NP-completeness reductions

Matvey Soloviev (Cornell University)

CS 4820, Summer 2020
We showed that the CNF-SAT problem is NP-complete, that is, it is in NP and (up to polynomial overhead) as hard as any problem in NP.
Last time

We showed that the CNF-SAT problem is **NP-complete**, that is, it is in NP and (up to polynomial overhead) as hard as any problem in NP.

In particular, if $P \neq NP$ – that is, if NP contains any problems that are not in P – then NP-complete problems are not in P:
We showed that the CNF-SAT problem is **NP-complete**, that is, it is in NP and (up to polynomial overhead) as hard as any problem in NP.

In particular, if $P \neq NP$ – that is, if NP contains any problems that are not in P – then NP-complete problems are not in P:
What now?

Are there more problems like that?
Are there more problems like that?

One thing is for sure: it’s going to be much easier to establish NP-completeness for any further problems. We just need to demonstrate a polynomial-time reduction from CNF-SAT.
NP-completeness architecture

Graph Iso. \quad \text{Ham. Cycle} \quad \cdots \quad \text{Flow} \geq k

\rightarrow

\text{CNF-SAT}

\rightarrow

\text{poly}

\rightarrow

\text{some other problem}
NP-completeness architecture

Graph Iso.  Ham. Cycle  ...  Flow \geq k

CNF-SAT

some other problem
NP-completeness architecture

Graph Iso.  Ham. Cycle  \cdots  Flow \geq k

\text{CNF-SAT}

\text{poly}

some other problem

\text{poly}

yet another problem

\text{poly}

\text{poly}

\text{poly}
Let’s start with a boring one.
Let’s start with a boring one.

Recall CNF-SAT: we have a conjunction of disjunctions like

\[(\ell_{1,1} \lor \ldots \lor \ell_{1,n_1}) \land \ldots \land (\ell_{m,1} \lor \ldots \lor \ell_{m,n_m})\],

and are asking if there is a satisfying assignment to all the variables.
Let’s start with a boring one.

Recall CNF-SAT: we have a conjunction of disjunctions like

\[(\ell_{1,1} \lor \ldots \lor \ell_{1,n_1}) \land \ldots \land (\ell_{m,1} \lor \ldots \lor \ell_{m,n_m})\],

and are asking if there is a satisfying assignment to all the variables.

The 3-CNF-SAT (short: 3-SAT) problem is basically the same thing, but with the stipulation that every clause contains exactly three literals:

\[(\ell_{1,1} \lor \ell_{1,2} \lor \ell_{1,3}) \land \ldots \land (\ell_{m,1} \lor \ell_{m,2} \lor \ell_{m,3})\].
Let’s start with a boring one.

Recall CNF-SAT: we have a conjunction of disjunctions like

\[(\ell_{1,1} \lor \ldots \lor \ell_{1,n_1}) \land \ldots \land (\ell_{m,1} \lor \ldots \lor \ell_{m,n_m}),\]

and are asking if there is a satisfying assignment to all the variables.

The 3-CNF-SAT (short: 3-SAT) problem is basically the same thing, but with the stipulation that every clause contains exactly three literals:

\[(\ell_{1,1} \lor \ell_{1,2} \lor \ell_{1,3}) \land \ldots \land (\ell_{m,1} \lor \ell_{m,2} \lor \ell_{m,3}).\]

(We impose additional restrictions on the formulae, so this looks... easier? Either way, it turns out to sometimes be convenient for further reductions.)
To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:
To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:

- Given a CNF-SAT instance $\varphi$, construct a 3-SAT instance $\psi$.
To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:

- Given a CNF-SAT instance $\varphi$, construct a 3-SAT instance $\psi$.
- Show that we did this in polynomial time. (That it is polynomial-size follows: we can only write polynomial amounts of data in polynomial time.)
Reducing between NP-complete problems

To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:

- Given a CNF-SAT instance $\varphi$, construct a 3-SAT instance $\psi$.
- Show that we did this in polynomial time. (That it is polynomial-size follows: we can only write polynomial amounts of data in polynomial time.)
- Show that this always produces the right answer:
To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:

- Given a CNF-SAT instance $\varphi$, construct a 3-SAT instance $\psi$.
- Show that we did this in polynomial time. (That it is polynomial-size follows: we can only write polynomial amounts of data in polynomial time.)
- Show that this always produces the right answer:
  - If the answer should be “yes” – that is, $\varphi$ was a “yes” instance of CNF-SAT – then $\psi$ should be a “yes” instance of 3-SAT.
To show that 3-SAT is NP-complete, we need to demonstrate a polynomial-time reduction from SAT to 3-SAT. The following simple architecture works most of the time:

- Given a CNF-SAT instance \( \varphi \), construct a 3-SAT instance \( \psi \).
- Show that we did this in polynomial time. (That it is polynomial-size follows: we can only write polynomial amounts of data in polynomial time.)
- Show that this always produces the right answer:
  - If the answer should be “yes” – that is, \( \varphi \) was a “yes” instance of CNF-SAT – then \( \psi \) should be a “yes” instance of 3-SAT.
  - If the answer should be “no” – that is, \( \varphi \) was a “no” instance of CNF-SAT – then \( \psi \) should be a “no” instance of 3-SAT.
The problem with general CNF-SAT is that clauses can sometimes have either too few or too many literals.
The problem with general CNF-SAT is that clauses can sometimes have either too few or too many literals.

**Idea:** Create a bunch of clauses to represent every problematic clause. Make sure that the new collection of clauses is **satisfiable** if and only if the original clause is.
The problem with general CNF-SAT is that clauses can sometimes have either too few or too many literals.

Idea: Create a bunch of clauses to represent every problematic clause. Make sure that the new collection of clauses is satisfiable if and only if the original clause is.

Is this alone sufficient? That is, will we get a correct reduction if we just ensure there is a correspondence between $\varphi$ and $\psi$ clauses like that?
No: We want (all $\varphi$ clauses satisfiable simultaneously) $\iff$ (all $\psi$ clauses satisfiable simultaneously).
No: We want (all $\varphi$ clauses satisfiable simultaneously) $\iff$ (all $\psi$ clauses satisfiable simultaneously).

For example, take $\varphi = (x_1) \land (\neg x_1)$.
No: We want (all $\varphi$ clauses satisfiable simultaneously) $\iff$ (all $\psi$ clauses satisfiable simultaneously).

For example, take $\varphi = (x_1) \land (\neg x_1)$.

Then take $\psi = (y_1 \lor y_2 \lor y_3) \land (y_4 \lor y_5 \lor y_6)$.

$\varphi$ is not satisfiable; but $\psi$ is. Clauses are satisfiable in isolation $\not\iff$ satisfiable simultaneously!
No: We want (all $\varphi$ clauses satisfiable simultaneously) $\iff$ (all $\psi$ clauses satisfiable simultaneously).

For example, take $\varphi = (x_1) \land (\neg x_1)$.

Then take $\psi = (y_1 \lor y_2 \lor y_3) \land (y_4 \lor y_5 \lor y_6)$.

$\varphi$ is not satisfiable; but $\psi$ is. Clauses are satisfiable in isolation $\varphi$ satisfiable simultaneously!

Fix: Make source and target clauses satisfiable by the same assignment, to preserve how clauses interact.
No: We want (all \( \varphi \) clauses satisfiable simultaneously) \( \iff \) (all \( \psi \) clauses satisfiable simultaneously).

For example, take \( \varphi = (x_1) \land (\neg x_1) \).

Then take \( \psi = (y_1 \lor y_2 \lor y_3) \land (y_4 \lor y_5 \lor y_6) \).

\( \varphi \) is not satisfiable; but \( \psi \) is. Clauses are satisfiable in isolation \( \not\iff \) satisfiable simultaneously!

Fix: Make source and target clauses satisfiable by the same assignment, to preserve how clauses interact. If we need extra variables, make them not interact with other source clauses’ image at all.
Too few literals: Just pad out with fresh variables.

\[(\ell_1 \lor \ell_2) \rightarrow (\ell_1 \lor \ell_2 \lor \ell)\]
Too few literals: Just pad out with fresh variables.

\[(\ell_1 \lor \ell_2) \mapsto (\ell_1 \lor \ell_2 \lor y) \land (\ell_1 \lor \ell_2 \lor \neg y)\]

Make sure that the clause(s) can’t be satisfied by \(y\) alone!
Too few literals: Just pad out with fresh variables.

$$(\ell_1 \lor \ell_2) \mapsto (\ell_1 \lor \ell_2 \lor y) \land (\ell_1 \lor \ell_2 \lor \neg y)$$

Make sure that the clause(s) can’t be satisfied by $y$ alone!

$$(\ell) \mapsto (\ell \lor y \lor z) \land (\ell \lor \neg y \lor z) \land (\ell \lor y \lor \neg z) \land (\ell \lor \neg y \lor \neg z).$$
Too many literals: Split clause. Say the input is

\[(\ell_1 \lor \ldots \lor \ell_k)\].

Introduce \(k - 3\) fresh variables \(z_3, \ldots, z_{k-1}\).
Too many literals: Split clause. Say the input is

$$(\ell_1 \lor \ldots \lor \ell_k).$$

Introduce $k - 3$ fresh variables $z_3, \ldots, z_{k-1}$.

Interpretation: $z_i$ will be true iff the clause had to be satisfied with some literal with index $\geq i$. Accordingly, $\neg z_i$ will be true iff it was not be satisfied with any literal with index $\geq i$. 

Check that this works!
Too many literals: Split clause. Say the input is
\[(\ell_1 \lor \ldots \lor \ell_k)\].

Introduce \(k - 3\) fresh variables \(z_3, \ldots, z_{k-1}\).

Interpretation: \(z_i\) will be true iff the clause had to be satisfied with some literal with index \(\geq i\). Accordingly, \(-z_i\) will be true iff it was not be satisfied with any literal with index \(\geq i\).

Output:
\[\begin{align*}
(\ell_1 \lor \ell_2 \lor z_3) & \land (\ell_3 \lor -z_3 \lor z_4) \\
& \land \ldots \\
& \land (\ell_{k-2} \lor -z_{k-2} \lor z_{k-1}) \\
& \land (\ell_{k-1} \lor \ell_k \lor -z_{k-1}).
\end{align*}\]

Check that this works!
Check: The original $k$-literal clause had a satisfying assignment $A \iff$ an extension of $A$ works for the $k - 2$ clauses we generated.
Check: The original \( k \)-literal clause had a satisfying assignment \( A \iff \text{an extension of } A \) works for the \( k - 2 \) clauses we generated.

\( \Rightarrow \): Let \( i \) be the first \( i \) such that \( \ell_i \) is true. Set \( z_j = T \) for all \( j \leq i \), \( z_j = F \) for all \( j > i \). If \( i \leq 2 \), then all \( z_j \) are false, and so the first output clause is satisfied by \( \ell_i \) and the rest are satisfied by the appropriate \( \neg z_j \). Otherwise, every clause that contains \( z_j \) for \( j \leq i \) is satisfied by \( z_j \), the one that contains \( \ell_i \) and \( \neg z_i \) is satisfied by \( \ell_i \), and the remaining ones are satisfied by \( \neg z_j \) for the appropriate \( j > i \).
“⇐”: Claim that some $\ell_i$ is true; then the input clause is satisfied by that literal. Suppose not. Then all of the $\ell_i$ are false, and so the output clauses simplify to
\[
(z_3) \land (\neg z_3 \lor z_4) \land \ldots \land (\neg z_{k-2} \lor z_{k-1}) \land (\neg z_{k-1}).
\]
This is not satisfiable (note that $(\neg x \lor y)$ is $x \to y$). □
3-SAT is NP-complete

We created a polynomial number of clauses for each original clause with only straightforward computations, and can check that we have a satisfying assignment to the resulting CNF iff we have one to the original CNF.

Hence done, and 3-SAT is NP-complete. □
We created a polynomial number of clauses for each original clause with only straightforward computations, and can check that we have a satisfying assignment to the resulting CNF iff we have one to the original CNF.

Hence done, and 3-SAT is NP-complete. □

This approach of taking discrete parts of the source problem and converting them to parts of the target problem, which interact with each other in a controlled manner, is common. We call the parts we construct in the target problem gadgets.
Intermission

Let’s take a short break. **Exercise:** Try converting

\[(\neg x \lor \neg y) \land (\neg x \lor a) \land (\neg x \lor y \lor \neg a \lor \neg b \lor c) \land (x \lor \neg c) \land (x \lor c)\]

to 3-SAT, and find two distinct satisfying assignments.
Given an undirected graph $G = (V, E)$, an independent set is a set of vertices $I \subseteq V$ such that no two vertices in $I$ share an edge: $\forall v, v' \in I. v \neq v' \land \{v, v'\} \in E$. 
The Independent Set problem

Given an undirected graph $G = (V, E)$, an independent set is a set of vertices $I \subseteq V$ such that no two vertices in $I$ share an edge: $\forall v, v' \in I. v \neq v' \land \{v, v'\} \in E$. 
The Independent Set problem

Given an undirected graph \( G = (V, E) \), an independent set is a set of vertices \( I \subseteq V \) such that no two vertices in \( I \) share an edge: \( \forall v, v' \in I. v \neq v' \land \{v, v'\} \in E. \)

The **INDEPENDENT SET** decision problem asks the following: given a graph \( G = (V, E) \) and an integer \( k \), is there an independent set of at least \( k \) vertices?
We’d like to show that INDEPENDENT SET is NP-complete.
We’d like to show that INDEPENDENT SET is NP-complete. In general, finding a polynomial-time reduction like this is not straightforward.
We’d like to show that INDEPENDENT SET is NP-complete.

In general, finding a polynomial-time reduction like this is not straightforward.

Here is a not-quite-rigorous intuitive framework that I have personally found useful.
Many NP-complete problems ask for the existence of a structured solution, which typically becomes the certificate.
Many NP-complete problems ask for the existence of a **structured solution**, which typically becomes the certificate.

This solution consists of many smaller components. For instance, for INDEPENDENT SET, the solution is an independent set, which amounts to a choice of whether we put in each vertex of the graph.
Many NP-complete problems ask for the existence of a structured solution, which typically becomes the certificate. This solution consists of many smaller components. For instance, for INDEPENDENT SET, the solution is an independent set, which amounts to a choice of whether we put in each vertex of the graph.

We can imagine a process of picking these components one by one, as if we ran a greedy algorithm. Every time we pick something, some future choices are ruled out: e.g. including vertices adjacent to the vertex we just picked.
If at some point we run out of choices, we get stuck and have to backtrack: undo some number of choices, and try adding a different component instead.

Otherwise, at some point we’ve chosen everything there is to choose, and found a certificate that we have a “yes” instance.
If at some point we run out of choices, we get stuck and have to backtrack: undo some number of choices, and try adding a different component instead.

Otherwise, at some point we’ve chosen everything there is to choose, and found a certificate that we have a “yes” instance.

(The search tree we get in this fashion looks a lot like an NTM execution cone, and is in general of exponential size.)
Sometimes, there isn’t a canonical way to break down the solution into components. For instance, what are the components of CNF-SAT?
Sometimes, there isn’t a canonical way to break down the solution into components. For instance, what are the components of CNF-SAT?

We could think of it as a process of picking assignments to variables one-by-one: first we say that $x_1$ is set to $T$, then we try to set $x_3$ to $F$, ...
Sometimes, there isn’t a canonical way to break down the solution into components. For instance, what are the components of CNF-SAT?

We could think of it as a process of picking assignments to variables one-by-one: first we say that $x_1$ is set to $T$, then we try to set $x_3$ to $F$, ... Then, an assignment to a variable is ruled out if we can immediately see it won’t lead to a solution: e.g. we just made some clause unsatisfiable by picking the opposite of all of its literals.
Alternatively, we could think of it as a process of **satisfying clauses** one-by-one. Every clause needs to be satisfied by some literal in it. So first we say that we satisfy \((x_1 \lor \neg x_3 \lor x_7)\) by setting \(x_3\) to \(F\), then we satisfy \((x_3 \lor x_4)\) by setting \(x_4\) to \(T\), \ldots
Alternatively, we could think of it as a process of satisfying clauses one-by-one. Every clause needs to be satisfied by some literal in it. So first we say that we satisfy \((x_1 \lor \neg x_3 \lor x_7)\) by setting \(x_3\) to \(F\), then we satisfy \((x_3 \lor x_4)\) by setting \(x_4\) to \(T\), …

Then, a way of satisfying a clause is ruled out if we previously did something that contradicts it: e.g. we already satisfied another clause by setting a variable so that this literal is rendered false.
Both views are valid and useful, and give rise to reductions that we will encounter.
Both views are valid and useful, and give rise to reductions that we will encounter.

For INDEPENDENT SET, we will use the second one.
Either way, once we have decided on a search tree structure for both the source and the target problem, we can build gadgets that simulate the structure of a source choice using target choices. Sometimes, we have to plug multiple of them together.
Either way, once we have decided on a search tree structure for both the source and the target problem, we can build gadgets that simulate the structure of a source choice using target choices. Sometimes, we have to plug multiple of them together.

For instance, imagine the source problem lets you choose between three different options:

A     B     C
On the other hand, the target problem has choices that lead to two options (which don’t interact with anything else), choices where your hand is forced (you only get one option), and choices that always lead to you getting stuck:
So can model the choice structure of the source problem with three choices of the target problem:
Towards NP-completeness of INDEPENDENT SET

As said earlier, we will use the clause-satisfying view of 3-SAT.

Example: input formula

\[(x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor x_2 \lor x_3) \land (\neg x_2 \lor x_3 \lor x_4).\]

Search tree if we try to satisfy clauses left to right:
We want to build a similar decision tree in INDEPENDENT SET, seen as the problem of repeatedly picking vertices to add to the independent set.
We want to build a similar decision tree in INDEPENDENT SET, seen as the problem of repeatedly picking vertices to add to the independent set.

As we picked one of three literals in each clause in 3-SAT, we’ll want to create three vertices that we can pick from:
We want to build a similar decision tree in INDEPENDENT SET, seen as the problem of repeatedly picking vertices to add to the independent set.

As we picked one of three literals in each clause in 3-SAT, we’ll want to create three vertices that we can pick from:

\[(X_1 \lor \neg X_3 \lor X_7)\]
Once we have already chosen to satisfy the $i$th clause using, say, $x_1$, we no longer need or want to choose another literal to satisfy the same clause.

In INDEPENDENT SET, we can make a choice (of vertex) block a subsequent choice (of vertex) by introducing edges between them, so we add edges between all the vertices of a clause:
Once we have already chosen to satisfy the $i$th clause using, say, $x_1$, we no longer need or want to choose another literal to satisfy the same clause.

In INDEPENDENT SET, we can make a choice (of vertex) block a subsequent choice (of vertex) by introducing edges between them, so we add edges between all the vertices of a clause:
Also, if we choose to satisfy a clause by, say, $\neg x_3$, then we can’t satisfy any other clauses using $x_3$ anymore: after all, in our assignment, $x_3$ would be set to $F$.

This is another straightforward case of two choices being mutually exclusive, which we can realise using an edge. So we add edges between any two contradictory literals:
Also, if we choose to satisfy a clause by, say, $\neg x_3$, then we can’t satisfy any other clauses using $x_3$ anymore: after all, in our assignment, $x_3$ would be set to $F$.

This is another straightforward case of two choices being mutually exclusive, which we can realise using an edge. So we add edges between any two contradictory literals:

\[
\begin{align*}
&(x_1 \lor \neg x_3 \lor x_7) \\
&\text{ith clause} \\
&(\neg x_1 \lor x_2 \lor x_3) \\
&\text{jth clause}
\end{align*}
\]

\[
\begin{align*}
i &: \neg x_3 \\
j &: x_2 \\
i &: x_1 \\
j &: \neg x_1 \\
i &: x_7 \\
j &: x_3
\end{align*}
\]
Finishing up

Now, set the minimum independent set size $k$ to the number of clauses, so we need to pick a literal to make every clause true.
Finishing up

Now, set the minimum independent set size $k$ to the number of clauses, so we need to pick a literal to make every clause true.

And we’re actually done! This conversion was evidently polynomial-time. Just need to check:
Now, set the minimum independent set size $k$ to the number of clauses, so we need to pick a literal to make every clause true.

And we’re actually done! This conversion was evidently polynomial-time. Just need to check:

- If we had a satisfying assignment to the 3-SAT instance, then we have an independent set of size $k$ in the graph.
Now, set the minimum independent set size $k$ to the number of clauses, so we need to pick a literal to make every clause true.

And we’re actually done! This conversion was evidently polynomial-time. Just need to check:

- If we had a satisfying assignment to the 3-SAT instance, then we have an independent set of size $k$ in the graph.
- If we have an independent set of size $k$, then we have a satisfying assignment to the 3-SAT instance the graph came from.
Now, set the minimum independent set size $k$ to the number of clauses, so we need to pick a literal to make every clause true.

And we’re actually done! This conversion was evidently polynomial-time. Just need to check:

- If we had a satisfying assignment to the 3-SAT instance, then we have an independent set of size $k$ in the graph.
- If we have an independent set of size $k$, then we have a satisfying assignment to the 3-SAT instance the graph came from.

Conclude: Independent Set is NP-complete. □
Given a graph $G = (V, E)$, a **vertex $k$-colouring** of this graph is an assignment of one of $k$ colours to each of the vertices of $G$ such that no two adjacent vertices share a colour.
Given a graph $G = (V, E)$, a vertex $k$-colouring of this graph is an assignment of one of $k$ colours to each of the vertices of $G$ such that no two adjacent vertices share a colour.

Formally, a map $c : V \rightarrow [k]$ such that $\forall \{u, v\} \in E, c(u) \neq c(v)$.
Given a graph $G = (V, E)$, a **vertex $k$-colouring** of this graph is an assignment of one of $k$ colours to each of the vertices of $G$ such that no two adjacent vertices share a colour.

Formally, a map $c : V \rightarrow [k]$ such that $\forall \{u, v\} \in E, c(u) \neq c(v)$. 

For which $k$ can we colour a given graph?
Given a graph \( G = (V, E) \), a **vertex \( k \)-colouring** of this graph is an assignment of one of \( k \) colours to each of the vertices of \( G \) such that no two adjacent vertices share a colour.

Formally, a map \( c : V \rightarrow [k] \) such that \( \forall \{u, v\} \in E, c(u) \neq c(v) \).

For which \( k \) can we colour a given graph?

The **Graph 3-colouring** decision problem asks: given a graph, can it be vertex 3-coloured?
Clearly, this problem is in NP. Let’s build some intuition for it.
Clearly, this problem is in NP. Let’s build some intuition for it. Suppose our three colours are with red, green and blue. What colours could we give the unpainted (white) vertex here?
Clearly, this problem is in NP. Let’s build some intuition for it.
Suppose our three colours are with red, green and blue.
What colours could we give the unpainted (white) vertex here?

What about this?
Clearly, this problem is in NP. Let’s build some intution for it. Suppose our three colours are with red, green and blue. What colours could we give the unpainted (white) vertex here?

What about this?
What colours could we assign to the two blank vertices here?
What colours could we assign to the two blank vertices here?
What colours could we assign to the two blank vertices here?

What about this chain?
What colours could we assign to the two blank vertices here?

What about this chain?
What colours could we assign to the two blank vertices here?

What about this chain?
What colours could we assign to the two blank vertices here?

What about this chain?
What colours could we assign to the two blank vertices here?

What about this chain?
What about this?

There is no 3-colouring!

Doesn't have to be a complete graph:
What about this?
What about this?

There is no 3-colouring!
What about this?

There is no 3-colouring!

Doesn’t have to be a complete graph:
Goal: Show that GRAPH 3-COLOURING is NP-complete by reducing from 3-SAT.
Goal: Show that GRAPH 3-COLOURING is NP-complete by reducing from 3-SAT.

As before, we want to use the search tree structure to drive our gadget design. However, this time, we’ll analyse 3-SAT as consisting of a series of choices of assignments to variables, which gets stuck whenever we make some clause unsatisfiable.
Goal: Show that GRAPH 3-COLOURING is NP-complete by reducing from 3-SAT.

As before, we want to use the search tree structure to drive our gadget design. However, this time, we’ll analyse 3-SAT as consisting of a series of choices of assignments to variables, which gets stuck whenever we make some clause unsatisfiable.

How do we build up a solution to 3-colouring?
**Goal:** Show that **GRAPH 3-COLOURING** is **NP-complete** by reducing from **3-SAT**.

As before, we want to use the search tree structure to drive our gadget design. However, this time, we’ll analyse 3-SAT as consisting of a *series of choices of assignments to variables*, which gets stuck whenever we make some clause unsatisfiable.

How do we build up a solution to 3-colouring?

Natural framing of what we did while building intuition: repeatedly *pick a colour for some vertex*, getting stuck when a vertex can’t be consistently coloured.
First idea: In our view of 3-SAT, we pick either $T$ or $F$ for each variable.
First idea: In our view of 3-SAT, we pick either $T$ or $F$ for each variable.

So, in our view of 3-colouring, identify some vertices with variables, and say that, for instance, colouring it green means we make the variable $T$, and red means we make the variable $F$. 
First idea: In our view of 3-SAT, we pick either $T$ or $F$ for each variable.

So, in our view of 3-colouring, identify some vertices with variables, and say that, for instance, colouring it green means we make the variable $T$, and red means we make the variable $F$.

Could something like this work?
**First idea:** In our view of 3-SAT, we pick either $T$ or $F$ for each variable.

So, in our view of 3-colouring, identify some vertices with variables, and say that, for instance, colouring it green means we make the variable $T$, and red means we make the variable $F$.

Could something like this work?

**No:** 3-colouring is invariant under permuting colours. That is, if we take a valid colouring and globally swap green and red, we still have a valid colouring.

But this is not true for $T$ and $F$ in 3-SAT assignments! (Consider $(x_1 \lor x_2 \lor x_3)$ with all true.)
The chain-of-diamonds graph earlier suggested something helpful: we do have some degree of control over when two vertices are the same colour.
Overcoming permutation invariance

The chain-of-diamonds graph earlier suggested something helpful: we do have some degree of control over when two vertices are the same colour.

So, second idea: introduce two special vertices to our graph as references. Whatever colour the first one gets will mean true; whatever colour the second one gets will mean false.

(While we’re at it, might as well “save” the third colour in a reference vertex too.)
Overcoming permutation invariance

The chain-of-diamonds graph earlier suggested something helpful: we do have some degree of control over when two vertices are the same colour.

So, second idea: introduce two special vertices to our graph as references. Whatever colour the first one gets will mean true; whatever colour the second one gets will mean false.

(While we’re at it, might as well “save” the third colour in a reference vertex too.)
The chain-of-diamonds graph earlier suggested something helpful: we do have some degree of control over when two vertices are the same colour.

So, second idea: introduce two special vertices to our graph as references. Whatever colour the first one gets will mean true; whatever colour the second one gets will mean false.

(While we’re at it, might as well “save” the third colour in a reference vertex too.)
The chain-of-diamonds graph earlier suggested something helpful: we do have some degree of control over when two vertices are the same colour.

So, second idea: introduce two special vertices to our graph as references. Whatever colour the first one gets will mean true; whatever colour the second one gets will mean false.

(While we’re at it, might as well “save” the third colour in a reference vertex too.)
Let’s do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(X_1 \lor \neg X_3 \lor X_4) \land (\neg X_1 \lor \neg X_2 \lor \neg X_3) \land (\neg X_2 \lor X_3 \lor X_4).\]
Let's do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(X_1 \lor \neg X_3 \lor X_4) \land (\neg X_1 \lor \neg X_2 \lor \neg X_3) \land (\neg X_2 \lor X_3 \lor X_4).\]

- \(X_1 \mapsto T\)
Let’s do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(X_1 \lor \neg X_3 \lor X_4) \land (\neg X_1 \lor \neg X_2 \lor \neg X_3) \land (\neg X_2 \lor X_3 \lor X_4).\]

- \(X_1 \mapsto T\)
- \(X_2 \mapsto T\)
3-SAT as a series of assignments to variables

Let’s do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor x_3 \lor x_4).\]

- \(x_1 \mapsto T\)
- \(x_2 \mapsto T\)
- \(x_3 \mapsto T.\) Stuck on clause 2!
Let’s do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor x_3 \lor x_4)\].

- \(x_1 \mapsto T\)
- \(x_2 \mapsto T\)
- \(x_3 \mapsto F\).
Let’s do a test run of how we want the implicit solution to 3-SAT to be picked.

\[(X_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor x_3 \lor x_4)\].

- \(X_1 \mapsto T\)
- \(X_2 \mapsto T\)
- \(X_3 \mapsto F\)
- \(X_4 \mapsto T\)
Need to capture two features of the search tree:
Need to capture two features of the search tree:

- Picking a truth value for a variable, and
Need to capture two features of the search tree:

- Picking a truth value for a variable, and
- Getting stuck when we happened to rule out all ways of making some clause true.
Replicating the choice structure (2)

The first one ("variable gadget") is simple enough:

Each $x_i$ can't get colour $B$, so must be $T$ or $F$. 

![Diagram showing the variable gadget with nodes T, B, X1, and X2 connected as described.](image-url)
The first one ("variable gadget") is simple enough:

Each \( x_i \) can’t get colour \( B \), so must be \( T \) or \( F \).
Add \( \neg x_i \) vertex for convenience.
What about the second one?

Need colouring to get stuck when we can no longer satisfy a clause.
What about the second one?

Need colouring to get stuck when we can no longer satisfy a clause.

When does colouring get stuck?
What about the second one?

Need colouring to get stuck when we can no longer satisfy a clause.

When does colouring get stuck?

As we saw before: when a vertex needs to be coloured but already has vertices in all three colours next to itself.
A clause can no longer be satisfied when we’ve made all literals in it false.
A clause can no longer be satisfied when we’ve made all literals in it false.

That is, e.g. for \((x_1 \lor \neg x_3 \lor x_4)\), the vertices we labelled \(x_1\), \(\neg x_3\) and \(x_4\) have all been coloured the same as \(F\).
A clause can no longer be satisfied when we’ve made all literals in it false.

That is, e.g. for \((x_1 \lor \neg x_3 \lor x_4)\), the vertices we labelled \(x_1\), \(\neg x_3\) and \(x_4\) have all been coloured the same as \(F\).

Want a gadget that forces three vertices to have different colours exactly when that is the case.
**Idea:** Use the triangle construction to create a vertex that is forced to have colour $T$ iff the first literal has colour $F$, and another that is forced to have colour $B$ iff the third literal has colour $F$. Use the second literal as is.

Then connect these three vertices to a new vertex associated with the clause.
**Idea:** Use the triangle construction to create a vertex that is forced to have colour $T$ iff the first literal has colour $F$, and another that is forced to have colour $B$ iff the third literal has colour $F$. Use the second literal as is.

Then connect these three vertices to a new vertex associated with the clause.

\[(\ell_1 \lor \ell_2 \lor \ell_3) \iff
\begin{array}{c}
T \\
\ell_1 \\
\ell_2 \\
\ell_3 \\
B
\end{array}\]
Proving the reduction correct (1)

Need to show: have a 3-colouring of this graph \(\Leftrightarrow\) we have a satisfying assignment.
Need to show: have a 3-colouring of this graph $\iff$ we have a satisfying assignment.

$\implies$: Set each variable based on which of the $T$ and $F$ vertices its colours agrees with.

This is a satisfying assignment: claim that each clause is satisfied.
Need to show: have a 3-colouring of this graph $\iff$ we have a satisfying assignment.

$\implies$: Set each variable based on which of the $T$ and $F$ vertices its colours agrees with.

This is a satisfying assignment: claim that each clause is satisfied.

Indeed, since we coloured the top of the clause gadget, the vertices it was connected to were not all different colours. If $\ell_2$ was coloured as $T$, then done. Otherwise, $\ell_2$ was coloured as $F$. So either one of the other two was also coloured as $F$, or the other two are coloured the same.
One of the other two coloured as $F$: then either $\ell_1$ or $\ell_3$ can't have been coloured as $F$, so done.
One of the other two coloured as $F$: then either $l_1$ or $l_3$ can’t have been coloured as $F$, so done.

The other two are coloured the same: they can’t both be $T$ or $B$ (since one is adjacent to $T$, and the other to $B$), so must both be $F$. So both $l_1$ and $l_2$ must in fact have been coloured $T$. 
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.
Proving the reduction correct (3)

“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$. 
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$.

Suppose $\ell_2$ is $T$. 

“$$\Leftarrow$$”: Pick arbitrary colours for \(T, F\) and \(B\), and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured \(T\).

Suppose \(\ell_2\) is \(T\).

Suppose \(\ell_1\) is \(F\), and \(\ell_3\) is \(T\).
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$.

Suppose $\ell_2$ is $T$.

Suppose $\ell_1$ is $F$, and $\ell_3$ is $T$.

Then the vertex above $\ell_1$ is $B$, and the vertex above $\ell_3$ is...
“\(\iff\)”: Pick arbitrary colours for \(T, F\) and \(B\), and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured \(T\).

Suppose \(\ell_2\) is \(T\).

Suppose \(\ell_1\) is \(F\), and \(\ell_3\) is \(T\).

Then the vertex above \(\ell_1\) is \(B\), and the vertex above \(\ell_3\) is... \(F\).

Wait.
This doesn’t actually work! And that’s why proving correctness is important.
This doesn’t actually work! And that’s why proving correctness is important.

What went wrong? We made it so that the gadget couldn’t be coloured when all literals were false, but we never stopped to make sure that it could be coloured when this is not the case.
This doesn’t actually work! And that’s why proving correctness is important.

What went wrong? We made it so that the gadget couldn’t be coloured when all literals were false, but we never stopped to make sure that it could be coloured when this is not the case.

In particular, when $\ell_3$ is $T$, the colouring of the node above it is still forced to a single choice ($F$). Circumstances then conspire so that we can assign the other two literals in such a way that once again we get three different colours next to the peak.
Can we build something on top of $\ell_3$ so that if it’s coloured $T$, the node next to the peak has two different choices?
Can we build something on top of $\ell_3$ so that if it’s coloured $T$, the node next to the peak has two different choices?

Yes:

$$(\ell_1 \lor \ell_2 \lor \ell_3) \rightarrow \ell_3$$
Can we build something on top of $\ell_3$ so that if it’s coloured $T$, the node next to the peak has two different choices?

Yes:

$\left( \ell_1 \lor \ell_2 \lor \ell_3 \right)$
Can we build something on top of $\ell_3$ so that if it’s coloured $T$, the node next to the peak has two different choices?

Yes:

$$ (\ell_1 \lor \ell_2 \lor \ell_3) \quad \rightarrow $$
Can we build something on top of $\ell_3$ so that if it’s coloured $T$, the node next to the peak has two different choices?

Yes:

$$(\ell_1 \vee \ell_2 \vee \ell_3)$$
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$. 
“$\Leftarrow$”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$.

Whatever $\ell_3$ is, we can colour the vertex above $\ell_3$ that is connected to $F$ as $T$: either it’s $F$, and the vertices above it must be $B$ and then $T$, or it’s $T$, the vertex above it can be $B$, and the vertex connected to $F$ and the peak must then be $T$. 
“⇐”: Pick arbitrary colours for $T$, $F$ and $B$, and colour the literal vertices according to the assignment.

Every clause is satisfied, so at least one of its incoming literal vertices must be coloured $T$.

Whatever $\ell_3$ is, we can colour the vertex above $\ell_3$ that is connected to $F$ as $T$: either it’s $F$, and the vertices above it must be $B$ and then $T$, or it’s $T$, the vertex above it can be $B$, and the vertex connected to $F$ and the peak must then be $T$.

If $\ell_2$ is $T$, we are then done, as two vertices adjacent to the peak are the same colour and so we can pick a colour for the peak and missing vertex in the gadget.
So suppose ℓ₂ is F. Then, either ℓ₁ must be T, in which case we can colour the vertex above it F as well; or ℓ₁ is F and ℓ₃ is T, in which case we can colour the vertex above ℓ₁ B, and the ones above ℓ₃ with F and B in order, and so the peak can be coloured T.
So suppose $\ell_2$ is $F$. Then, either $\ell_1$ must be $T$, in which case we can colour the vertex above it $F$ as well; or $\ell_1$ is $F$ and $\ell_3$ is $T$, in which case we can colour the vertex above $\ell_1$ $B$, and the ones above $\ell_3$ with $F$ and $B$ in order, and so the peak can be coloured $T$.

In this fashion, we proceed to colour all clauses, and thus have a 3-colouring. $\square$

Clearly, everything is polynomial. So GRAPH 3-COLOURING is NP-complete.