslant |I|$, and so thie certificate is polynomial in the size of the input.
   4. The verifier would be:
      $V =$ "On input $\langle S, t, S' \rangle$:
      1. Check that $S' \subseteq S$. If it is not, *reject*.
      2. Let $x = 0$.
      3. For $y \in S'$:
      4. $\quad x+ = y$.
      5. If $x == t$, *accept*, otherwise *reject*.
   5. Line 1 can be done in at worst quadratic time, and lines 2-4 take linear time. So the entire algorithm takes polynomial time.

2. Proof that 3SAT $\leqslant_p$ SUBSET-SUM:

Recall the idea of a reduction — given a formula ϕ in 3CNF, we want to output an S and t such that S has a subset summing to t iff $\phi \in$ 3SAT.

> **Note**: This is a reduction that a lot of students find tricky at first — so don't panic if it seems difficult!
>
> We'll look more at this reduction soon, though, and that should help clear things up. So make sure you can follow the basic steps — it'll make the later ideas easier to follow.

So, given an input 3CNF ϕ, we'll write the variables of ϕ as $x_1, \ldots, x_\ell$ and its clauses as $c_1, \ldots, c_k$.

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

	1	2	3	...	ℓ	
x_1	1	0	0	...	0	⟨ something ⟩
$\overline{x_1}$	1	0	0	...	0	⟨ something ⟩
x_2	0	1	0	...	0	⟨ something ⟩
$\overline{x_2}$	0	1	0	...	0	⟨ something ⟩
x_3	0	0	1	...	0	⟨ something ⟩
$\overline{x_3}$	0	0	1	...	0	⟨ something ⟩
⋮				⋮		⋮
x_ℓ	0	0	0	...	1	⟨ something ⟩
$\overline{x_\ell}$	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 $(x_1 \vee \overline{x_2} \vee x_3)$. 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	...
$\overline{x_1}$	1	0	0	...	0	0	...
x_2	0	1	0	...	0	0	...
$\overline{x_2}$	0	1	0	...	0	1	...
x_3	0	0	1	...	0	1	...
$\overline{x_3}$	0	0	1	...	0	0	...
⋮				⋮			⋮
x_ℓ	0	0	0	...	1	0	...
$\overline{x_\ell}$	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	...	ℓ	c_1	c_2	...	c_k
x_1	1	0	0	...	0	1	0	...	0
$\overline{x_1}$	1	0	0	...	0	0	0	...	0
x_2	0	1	0	...	0	0	1	...	0
$\overline{x_2}$	0	1	0	...	0	1	0	...	0
x_3	0	0	1	...	0	1	0	...	0
$\overline{x_3}$	0	0	1	...	0	0	1	...	1
$\vdots$				$\vdots$				$\vdots$	
x_ℓ	0	0	0	...	1	0	0	...	1
$\overline{x_\ell}$	0	0	0	...	1	0	0	...	0
	1	1	1	...	1	?	?	...	?

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	...	ℓ	c_1	c_2	...	c_k
x_1	1	0	0	...	0	1	0	...	0
$\overline{x_1}$	1	0	0	...	0	0	0	...	0
x_2	0	1	0	...	0	0	1	...	0
$\overline{x_2}$	0	1	0	...	0	1	0	...	0
x_3	0	0	1	...	0	1	0	...	0
$\overline{x_3}$	0	0	1	...	0	0	1	...	1
$\vdots$				$\vdots$				$\vdots$	
x_ℓ	0	0	0	...	1	0	0	...	1
$\overline{x_\ell}$	0	0	0	...	1	0	0	...	0
d_1						1	0	...	0
d'_1						1	0	...	0
d_2						0	1	...	0
d'_2						0	1	...	0
$\vdots$								$\vdots$	
d_k						0	0	...	1
d'_k						0	0	...	1
t	1	1	1	...	1	3	3	...	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.

Here are some other NP-complete problems. The proof of membership in $\mathcal{NP}$ and some proof details can also be found in the Polytime Reductions handout.

INDEPENDENT-SET (IS)

> **INPUT:** A graph G and a number $k \in \mathbb{N}$.
>
> **QUESTION:** Does G have an independent set of size k?
> *An independent set S is a collection of vertices in G such that for all $u, v \in S$, there is no edge between u and v.*

Reduction: CLIQUE $\leqslant_p$ INDEPENDENT-SET.

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

We return the pair $\langle G' = (V, \overline{E}), k \rangle$, where $\overline{E} = \{\{u, v\} \mid u \neq v \in V \wedge \{u, v\} \notin E\}$. *(Can you see why this is polytime?)*

Proof that $\langle G, k \rangle \in$ CLIQUE iff $\langle G', k \rangle \in$ INDEPENDENT-SET:

Suppose G has a size-k clique C:

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

  Then for any $u \neq v \in C$, $\{u, v\} \notin \overline{E}$ in G'.

  $\Rightarrow C$ is a size-k independent set in G'.

Suppose G' has a size-k independent set I:

- Then for any $u \neq v \in I$, $\{u, v\} \notin \overline{E}$ in G'.

  Then for any $u \neq v \in I$, $\{u, v\} \in E$ in G.

  $\Rightarrow I$ is a size-k clique in G.

VERTEX-COVER (VC)

INPUT: A graph G and a number $k \in \mathbb{N}$.

QUESTION: Does G have a vertex cover of size k?
A vertex cover S is a collection of vertices in G such that for all edges $(uv) \in G$, at least one of u or v is in S.

Reduction: INDEPENDENT-SET $\leqslant_p$ VERTEX-COVER.

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

We return the pair $\langle G, n - k\rangle$. *(Is this polytime?)*

Proof that $\langle G, k\rangle \in$ INDEPENDENT-SET iff $\langle G, n - k\rangle \in$ VERTEX-COVER:

Suppose G has a size-k independent set I:

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

  Then for any $\{u, v\} \in E$, at least one of u or v is in $V \setminus I$.

  $\Rightarrow V \setminus I$ is a size-$(n - k)$ vertex cover in G.

Suppose G has a size-$(n - k)$ vertex cover V':

- Then for any $\{u, v\} \in E$, at least one of u or v is in V'.

  Then for any $u \neq v \in V \setminus V'$, $\{u, v\} \notin E$

  $\Rightarrow V \setminus V'$ is a size-$(n - (n - k)) = k$ independent set in G.

PARTITION-INTO-TRIANGLES

INPUT: A graph G.

QUESTION: Can G be split into disjoint triangles?

Reduction: 3SAT $\leqslant_p$ PARTITION-INTO-TRIANGLES.

Suppose we are given an input 3CNF ϕ.

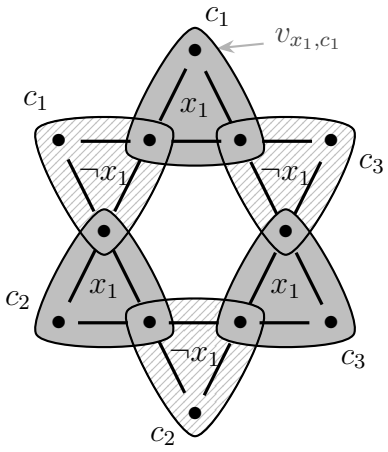
$$\text{e.g., } \phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

We'll try to address the following questions:

- When we reduced to 3COL we encoded our truth assignments into a series of variable widgets. Can we do the sme here?
- How can we program the clauses?
- Will we need to do a bit extra to finish the job?

Let's start by building a variable widget:

Here's one of the widgets we'll use for our example, this one for x_1. Notice that it has two "leaves" for every clause C_j.



We'll connect the points of the "leaves" to the different clauses.

The big idea behind this widget is that it's only the outer nodes that can connect to the rest of the graph. So the choices we make for the inner nodes is binary:

- If we choose to include the top triangle for c_1 and x_1 in our partition, then we've used up its two innder nodes. We can't use them in any other triangle.

  This means that we cannot include the neighbouring trinagles for c_1 and $\neg x_1$ or for c_3 and $\neg x_1$.

  But both of these trinagles will have an unused inner node, and there will be only one remaining triangle that can be used for each node.

  So we'll have to include the triangles for c_2 and x_1 and for c_3 and x_1.

  Basically, if we include any one of the solid-shaded triangles in our partition, we have to include all of them.
- If we do not include the triangle for c_1 and x_1 in our partition, we'll hve to find other trinagles to hold the two inner nodes.

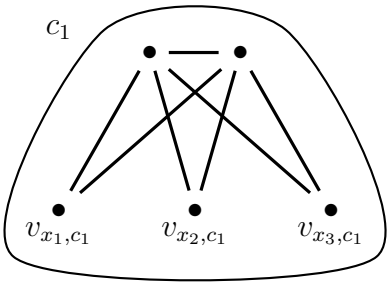
  So we'll have to include the triangles for c_1 and $\neg x_1$ and for c_3 and $\neg x_1$.

  Following a similar line of reasoning s before, we see that if we do not include any of the solid-shaded triangles in our partition, then we must include all of the striped ones.

You can see that this widget allows us to encode a binary choice from our varibles into our graph. We can extend this to all of the variables to get a a truth assignment – so long as we can deal with the outer nodes in the widget.

It turns out that the clause itself is easy to encode:

We can just attach the base of a triangle to each leaf corresponding to its literal:

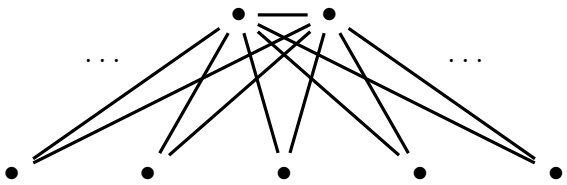


If the clause is satisfied by our truth assignment, we should be able to pick one true literal from each clause. We'll just attach the node from that literal to the two clause nodes to make one more triangle.

We're almost there, but we've got a bunch of those outer vertices in our variable widgets left over. . .

We'll build a bunch of oversized clause widgets as garbage collectors:

It's basically like a large clause widget, but connected to every leaf in every variable widget:



We put in just enough of these to cover all the vertices (if we use counting argument we'll see that we need $n(k-1)$ copies).

Again, please see the Polytime Reductions handout to see the details as to why this reduction is polytime.

Proof that $\phi \in$ 3SAT iff $\langle G \rangle \in$ PARTITION-INTO-TRIANGLES:

Suppose that $\langle \phi \rangle \in$ 3SAT, with n clauses and k variables:

- 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.

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.

3DM

INPUT: Three sets A, B, C, all of size n, and a set T of triples in $A \times B \times C$.

QUESTION: Is there an $M \subseteq T$ such that every element of A, B, and C occurs exactly once in M?

Reduction: 3SAT $\leqslant_p$ 3DM.

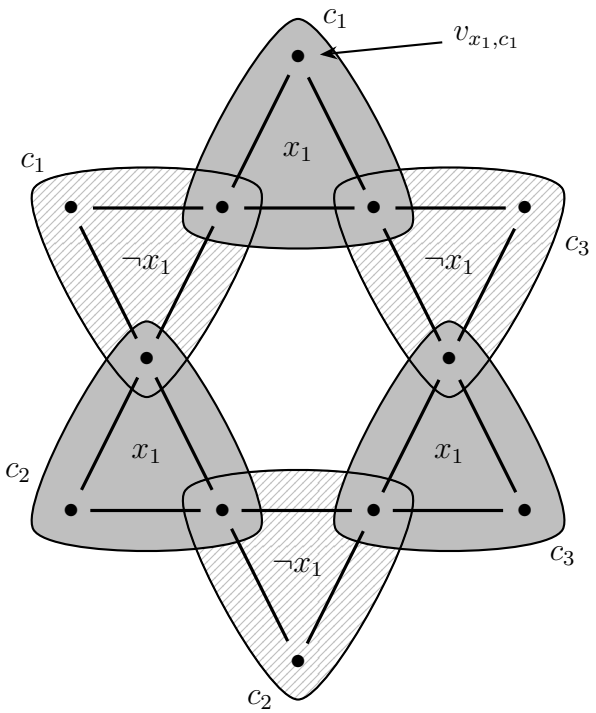
What's interesting about this reduction is that we'll basically build it by extending the reduction for PARTITION-INTO-TRIANGLES: Suppose we are given an input 3CNF ϕ.

$$\text{e.g., } \phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

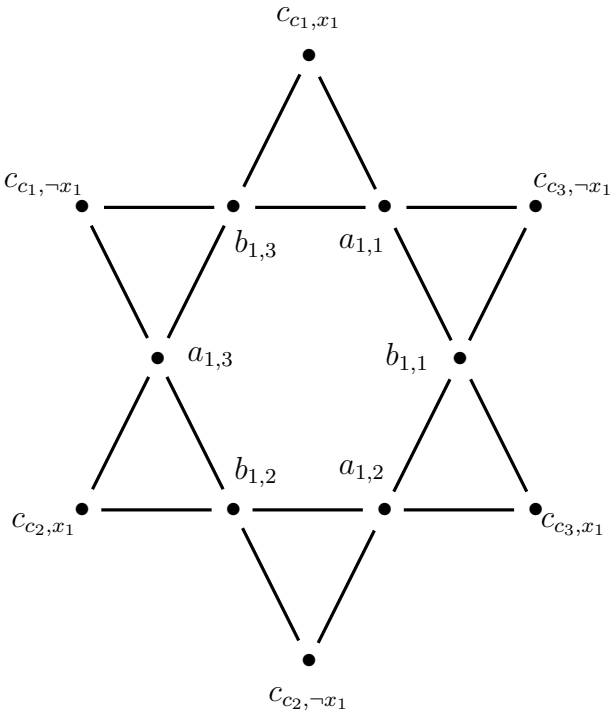
We'll actually start by building the same graph G. Our new key observation is that this G is always 3-colourable – and that in fact, there is an easy-to-find 3-colouring (finding a 3-colouring is usually hard, but we're lucky in this case).

The sets A, B, and C that we use in our 3DM instance are the *green*, *red*, and *blue* vertices. The set T is the set of triangles in G.

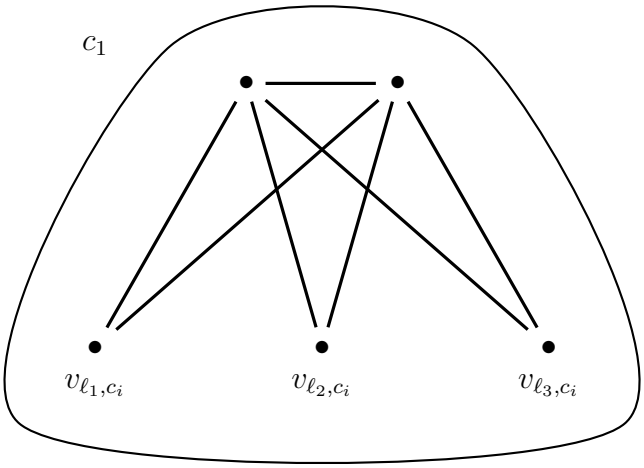
Recall that in our PARTITION-INTO-TRIANGLES reduction we built the following variable widgets:



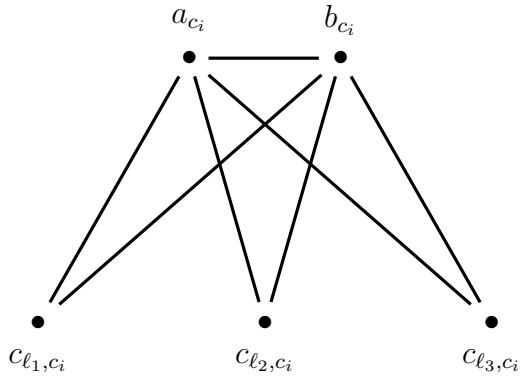
If we remove the shading on the triangles we'll be able to see the colourings a bit better. The colouring we'll use is as follows (the three colours here are a, b, and c):



We started with the following clause widgets:



And we can colour these widgets like so:



And finally, we can colour the garbage collection widget in the same way.

If we do this, we we'll collect all of the a vertices into the set A, the b vertices into the set B, and the C vertices into the set C. The set T will be made up of the trinagles from G (note that since we have a 3-colouring every trinagle has one node of each colour). Once this process is done, we see that $|A| = |B| = |C| = 2nk$, and so we have a valid 3DM instance.

We observe that any choice M of triples from T corresponds to a choice of triangles from G, and that M contains ech element exactly once iff the triangles form partition of the vertices in G. Hence, the iff proof that hows that $\langle G \rangle \in$ PARTITION-INTO-TRIANGLES iff $\langle \phi \rangle \in$ 3SAT also show that $\langle A, B, C, T \rangle \in$ 3DM iff $\langle \phi \rangle \in$ 3SAT.

Some Problems in $\mathcal{P}$

We've seen that a problem can be $\mathcal{NP}$-complete when the process of searching for a solution is difficult. But sometimes we can see problems that are very similar to $\mathcal{NP}$-complete problems, except the choices of solutions are limited enough that we can effectively search them. These include:

- 2SAT

  INPUT: A boolean formula ϕ in 2CNF.

  QUESTION: Does ϕ have a satisfying assignment?

- 2COL

  INPUT: A graph G.

  QUESTION: Is G 2-colourable?

  Question: *Why might this be the case?*

The two examples are in $\mathcal{P}$ for similar reasons. Let's look at an example of the 2-colouring problem as an example:
High-Level Idea:

> Given a graph $G = (V, E)$, we pick a starting point s and run a BFS of the graph. Every time we search a vertex, we colour it *red* if its distance from s is even, and *blue* if it is odd. Repeat this process for every component.
>
> Once we're finished, just check to see if there are any neighbours of the same colour.

We can see that once we've chosen the colour of the first vertex, the possible colours of all vertices in its connected component are immediately fixed. So there's only one choice to worry about.

If we want to see if a graph G is 3-colourable, though, we have many choices to search through.

> **Takeaway:** If a problem looks like an NP-complete problem we've already seen, it's likely to be NP-complete itself, but we still need a proof (such as a polytime reduction) in order to make sure.

NP-Intermediate Problems

We have described the class $\mathcal{P}$ of efficiently solvable problems and the $\mathcal{NP}$-complete class of problems. Is every language in $\mathcal{NP}$ in one of these classes?

Ladner's Theorem: If $\mathcal{P} \neq \mathcal{NP}$, then there are problems in $\mathcal{NP}$ which are neither $\mathcal{NP}$-complete nor in $\mathcal{P}$.

We call these the $\mathcal{NP}$-intermediate $\mathcal{NPI}$ problems. We won't go over the proof of this theorem here, however.

As a matter of fact, there are several problems we believe to be in this class, but we don't currently know how to prove it. One such problem is the problem of integer factorization:

FACT:

INPUT: An $\langle n \rangle, n \in \mathbb{N}$, $\langle a \rangle, a \in \mathbb{N}$.

QUESTION: Does n have a factor k such that $k \leq a$?

It turns out that determining if a number is prime or composite is in $\mathcal{P}$. But if a number is not prime, we do not believe that there is an algorithm in $\mathcal{P}$ that can help us efficiently find its factors. This is why public-key cryptography systems such as RSA are possible.

The Take-Away from this Lesson:

- We've described several $\mathcal{NP}$-complete problems.
- We've described some problems in $\mathcal{P}$ that might look to be $\mathcal{NP}$-complete, but aren't.
- We've Talked about $\mathcal{NP}$-intermediate problems.

Glossary:

$\mathcal{NP}$-intermediate

***Languages*:**

2COL, 2SAT, 3DM, FACT,
INDEPENDENT-SET,
PARTITION-INTO-TRIANGLES, VERTEX-COVER