CSCC63 – Week 7

This week: We'll introduce some more ideas about polynomial decidability, and we'll start using these definitions to analyze languages.

Decidability and Polytime Decidability

Last time we introduced the classes $\mathcal{P}$ and $\mathcal{NP}$, and argued that they give us a robust model of efficient computation.

> $\mathcal{P}$ is the class of problems that can be solved in polynomial time.
>
> *i.e., Problems that can be solved efficiently.*
>
> $\mathcal{NP}$ is the class of problems that can be verified in polynomial time.
>
> *i.e., Problems whose solutions can be checked efficiently.*

We have conjectured that these classes are different: we have no proof at this time, but checking whether a solution is correct fundamentally seems to be much easier than actually finding that solution.

We noticed that there is a parallel between these ideas and the ideas of decidability and recognizability:

> The **decidable** languages can be solved in a finite amount of time.
>
> *i.e., Problems that can be solved at all.*
>
> The **recognizable** languages can be verified in a finite amount of time.
>
> *i.e., Problems whose solutions can be checked at all.*

We shouldn't really be surprised at these parallels — we're asking similar questions, except now we're more concerned with whether we can compute problems efficiently rather than whether we can compute them at all.

> **Question**: *We're not only interested in showing that some languages are efficiently solvable, but that some languages are not.*
>
> *Can we extend the above parallels as a way of developing the tools to do so?*

The answer is *yes*. So let's try to extend this parallel.

Computability	Complexity
The decidable languages	The class $\mathcal{P}$
The recognizable languages	The class $\mathcal{NP}$
The co-recognizable languages	????
Mapping reductions	????
The language HALT	????

Question: *Do these ideas have parallels when we're looking at efficient computation?*

The Class co-$\mathcal{NP}$

The first parallel we'll give is to extend the idea of co-recognizability — we'll describe the class co-$\mathcal{NP}$:

Definition: The class co-$\mathcal{NP}$ is the set of languages L whose complements are in $\mathcal{NP}$:

$$L \in \text{co-}\mathcal{NP} \iff \overline{L} \in \mathcal{NP}.$$

We have conjectured (again) that this class is separate from $\mathcal{NP}$:

- It is easy, e.g., to come up with a proof that a given graph G has a Hamiltonian path from some node s to t.
- It is much harder to show that there is no such path — you have to show that *every* possible path is not a Hamiltonian path.

> It's actually even worse than that — if we could show that $\mathcal{NP}$ = co-$\mathcal{NP}$, then we'd be able to efficiently verify not only when a graph doesn't have a Hamiltonian path, but we could show that much harder problems, like the polynomial time analogues of the ALL_{TM} problem, would also have efficient certificates and verifiers. But that goes beyond what we're covering today.

Since co-$\mathcal{NP}$ is the class of the languages whose complement is in $\mathcal{NP}$, we can show a language L is in co-$\mathcal{NP}$ by using the five steps from the last lecture to show $\overline{L} \in \mathcal{NP}$:

1. Show that L (and so $\overline{L}$) is a yes/no question.
2. Give a certificate for $\overline{L}$.
3. Show this certificate is polynomial in the size of the input to L.
4. Give a verifier for $\overline{L}$.
5. Show this verifier is polytime in the size of the input to L.

Question: *How are the classes $\mathcal{P}$, $\mathcal{NP}$, and co-$\mathcal{NP}$ related?*

We don't know: In fact, the following questions are still unresolved, and are among the "Millennium Problems" recognized by the Clay Mathematics Institute. If you can fully resolve any of them you can win a million dollars[1]:

[1] And more importantly, get 100% in this course.

- Is $\mathcal{P}$ = $\mathcal{NP} \cap$ co-$\mathcal{NP}$?

  (no strong consensus)

- Is $\mathcal{NP}$ = co-$\mathcal{NP}$?

  (strongly believed to be no)

- Is $\mathcal{P}$ = $\mathcal{NP}$?

  (strongly believed to be no)

Polytime Reductions

The second parallel we'll give is to the concept of mapping reductions.

> Recall: $A \leqslant_m B$ iff there is a computable function $f : \Sigma^* \to \Sigma^*$ such that
>
> $$\forall w \in \Sigma^*, w \in A \leftrightarrow f(w) \in B.$$
>
> The idea was that we can transform an instance of A to an instance of B in such a way that if we could determine whether $w \in B$, we could also determine whether $x \in A$ by asking whether $f(x) \in B$.

We used this idea to show that either B is at least as hard as A, or that A is at least as easy as B — in particular, if A is known to be hard, then so is B.

We can do the same thing in the polytime realm by simply requiring that f be polytime:

Definition: We say A is polytime reducible to B ($A \leqslant_p$) iff there is a polytime computable function $f : \Sigma^* \to \Sigma^*$ such that

$$\forall w \in \Sigma^*, w \in A \leftrightarrow f(w) \in B.$$

We can use polytime reductions in the same way as we used mapping reductions: If $A \leqslant_p B$, then a function f as described above exists.

If a program g can decide B in polytime (so running $g(w)$ can tell if $w \in B$), then $g \circ f$ can decide A in polytime (to tell if $x \in A$, we let $w = f(x)$, then we run $g(w)$).

Similarly, if we know that A cannot be decided in polytime, then $g \circ f$ cannot decide A in polytime, and so g cannot decide B in polytime.

This tells us that:

Theorem 7.31:

If $A \leqslant_p B$ and $B \in \mathcal{P}$, then so is A.

If $A \leqslant_p B$ and $B \in \mathcal{NP}$, then so is A.

Corollary:

> If $A \leqslant_p B$ and $A \notin \mathcal{P}$, then neither is B.
>
> If $A \leqslant_p B$ and $A \notin \mathcal{NP}$, then neither is B.

We're interested in the class $\mathcal{NP}$. So we'd like to be able to tell if a language $L \in \mathcal{NP}$ can be expected not to be in $\mathcal{P}$ (given the conjecture that $\mathcal{P} \neq \mathcal{NP}$).

> **Question**: *Up until now, we've showed that languages are undecidable by starting with an undecidable language, such as* $\mathrm{A_{TM}}$ *or* HALT *and reducing these languages to* L*. It's going to be difficult to start with a language that we know is in* $\mathcal{NP} \setminus \mathcal{P}$*, since we haven't proven that these classes are unequal. But is there a language that is so unlikely to be in* $\mathcal{P}$ *that we can use it in a similar way? Is there an analogue to* HALT*?*

An analogue to HALT:

In order to give an analogue to HALT, we'll need to look a bit at what role it plays for the recognizable languages:

> **Question**: *Why is* HALT *a good language to use in our reductions?*

We know that HALT is undecidable, so we can use it to show that other languages are undecidable.

> **Note:** We can argue that HALT $\leqslant_m L$ implies that L is undecidable.
>
> We can, however, find recognizable but undecidable L such that HALT $\not\leqslant_m L$. We haven't covered them in this course, but they do exist.

So we're looking for a language in $\mathcal{NP}$ that we do not believe to be in $\mathcal{P}$. We'll find that there's one type of language that best fits this role. To see why, we'll note that HALT has another interesting property:

For any recognizable language L, $L \leqslant_m$ HALT.

Why? You've already seen this idea in tutorial 4: If L is recognizable, then it has an recognizer R. So we can write the TM

> Let R' = " On input $\langle x \rangle$:
>
> 1. Run R on $\langle x \rangle$.
> 2. If it accepts, *accept*, otherwise *loop*.

We can clearly see that $w \in L$ iff $\langle R', w \rangle \in \text{HALT}$.

So not only is HALT *undecidable, but it has the property that we can use it to solve any other recognizable language!*

Definition: For any class $\mathcal{C}$ of languages, we say that a language L is *hard* for that class if every $A \in \mathcal{C}$ reduces to it.

So HALT *is hard for the recognizable languages.*

Definition: For any class $\mathcal{C}$ of languages, we say that a language L is *complete* for that class if $L \in \mathcal{C}$ and every $A \in \mathcal{C}$ reduces to it.

So HALT *is complete for the recognizable languages.*

Question: *Is there an $\mathcal{NP}$-complete language?*

Note that when we were talking about hardness and completeness for recognizable languages, reducible meant mapping reducible. Now we mean polytime reducible.

Cook-Levin Theorem: The language 3SAT described in the last class is $\mathcal{NP}$-complete.

We'll prove this later on, but it's not really a surprise: intuitively, computers are large Boolean circuits. So asking whether a large Boolean circuit can be set to true is a bit like asking if a computer (and specifically a computer program) has some input for which it will output yes.

What this means is that if $\text{3SAT} \in \mathcal{P}$, then any problem in $\mathcal{NP}$ can also be solved in polynomial time.

Since we believe this is not the case, we strongly suspect that $\text{3SAT} \notin \mathcal{P}$

Furthermore, if $L \in \mathcal{NP}$ and $\text{3SAT} \leqslant_p L$, then L is also $\mathcal{NP}$-complete.

If $\mathcal{P} \neq \mathcal{NP}$, then no such L is in $\mathcal{P}$.

This gives us a tool for determining that a language L is highly unlikely to be in $\mathcal{P}$.

In particular, the HAM-PATH and CLIQUE problems we covered in the last class will turn out to be $\mathcal{NP}$-complete (in fact, all of the languages we covered other than PATH are known to be $\mathcal{NP}$-complete).

> *You'll assume this in your assignment two — When I ask you to show a language is $\mathcal{NP}$-complete, you need to show me it's in $\mathcal{NP}$, and then give me a polytime reduction from the indicated language.*

So we've introduced the $\mathcal{NP}$-complete class of languages. These are the "hardest" problems in $\mathcal{NP}$, in that a solver for any one of these languages can also be used to solve any problem in $\mathcal{NP}$.

To recap the idea:

A Language L is $\mathcal{NP}$-complete if:

1. $L \in \mathcal{NP}$.
2. $\forall A \in \mathcal{NP}$, $A \leqslant_p L$.

Here is why we care about this idea:

If any $\mathcal{NP}$-complete problem is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Since we believe that $\mathcal{P} \neq \mathcal{NP}$, showing that a language is $\mathcal{NP}$-complete is strong evidence that it is unlikely to be in $\mathcal{P}$.

In order for this class to be useful to us, we need to be able to show that a language in L is $\mathcal{NP}$-complete. We've covered item 1. of the above definition in previous classes, so let's have a look at the second.

> **Question**: *How can we tell if, for all A in $\mathcal{NP}$, $A \leqslant_p L$?*

Proving this for *all* A in $\mathcal{NP}$, of course, is difficult. But polytime reductions are transitive:

- If $A \leqslant_p B$, there is some polytime f that maps inputs to A into the inputs of B.
- If $B \leqslant_p C$ there is some polytime g that maps inputs to B into the inputs of C.
- So $g \circ f$ is polytime, and maps the inputs of A to the inputs of C, and so $A \leqslant_p C$.

Idea: If we know that one problem S (such as 3SAT) is $\mathcal{NP}$-complete, we just have to show that $S \leqslant_p L$.

So what does a polytime reduction look like?

> *We won't be using a single format of proof this time in the way that we did when proving undecidability. Since the problems we will be looking at are more varied (e.g., embedding questions about Boolean formulas into questions about cliques in graphs), no single template will work quite as well.*
>
> *We'll have to vary how we approach these reductions in order to fit the different languages we see. We'll try to cover several languages so you can have examples to learn from, but there are a number of useful ways to start our reductions.*

If we're lucky, we can cast our problem L as a generalized or extended version of a known NP-complete problem. When this is the case, the reduction is trivial – just show that the known NP-complete problem is or maps to a natural subset of L.

Let's See a Few Examples:

Example 1: 3COL

Recall the 3COL problem:

Input: A graph G.
Question: Does G have a valid 3-colouring?

To show 3COL is NP-complete, we need to show that it is in NP, and we need to show that some NP-complete problem L reduces to it. The one $\mathcal{NP}$-complete languge that we have so far is 3SAT, so we'll use that.

What we want to do now is figure out how to do a polytime reduction: we want to write a polytime algorithm such that:

- The input to our process is a 3SAT instance – i.e., a 3CNF Boolean formula ϕ.
- The output of our process is an undirected graph G.
- We want to ensure that G is 3-colourable iff ϕ is satisfiable.

Important: *We don't think that we can actually solve* 3SAT *in polytime, so we can't try to solve it as part of our reduction. Neither can we reference any satisfying truth assignment, nor take any other step that assumes that we know whether ϕ is satisfiable.*

So where do we start?

Well, if we have no other ideas about how to start, we'll just try looking at a few instances of our target langauge (in this case 3COL). The idea is that ultimately we want to "program" a Boolean circuit into 3COL – so as we go we'll look for different sorts of "forced behaviours" that we might be able to use to program.

Spcifically, we'll be writing G with the understnding that someone else will be looking at G and trying to 3-colour it. When we say "forced behviour", what we mean is a graph construction that forces our user to make certain colour choices while searching for a colouring – to be exact, that forces our user to make the colour choices that we want to see.

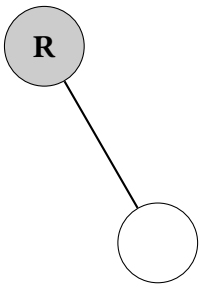
So let's start. We'll build a simple graph and try to colour it. We'll start with the colours red (**R**), green (**G**), and blue(**B**), although we'll change that in short time.

The simplest graph we can build is a single node:

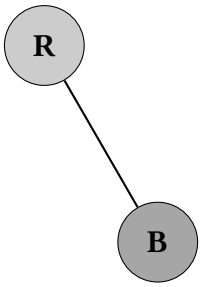
It's also very easy to colour.

We could also colour this node green or blue, by the way: the red colour is arbitrary.

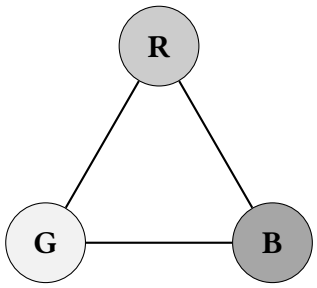
OK, let's grow the graph by one node and see what happens:



We want to try to colour this graph – in fact, we'll try to extend the colouring we already have. There were three colours to choose from for our first node, but now there are only two for the second, since the first is taken. On the other hand, we can easily choose either of the remaining colours for the new node:

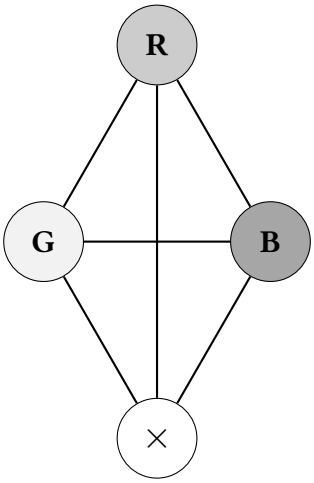


If we add a third node to make a triangle there will be only one choice for its colour:

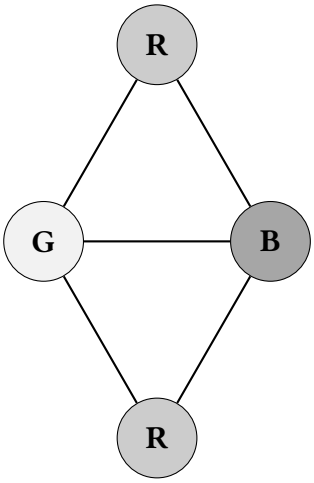


We could add the new node in ways that don't make triangles, of course – but remember we're looking for what we've called forced choices. We're writing the graph G so that if someone else were to try to 3-colour it, then the colour choices made in this process would be the choices we want. So we'll try to make G as constrained in this regard as possible. Here we've got a situation where there's only one choice, which is what we want.

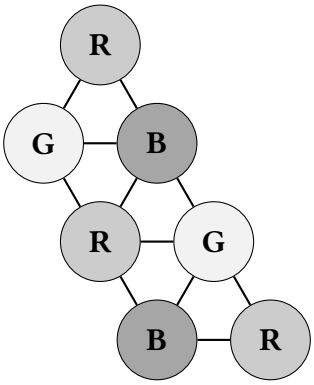
Now, we can't connect a new node to all three nodes here:



So this is about as constrained as we can make G. But we can still add nodes: if we remove one of the new edges we'll have another forced choice:

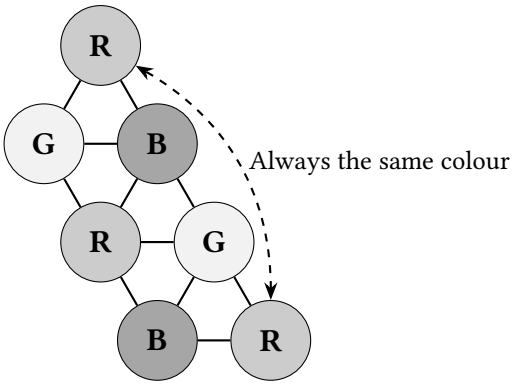


And we can repeat this pattern as long as we want:



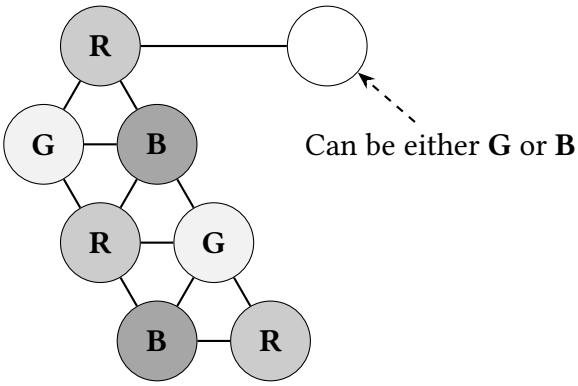
. . .

Note that the fact that we have these forced choices means that nodes that are far away from each other are still mutually dependent.



In fact, what we have here can be thought of as a way of copying a value from one node to another.

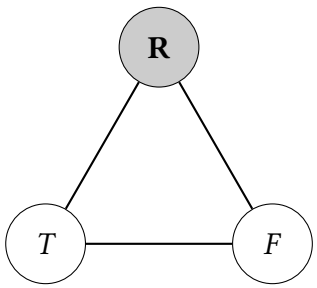
So we can force a choice of colours in our graph (once we've made the initial choice of **R** and **B**). We can also force out hypothetical searcher to make a *binary* colour choice:



Binary choices are interesting to us: we want to encode a Boolean circuit, and the values that the variables take in these circuits are binary: *T* or *F*. So while we've just been playing around a bit with the possible ways of building G, we've already found a way to encode a binary *true*/*false* choice and to copy values from one node to another (or rather, to force our searcher to copy values). That's actually a good start: if we can build logic gates into G in the same way, we'll have enough to do a reduction. But first, let's formalize what we've got.

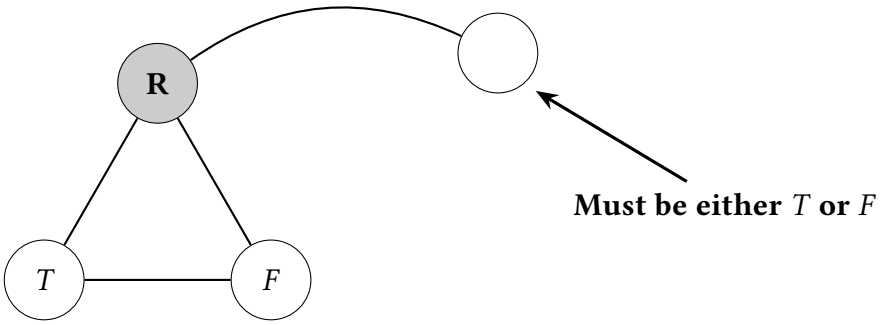
Up until now we've been using the colours *red*, *green*, and *blue*. But we want two of our colours to represent *true* and *false* values. To let's rename the *green* and *blue* colours to *true* and *false*. This means that now we'll label the vertices as **R**, *T*, or *F*.

And in order to enforce our colour choices we'll initalize our G with our first maximally-constrained graph – the triangle:

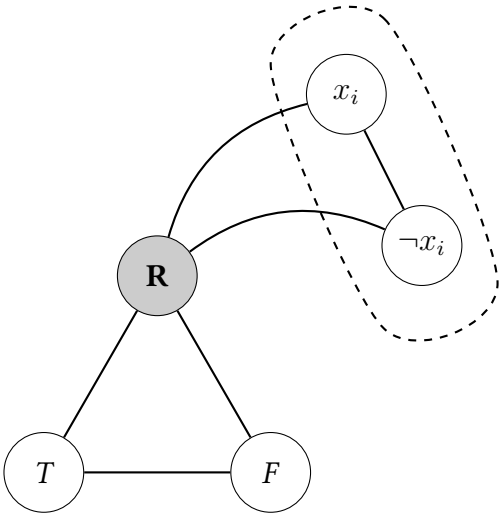


We'll refer to this triangle in G as our *palette* widget. As we've indicated before, it's possible for someone colouring this graph to make a different set of colour choices for these vertices (e.g., swapping the **R** and T labels), but any possible choice will be equivalent to this one up to some such permutation of colours.

We'll now start encoding our input 3CNF ϕ itself into G. To start, we reiterate the fact that if we attach the **R** palette node to any other node, we force that other node to take one of the two Boolean values.

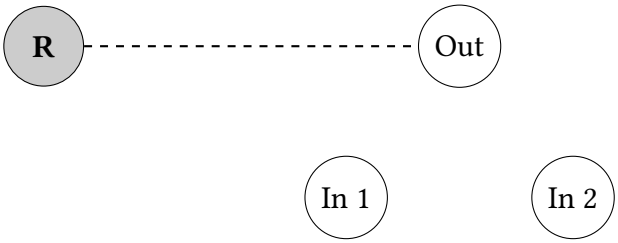


Using this trick, the first thing we want to encode are the variable assignments for ϕ. We can do this by creating a node for every variable x_i, and for its negation $\neg x_i$. If we connect both to **R**, they must take values of *T* or *F*, and if we connect them to each other they must take opposite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



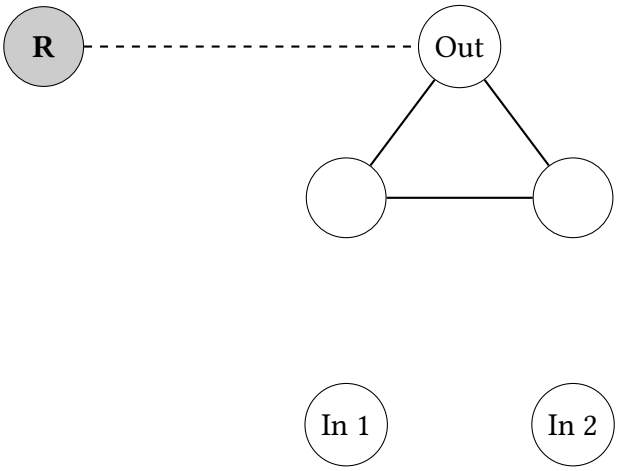
We can repeat this process for every variable.

If we can get this far, we just need to figure out way of putting together an *or*-widget in our graph (it'll be clear soon why we're not worrying about *and*-widgets). As a first step to doing this, let's write the inputs and outputs of the widget:

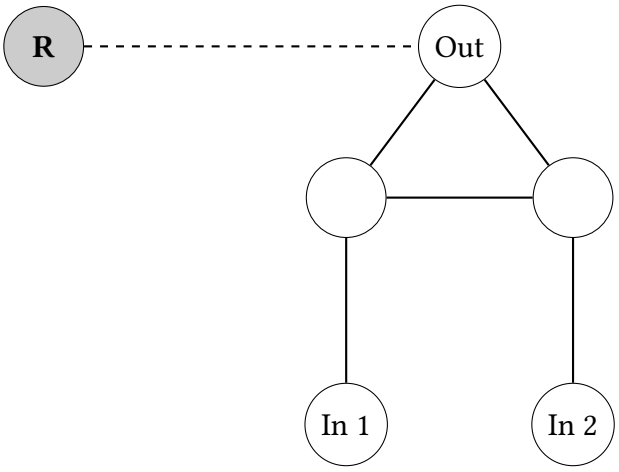


The inputs will be whichever varibles (or their negations) that we want to plug into the *or*-gate. The output should be Boolean, so we'll attach it to the **R** node on the palette. We won't attach it to the other nodes in the palette, since the *or*-gate can output either T or F.

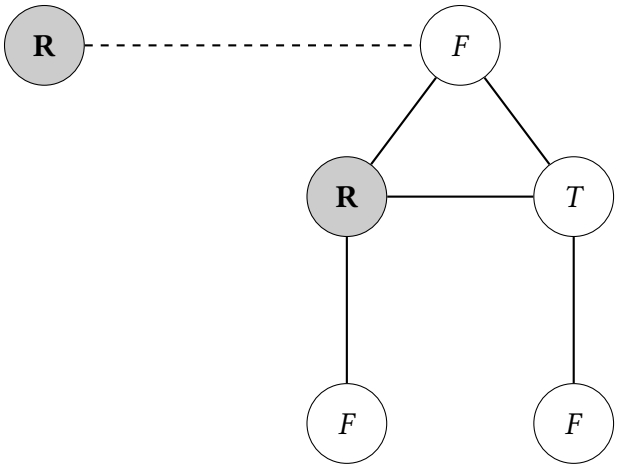
Now, if the output node of our *or*-gate takes some value, and if we make it part of a triangle, then the other corners of that triangle must take the other two colour values:



For this to be an effective *or*-gate we want to set it up so that we cannot set the output to T if both outputs are F, but so that we can set it to T otherwise. Here's a way to do that:

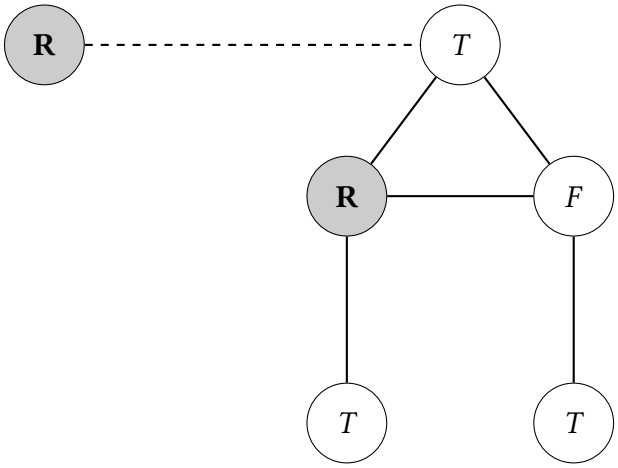


To be exact, if both input values are *F*, the only way to fill colour the widget is

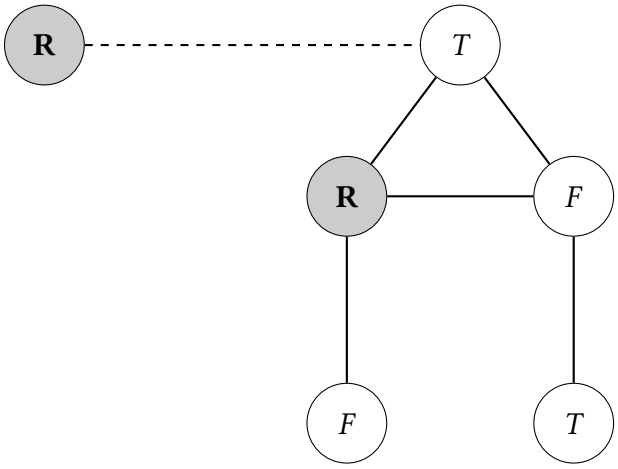


(Or the mirror image.)

On the other hand, If we have one of the other inputs then we can extend the colourings to



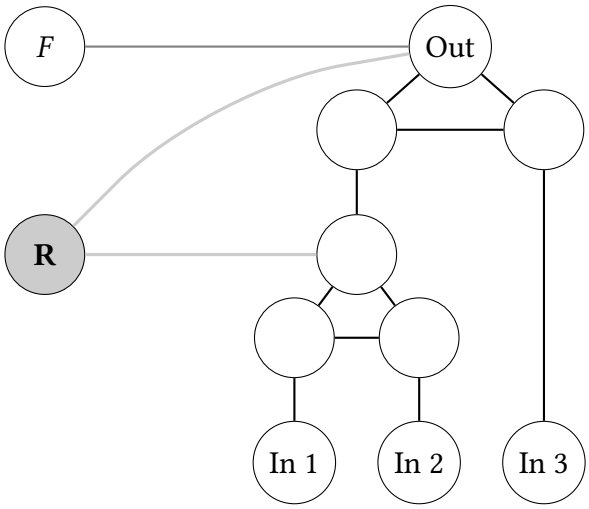
or



(Again, or their mirror images.)

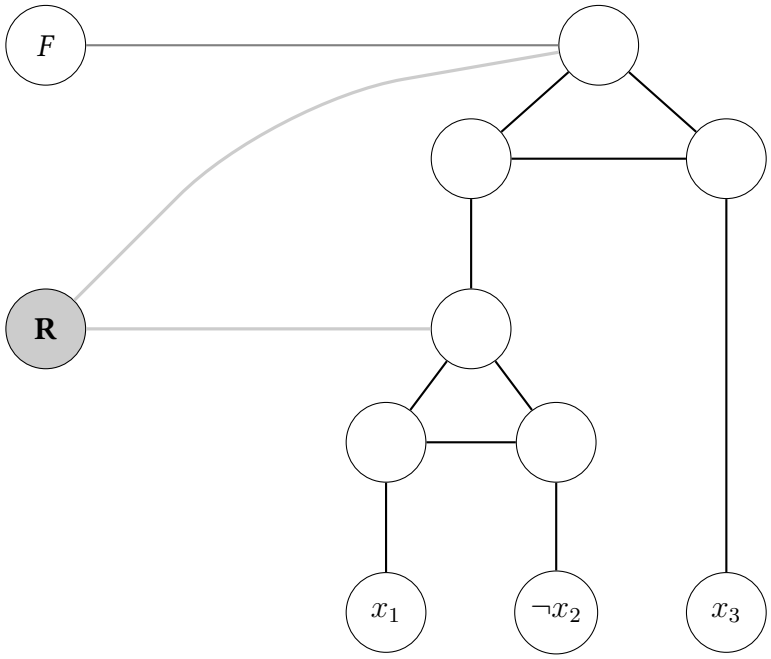
Are all possible. So it is possible to assign a T to the top node if and only if at least one bottom node is T, as desired.

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 T iff at least one of the bottom nodes is also T, but the fact that we have connected this node to **R** and F 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. E.g., if our first clause is $(x_1 \vee \neg x_2 \vee x_3)$, we'd build the following widget:



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.

Suppose ϕ is satisfiable: 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.

Suppose G is 3-colourable: If our output graph G is 3-colourable, then we can argue wlog. that its palette widget has the **R**/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 need to show this construction can be found in polytime. We can see that for an ℓ-variable, k-clause 3CNF 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 eleven 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.

Note that to save space we won't always go over the process we use to find the reduction – in mny of our examples we'll give the reduction itself. But what we see here is the process.

Reductions and Hardness

When we prove that, e.g., 3COL is $\mathcal{NP}$-complete, we prove that it's one of the hardest languages in $\mathcal{NP}$ to solve. But in a sense, this is a secondary point – it's more of a corollary to what we prove, rather than what we actually do prove.

What we've actually done is demonstrate that we can embed any languge in $\mathcal{NP}$ into 3COL. We can build logic gates (in this case *or*-gates) into a 3COL instance, and so anything we can do with a Boolean circuit we can also do with 3COL as well.

And we can build computers out of Boolean circuits.

When we prove that 3SAT $\leqslant_p$ 3COL what we're really proving is that 3COL is one of the most *expressive* languages in $\mathcal{NP}$. The hardness of the problem is a side effect of this expressivity.

Let's look at a few more examples.

Example 2: CUBIC-SUBGRAPH

Here's a new language for us to play with. The definition is as follows:

> **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.

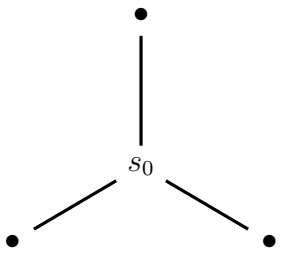
Since we're looking at the polytime reductions, we'll skip for the moment the proof that CUBIC-SUBGRAPH $\in \mathcal{NP}$. But you're encourged to think about it as an exercise.

So let's look at the 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:

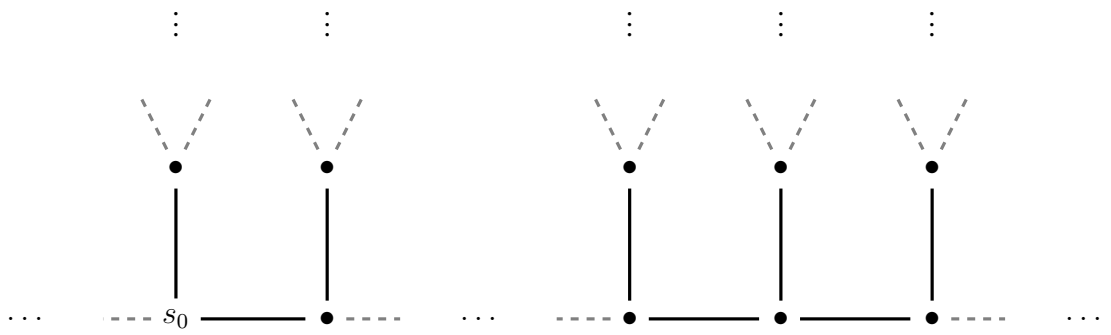
1. We'll use G to build a graph G' with a specified node s_0 such that G' contains a cubic subgraph that contains s_0 iff G has a 3-colouring.
2. We'll argue that every cubic subgraph of G' must contain this node s_0.

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'.

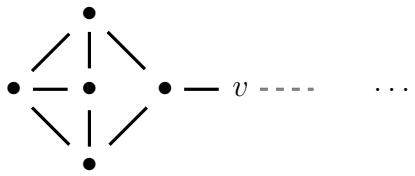
We can extend this idea. Suppose we build a whole chain of vertices from s_0 such that each vertex in the chain has three neighbours:



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):

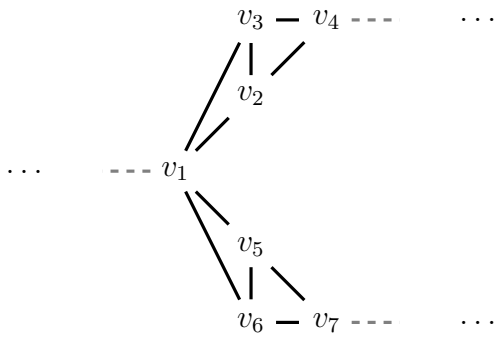


All of the vertices other than v in this subgraph have degree three – so if we had a subgraph of G' that was almost cubic, but which had one degree two vertex v, we could attach this widget to v to get a cubic subgraph.

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).

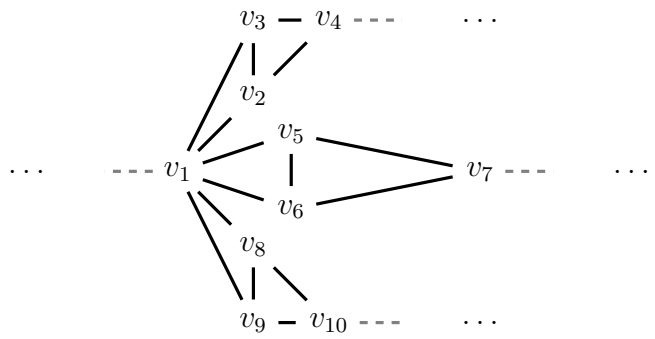
But this means that we cannot include both v_2 and v_5 – otherwise, v_1 would have a degree of at least four.

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

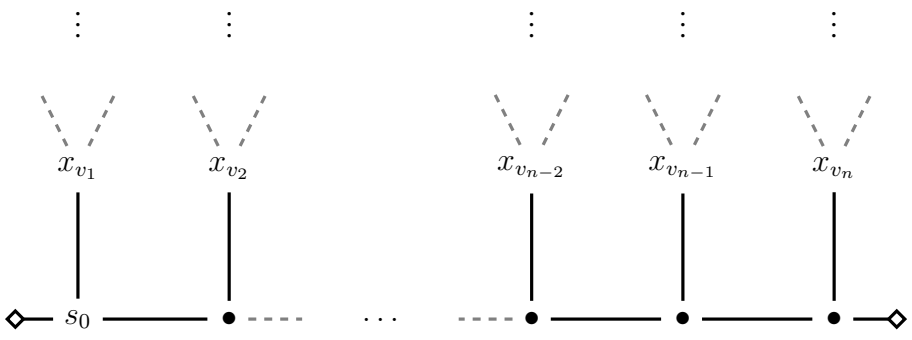
In fact, we have as many branches as we want coming out of the widget:



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

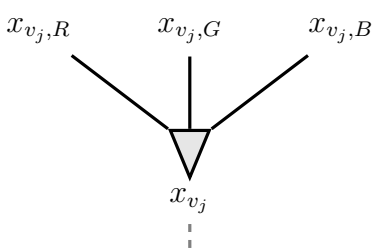
With these widgets, we are now ready to do our reduction.

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



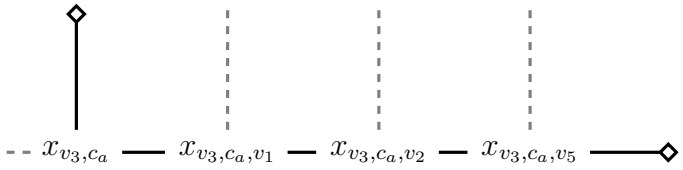
Note that X_G contains the vertex s_0, and so we'll start by looking only at cubic subgraphs that contain this vertex. From our previous discussion, we know that if G' contains X_G then any cubic subgraph of G' must either contain all of X_G or none of it. Since we insist that our cubic subgraph must 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:



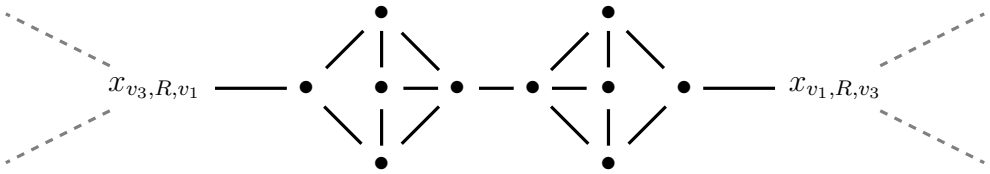
From our discussion about the choice widgets, we know that if any node in these widgets in included in a cubic subgraph of G', then the node x_{v_j} must be in that subgraph. Furthermore, if x_{v_j} is included, then exactly one of the three $x_{v_j,R}$, $x_{v_j,G}$, or $x_{v_j,B}$ branches in included. 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.

At this time we still need to finish our proof with an argument that $\langle G \rangle \in$ 3COL iff $\langle G' \rangle \in$ CUBIC-SUBGRAPH, and with an argument that G' can be constructed from G in polynomial time. We'll forego these steps for the moment, but there will be a Polytime Reductions handout in the Extras section of the course files that has this reduction and the associated iff proof and polytime anlysis, and you are encourged to look there for the details.

So 3COL $\leqslant_p$ CUBIC-SUBGRAPH, and so CUBIC-SUBGRAPH is $\mathcal{NP}$-complete.

Example 3: CLIQUE

Recall the definition of CLIQUE from last time:

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

Question: Does G have a clique of size k?

We've already showed that CLIQUE $\in \mathcal{NP}$, so let's shopw that it's $\mathcal{NP}$-hard.

Proof that 3SAT $\leqslant_p$ CLIQUE**:**

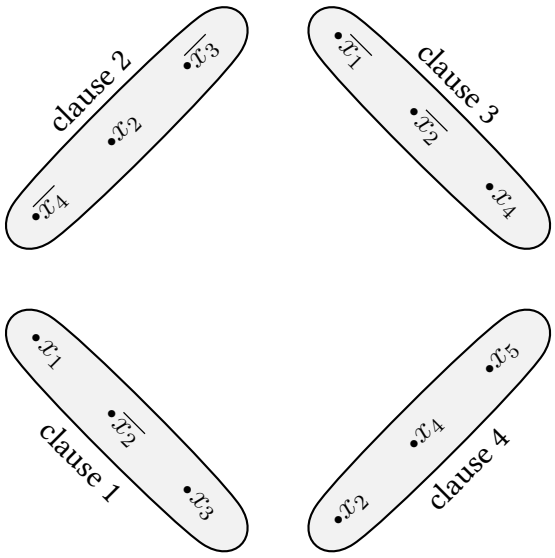
Let's start with the usual input: a 3CNF formula ϕ.

e.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_4} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (x_2 \vee x_4 \vee x_5)$

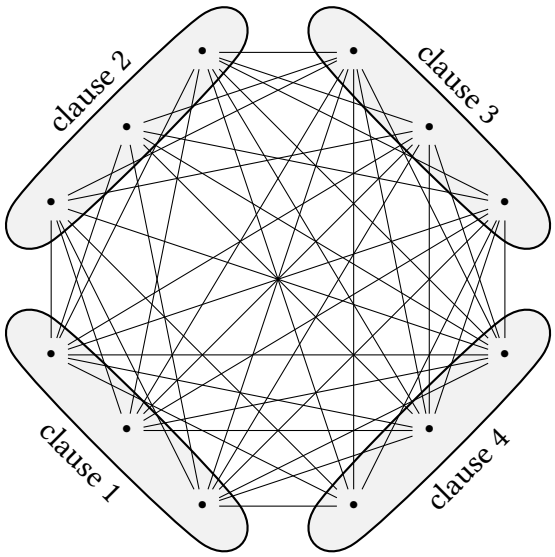
Idea: This time we'll do approch the whole process a bit differently then we did in the 3COL reduction. In that reduction we encoded a set of binary choices into our G, used those choices to represent a truth assignment, and then built a set of widgets to verify whether the truth assignment was a satisfying one.

This time we'll start with the clauses instead of the variables. If have a satisfying truth assignment τ for ϕ, then this truth assignment will set at least one clause per literal to *true*. We'll build a set of vertices for each clause, and we'll encode these clause widgets so that a choice of a vertex in any widget (to be included in a k-clique) represents that clause being one of the true literals according to τ.

So we'll build one widget per clause, and each widget will have one vertex for each literal that could be set to *true* (so three vertices per widget):

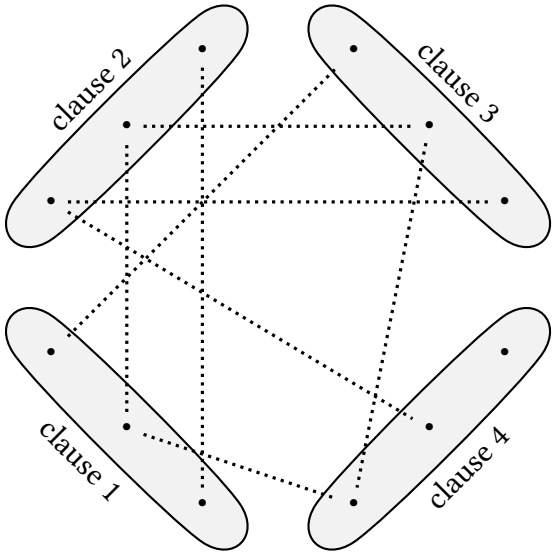


Now, we want to get a k-clique iff our choice of true literals corresponds to some satisfying truth assignment τ. If you think about this you'll see that this can be done as long as we don't choose conflicting literals: we can't, for example, choose x_1 to be true in clause 1 and $\overline{x_1}$ to be true in cluse 3. So we'll 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 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.

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

($\Leftarrow$): 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.

($\Rightarrow$): Suppose G has a k-clique S:

- S must have exactly one vertex in each clause group
- Every variable node in S 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 S, we can set it however we want).
- Every vertex in S 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.

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

Example 4: SUBSET-SUM

So far all of the reductions we've seen have been graph problems, but we'll actully run into $\mathcal{NP}$-complete problems from many different areas. The SUBSET-SUM problem is a well-known $\mathcal{NP}$-complete problem that deals with numerical sets and partitions:

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

We'll skip for the moment the proof that this language is in $\mathcal{NP}$, but it will be in the forthcoming polytime reductions handout. Readers are encouraged to do this proof as an exercise, as well. For the moment, though, we'll move on to the $\mathcal{NP}$-hardness part.

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 do a number of exercises with this reduction soon, though, and that should help clear things up. So make sure you can follow the basic steps — it'll make the exercises easier.

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	...
$\vdots$				$\vdots$			$\vdots$
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.

Extra

We'll give more examples of $\mathcal{NP}$-completeness reductions in class and in the tutorials, but there will also be several examples given in a pair of handouts in the extra notes section of the course files. So keep an eye out for that.

The Take-Away from this Lesson:

- We've introduced the class co-$\mathcal{NP}$.
- We've introduced the concept of polytime reductions.
- We've introduced the concept of hardness and completeness for a class, and stated without proof that 3SAT is $\mathcal{NP}$-complete.
- We have argued that if a language is $\mathcal{NP}$-complete it is likely not in $\mathcal{P}$, and we have given $\mathcal{NP}$-completeness reductions for:
  - 3COL
  - CUBIC-SUBGRAPH
  - CLIQUE
  - SUBSET-SUM

Glossary:

completeness, co-NP, hardness, polytime reductions

***Languages*:**

CUBIC-SUBGRAPH, SUBSET-SUM