SAT and the class NP
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Cook reductions

Throughout the course, we've been using **reductions** to come up with new algorithms: rather than try to attack a problem directly, we try to express it in terms of another problem we already know how to solve.

For example, we solved the circulation problem by reducing it to a maximum flow problem. (And vertex capacities, rational weights, bipartite matchings and so on and so on...)

Cook reductions

Throughout the course, we've been using **reductions** to come up with new algorithms: rather than try to attack a problem directly, we try to express it in terms of another problem we already know how to solve.

For example, we solved the circulation problem by reducing it to a maximum flow problem. (And vertex capacities, rational weights, bipartite matchings and so on and so on...)

Importantly, none of our reductions have depended on how we solve the problem we're reducing to. If we used Ford-Fulkerson to solve the circulation problem instead of Edmonds-Karp, nothing would change but the time analysis. Let's make this idea formal.

Cook reductions

Throughout the course, we've been using **reductions** to come up with new algorithms: rather than try to attack a problem directly, we try to express it in terms of another problem we already know how to solve.

For example, we solved the circulation problem by reducing it to a maximum flow problem. (And vertex capacities, rational weights, bipartite matchings and so on and so on...)

Importantly, none of our reductions have depended on how we solve the problem we're reducing to. If we used Ford-Fulkerson to solve the circulation problem instead of Edmonds-Karp, nothing would change but the time analysis. Let's make this idea formal.

Suppose we have a polynomial-time algorithm for problem X which calls a polynomial-time algorithm for Y as a subroutine. Then **regardless of whether or not we actually have a polynomial-time algorithm for** Y, we call this a **Cook reduction** from X to Y and write $X \leq_c Y$.

Suppose we have a polynomial-time algorithm for problem X which calls a polynomial-time algorithm for Y as a subroutine. Then **regardless of whether or not we actually have a polynomial-time algorithm for Y**, we call this a **Cook reduction** from X to Y and write $X \leq_c Y$.

A *little* more formally: an **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

Suppose we have a polynomial-time algorithm for problem X which calls a polynomial-time algorithm for Y as a subroutine. Then **regardless of whether or not we actually have a polynomial-time algorithm for Y**, we call this a **Cook reduction** from X to Y and write $X \leq_c Y$.

A *little* more formally: an **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

An oracle is explicitly a cheat — we are washing our hands of any responsibility for actually solving problem Y. Maybe a wizard did it.
Or a library function whose code is indistinguishable from wizardry.

Suppose we have a polynomial-time algorithm for problem X which calls a polynomial-time algorithm for Y as a subroutine. Then **regardless of whether or not we actually have a polynomial-time algorithm for** Y, we call this a **Cook reduction** from X to Y and write $X \leq_c Y$.

A *little* more formally: an **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is an algorithm for problem X which, given an input of size s, runs in time poly(s) while making poly(s) calls to an oracle for Y whose input instances are all of size poly(s).

Suppose we have a polynomial-time algorithm for problem X which calls a polynomial-time algorithm for Y as a subroutine. Then **regardless of whether or not we actually have a polynomial-time algorithm for** Y, we call this a **Cook reduction** from X to Y and write $X \leq_c Y$.

A *little* more formally: an **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is an algorithm for problem X which, given an input of size s, runs in time poly(s) while making poly(s) calls to an oracle for Y whose input instances are all of size poly(s).

The point of the definition is: Given a Cook reduction from X to Y, and a polynomial-time algorithm for Y, we get a polynomial-time algorithm for X. We just simulate the oracle using our algorithm for Y.

(The "correct" definition is more complicated, involving so-called oracle Turing machines, but the one above is good enough for our purposes.)

An **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is a poly-time algorithm for problem X which, given an input of size s, makes poly(s) calls to an oracle for Y whose input instances are all of size poly(s). We write $X \leq_c Y$.

If $X \leq_c Y$, then poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.

An **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is a poly-time algorithm for problem X which, given an input of size s, makes poly(s) calls to an oracle for Y whose input instances are all of size poly(s). We write $X \leq_c Y$.

If $X \leq_c Y$, then poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.

As the notation suggests, if $X \leq_c Y$ and $Y \leq_c Z$ then $X \leq_c Z$, so we can build up chains of reductions. For example:

Finding a maximum matching in a bipartite graph $\leq_c$ Finding a maximum flow

$\leq_c$ Solving a linear program.

An **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is a poly-time algorithm for problem X which, given an input of size s, makes poly(s) calls to an oracle for Y whose input instances are all of size poly(s). We write $X \leq_c Y$.

If $X \leq_c Y$, then poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.

As the notation suggests, if $X \leq_c Y$ and $Y \leq_c Z$ then $X \leq_c Z$, so we can build up chains of reductions. For example:

$$\begin{array}{c} \text{Finding a maximum matching} \\ \text{in a bipartite graph} \end{array} \leq_c \text{Finding a maximum flow}$$
$$\leq_c \text{Solving a linear program.}$$

As this example shows, Cook reductions don't always give the best algorithms! In making the definition so general, we have lost the ability to constrain the running time beyond "polynomial".

An **oracle** for Y is a black box which, given an instance of problem Y, outputs a valid solution in $O(1)$ time.

A **Cook reduction** from X to Y is a poly-time algorithm for problem X which, given an input of size s, makes poly(s) calls to an oracle for Y whose input instances are all of size poly(s). We write $X \leq_c Y$.

If $X \leq_c Y$, then poly-time algorithm for $Y \Rightarrow$ poly-time algorithm for X.

As the notation suggests, if $X \leq_c Y$ and $Y \leq_c Z$ then $X \leq_c Z$, so we can build up chains of reductions. For example:

$$\begin{aligned}\text{Finding a maximum matching in a bipartite graph} &\leq_c \text{Finding a maximum flow} \\ &\leq_c \text{Solving a linear program.}\end{aligned}$$

As this example shows, Cook reductions don't always give the best algorithms! In making the definition so general, we have lost the ability to constrain the running time beyond "polynomial".

"Polynomial" can hide a multitude of sins...

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n)$.

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n)$. No, wait, that's not right.

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n^{\mathbf{10}})$.

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n^{10})$. No, a little more...

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n^{10^{100}})$. *There* we go.

We are all sinners, the end is nigh

SODA is one of the best conferences in algorithm design. I work in the field myself, and when I get a paper in there it's a very good day.

In 2013, they published a paper giving a new polynomial-time approximation algorithm for the "max bisection" problem by Austrin, Benabbas, and Georgiou, with better error than any previously known.

The running time? $O(n^{10^{100}})$. *There* we go.

To be clear, this was a genuinely good paper! Just not exactly practical.

Even algorithms with reasonable exponents can hide horrors.

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

The lemma is stated in terms of an approximation parameter ε. The smaller ε is, the more useful the result. Typically you need $\varepsilon \leq 1/100$ or so.

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

The lemma is stated in terms of an approximation parameter ε. The smaller ε is, the more useful the result. Typically you need $\varepsilon \leq 1/100$ or so.

So how large does the graph have to be in terms of ε? $1/\varepsilon$ vertices, perhaps? Maybe $2^{1/\varepsilon}$? That would be unpleasant to deal with.

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

The lemma is stated in terms of an approximation parameter ε. The smaller ε is, the more useful the result. Typically you need $\varepsilon \leq 1/100$ or so.

So how large does the graph have to be in terms of ε? $1/\varepsilon$ vertices, perhaps? Maybe $2^{1/\varepsilon}$? That would be unpleasant to deal with.

Well, the good news is that it's not $2^{1/\varepsilon}$.

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

The lemma is stated in terms of an approximation parameter ε. The smaller ε is, the more useful the result. Typically you need $\varepsilon \leq 1/100$ or so.

So how large does the graph have to be in terms of ε? $1/\varepsilon$ vertices, perhaps? Maybe $2^{1/\varepsilon}$? That would be unpleasant to deal with.

Well, the good news is that it's not $2^{1/\varepsilon}$. It's $2^{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}$, with $1/\varepsilon^5$ twos.

For comparison, there are about 2^{265} atoms in the universe. So it's technically $O(1)$ overhead, but on any conceivable actual input, the algorithm is "solve by brute force"...

Even algorithms with reasonable exponents can hide horrors.

There's a very powerful result for finding subgraphs in large graphs called the Szemerédi regularity lemma. So some graph algorithms start by saying "suppose the input is large enough that we can apply the regularity lemma, and otherwise solve by brute force", which adds a mere $O(1)$ overhead.

The lemma is stated in terms of an approximation parameter ε. The smaller ε is, the more useful the result. Typically you need $\varepsilon \leq 1/100$ or so.

So how large does the graph have to be in terms of ε? $1/\varepsilon$ vertices, perhaps? Maybe $2^{1/\varepsilon}$? That would be unpleasant to deal with.

Well, the good news is that it's not $2^{1/\varepsilon}$. It's $2^{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}$, with $1/\varepsilon^5$ twos.

For comparison, there are about 2^{265} atoms in the universe. So it's technically $O(1)$ overhead, but on any conceivable actual input, the algorithm is "solve by brute force"...

The point is: if you're trying to find an algorithm for X, then just knowing $X \leq_c Y$ doesn't help you much. So why use the formalism?

Weakness as a strength: using reductions to prove hardness

We introduced these thinking: "If $X \leq_c Y$, and we have a polynomial-time algorithm for Y, then we can get a polynomial-time algorithm for X."

Weakness as a strength: using reductions to prove hardness

We introduced these thinking: “If $X \leq_c Y$, and we have a polynomial-time algorithm for Y, then we can get a polynomial-time algorithm for X.”

But this is equivalent to: “If $X \leq_c Y$, and there is **no** polynomial-time algorithm for X, then there is **no** polynomial-time algorithm for Y.”

And this is *incredibly* useful in designing algorithms!

Weakness as a strength: using reductions to prove hardness

We introduced these thinking: "If $X \leq_c Y$, and we have a polynomial-time algorithm for Y, then we can get a polynomial-time algorithm for X."

But this is equivalent to: "If $X \leq_c Y$, and there is **no** polynomial-time algorithm for X, then there is **no** polynomial-time algorithm for Y."

And this is *incredibly* useful in designing algorithms!

If you know the world's brightest minds have tried to solve problem X for decades and failed, and X reduces to your problem, then you can go back to the drawing board *now* rather than trying for a decade yourself first.

Weakness as a strength: using reductions to prove hardness

We introduced these thinking: "If $X \leq_c Y$, and we have a polynomial-time algorithm for Y, then we can get a polynomial-time algorithm for X."

But this is equivalent to: "If $X \leq_c Y$, and there is **no** polynomial-time algorithm for X, then there is **no** polynomial-time algorithm for Y."

And this is *incredibly* useful in designing algorithms!

If you know the world's brightest minds have tried to solve problem X for decades and failed, and X reduces to your problem, then you can go back to the drawing board *now* rather than trying for a decade yourself first.

And the really nice thing is: most of the time, from a practical perspective, there's only one problem X that matters.

Decision problems versus search problems

We focus on **decision problems**, where the desired answer is `Yes` or `No`:

- "Does the input graph contain a matching of size at least k?"
- "Does the input flow network contain a flow of value at least k?"
- "Does the input linear program have a solution of value at least k?"
- "Is the input a composite number?"

Decision problems versus search problems

We focus on **decision problems**, where the desired answer is `Yes` or `No`:

- "Does the input graph contain a matching of size at least k?"
- "Does the input flow network contain a flow of value at least k?"
- "Does the input linear program have a solution of value at least k?"
- "Is the input a composite number?"

Why focus on these rather than just finding the matching? Because:

- We are interested in proving problems are hard, not easy — if it's hard to decide whether something exists, then it's certainly hard to find it!

Decision problems versus search problems

We focus on **decision problems**, where the desired answer is `Yes` or `No`:

- "Does the input graph contain a matching of size at least k?"
- "Does the input flow network contain a flow of value at least k?"
- "Does the input linear program have a solution of value at least k?"
- "Is the input a composite number?"

Why focus on these rather than just finding the matching? Because:

- We are interested in proving problems are hard, not easy — if it's hard to decide whether something exists, then it's certainly hard to find it!
- Decision problems have a simpler theory associated with them.

Decision problems versus search problems

We focus on **decision problems**, where the desired answer is `Yes` or `No`:

- "Does the input graph contain a matching of size at least k?"
- "Does the input flow network contain a flow of value at least k?"
- "Does the input linear program have a solution of value at least k?"
- "Is the input a composite number?"

Why focus on these rather than just finding the matching? Because:

- We are interested in proving problems are hard, not easy — if it's hard to decide whether something exists, then it's certainly hard to find it!
- Decision problems have a simpler theory associated with them.
- It's rare for the decision problem to be easy while the search problem is hard, and often there are easy Cook reductions between them. (See the problem sheet for some examples.)

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

- "Does the input graph contain a matching of size at least k?" is in NP, since we can easily verify that a collection of edges belongs to the input graph and is a matching.

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

- "Does the input flow network contain a flow of value at least k?" is in NP, since we can easily verify that a function is a valid flow with value at least k.

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

- "Does the input linear program have a solution of value at least k?" is in NP, since we can easily verify that a solution is feasible and has value at least k.

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

- "Is the input a composite number?" is in NP, since given an input x and a pair of integers y and z, we can easily verify that $x = yz$ and $y, z > 1$.

Almost every decision problem you run into in the real world can be formulated as a problem in NP.

The class NP

Within decision problems, we will focus on problems where we can easily verify a `Yes` answer.

Formally, **NP** is the class of all decision problems X with the following property: There is a polynomial-time algorithm `Verify` such that if and only if x is a `Yes` instance of X, then there is some bit string w (called a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Think of these problems as looking for a needle in a haystack: you might not be able to find the needle, but you know it when you see it. E.g.:

- "Is the input a composite number?" is in NP, since given an input x and a pair of integers y and z, we can easily verify that $x = yz$ and $y, z > 1$.

Almost every decision problem you run into in the real world can be formulated as a problem in NP.

We will reduce the whole of NP to a single problem!

Key properties of NP

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Key properties of NP

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Remark 1: The definition of NP is asymmetric, and does **not** include problems where we can easily verify `No` answers but not `Yes` answers. For example, it is not clear that "Is the input a **prime** number?" is in NP.

For what we're doing now this is just a technical quirk, since we can always just rephrase the question. Later it will be very important!

Key properties of NP

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Remark 1: The definition of NP is asymmetric, and does **not** include problems where we can easily verify `No` answers but not `Yes` answers. For example, it is not clear that "Is the input a **prime** number?" is in NP.

For what we're doing now this is just a technical quirk, since we can always just rephrase the question. Later it will be very important!

Remark 2: We define **P** to be the class of all decision problems which have a polynomial-time algorithm. Then $\mathrm{P} \subseteq \mathrm{NP}$. Why?

Key properties of NP

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Remark 1: The definition of NP is asymmetric, and does **not** include problems where we can easily verify `No` answers but not `Yes` answers. For example, it is not clear that "Is the input a **prime** number?" is in NP.

For what we're doing now this is just a technical quirk, since we can always just rephrase the question. Later it will be very important!

Remark 2: We define **P** to be the class of all decision problems which have a polynomial-time algorithm. Then P $\subseteq$ NP. Why?

Because `Verify` can simply ignore w, solve x, and return the solution. (So "is the input a prime number?" actually is in NP.)

Recap of propositional logic

We write Boolean OR as $\vee$, Boolean AND as $\wedge$, and Boolean NOT as $\neg$.

Recap of propositional logic

We write Boolean OR as $\vee$, Boolean AND as $\wedge$, and Boolean NOT as $\neg$.

A **literal** is either a Boolean variable x or its negation $\neg x$.

An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$.

A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$. Any formula can be expressed in CNF.

Recap of propositional logic

We write Boolean OR as $\vee$, Boolean AND as $\wedge$, and Boolean NOT as $\neg$.

A **literal** is either a Boolean variable x or its negation $\neg x$.

An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$.

A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$. Any formula can be expressed in CNF.

An **assignment** for a formula is a map from its variables to the set $\{\mathtt{True}, \mathtt{False}\}$, and the formula's **truth value** under that assignment is calculated as you would expect. For example, under the assignment $x \mapsto \mathtt{True}$ and $y \mapsto \mathtt{False}$, the truth value of $x \wedge (\neg x \vee y)$ is

$$\begin{aligned} \mathtt{True} \wedge (\neg\mathtt{True} \vee \mathtt{False}) &= \mathtt{True} \wedge (\mathtt{False} \vee \mathtt{False}) \\ &= \mathtt{True} \wedge \mathtt{False} = \mathtt{False}. \end{aligned}$$

Recap of propositional logic

We write Boolean OR as $\vee$, Boolean AND as $\wedge$, and Boolean NOT as $\neg$.

A **literal** is either a Boolean variable x or its negation $\neg x$.

An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$.

A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$. Any formula can be expressed in CNF.

An **assignment** for a formula is a map from its variables to the set $\{\mathtt{True}, \mathtt{False}\}$, and the formula's **truth value** under that assignment is calculated as you would expect. For example, under the assignment $x \mapsto \mathtt{True}$ and $y \mapsto \mathtt{False}$, the truth value of $x \wedge (\neg x \vee y)$ is

$$\begin{aligned} \mathtt{True} \wedge (\neg \mathtt{True} \vee \mathtt{False}) &= \mathtt{True} \wedge (\mathtt{False} \vee \mathtt{False}) \\ &= \mathtt{True} \wedge \mathtt{False} = \mathtt{False}. \end{aligned}$$

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

SAT

A **literal** is either a variable x or its negation $\neg x$. An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$. A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

SAT

A **literal** is either a variable x or its negation $\neg x$. An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$. A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

The **SAT** problem asks: "Is the input CNF formula satisfiable?"

SAT

A **literal** is either a variable x or its negation $\neg x$. An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$. A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

The **SAT** problem asks: "Is the input CNF formula satisfiable?"

This is in NP, since we can quickly check whether a given assignment makes the formula true. Conversely...

SAT

A **literal** is either a variable x or its negation $\neg x$. An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$. A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

The **SAT** problem asks: "Is the input CNF formula satisfiable?"

This is in NP, since we can quickly check whether a given assignment makes the formula true. Conversely...

Cook-Levin Theorem: Every problem in NP is Cook-reducible to SAT.

SAT

A **literal** is either a variable x or its negation $\neg x$. An **OR clause** is an OR of distinct literals, e.g. $x \vee (\neg y) \vee z$. A formula in **conjunctive normal form (CNF)** is an AND of OR clauses, such as $x \wedge (y \vee z) \wedge (\neg x \vee \neg z)$.

We say a propositional formula is **satisfiable** if there's some assignment (a **satisfying assignment**) that makes it true, and **unsatisfiable** otherwise.

The **SAT** problem asks: "Is the input CNF formula satisfiable?"

This is in NP, since we can quickly check whether a given assignment makes the formula true. Conversely...

Cook-Levin Theorem: Every problem in NP is Cook-reducible to SAT.

So if there's a polynomial algorithm for SAT, then there's a polynomial algorithm for **every** problem in NP — that is, P = NP!

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it! This is normally how we prove NP-hardness.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it! This is normally how we prove NP-hardness.

The Million Dollar Conjecture: $\mathrm{P} \neq \mathrm{NP}$.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it! This is normally how we prove NP-hardness.

The Million Dollar Conjecture: $\mathrm{P} \neq \mathrm{NP}$.

If this is true, then **no** NP-**hard** problem has a polynomial-time algorithm.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm `Verify` such that if x is a `Yes` instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it! This is normally how we prove NP-hardness.

The Million Dollar Conjecture: $\mathrm{P} \neq \mathrm{NP}$.

If this is true, then **no** NP-**hard** problem has a polynomial-time algorithm. If it is false, then **every** NP-**complete** problem has a poly-time algorithm.

NP-completeness

P is the class of all decision problems with a polynomial-time algorithm.

NP is the class of all decision problems X with a polynomial-time algorithm $\texttt{Verify}$ such that if x is a $\texttt{Yes}$ instance of X, then there is some bit string w (a **witness**) with $\texttt{Verify}(x, w) = \texttt{Yes}$.

Cook-Levin Theorem: Any problem in NP is Cook-Reducible to SAT.

We say a problem is **NP-hard*** if any problem in NP is Cook-reducible to it, and **NP-complete*** if it is also in NP. So SAT is NP-complete.

* There are two definitions of these terms; I'll cover the other one next week.

Important: By the Cook-Levin Theorem, a problem is NP-hard if and only if SAT reduces to it! This is normally how we prove NP-hardness.

The Million Dollar Conjecture: $\mathrm{P} \neq \mathrm{NP}$.

If this is true, then **no** NP-**hard** problem has a polynomial-time algorithm. If it is false, then **every** NP-**complete** problem has a poly-time algorithm.

A proof either way is worth $1,000,000 from the Clay Foundation...

What does this mean in practice?

- If a problem is NP-hard, there's probably no poly-time algorithm for it.

What does this mean in practice?

- If a problem is NP-hard, there's probably no poly-time algorithm for it.
- Even if there is a poly-time algorithm, you won't be able to find it.
    - No, really. Please don't try. I get too many crank emails already.

What does this mean in practice?

- If a problem is NP-hard, there's probably no poly-time algorithm for it.
- Even if there is a poly-time algorithm, you won't be able to find it.
    - No, really. Please don't try. I get too many crank emails already.
- In practice, almost every problem either has a poly-time algorithm or is NP-hard.
    - Technically, if P $\neq$ NP then there are infinitely many complexity classes between them, but they're really weird and artificial...

What does this mean in practice?

- If a problem is NP-hard, there's probably no poly-time algorithm for it.
- Even if there is a poly-time algorithm, you won't be able to find it.
  - No, really. Please don't try. I get too many crank emails already.
- In practice, almost every problem either has a poly-time algorithm or is NP-hard.
  - Technically, if P $\neq$ NP then there are infinitely many complexity classes between them, but they're really weird and artificial...

So proving your problem is NP-hard means it's a **dead end** — you won't be able to solve it, so you need to find an alternative. (More next week...)

What does this mean in practice?

- If a problem is NP-hard, there's probably no poly-time algorithm for it.
- Even if there is a poly-time algorithm, you won't be able to find it.
  - No, really. Please don't try. I get too many crank emails already.
- In practice, almost every problem either has a poly-time algorithm or is NP-hard.
  - Technically, if $\mathrm{P} \neq \mathrm{NP}$ then there are infinitely many complexity classes between them, but they're really weird and artificial...

So proving your problem is NP-hard means it's a **dead end** — you won't be able to solve it, so you need to find an alternative. (More next week...)

The good news is: this means you spent a couple of hours writing a hardness proof rather than weeks or months failing to write an algorithm!

NP-hardness can also be a good way of ruling out approaches: "If this worked for problem X, then it would also work for [insert NP-hard problem here], so it's not going to work."

What if P = NP?

If P = NP, then we're very unlikely to get a practical algorithm.
There will be shenanigans. Possibly even tomfoolery.

What if P = NP?

If P = NP, then we're very unlikely to get a practical algorithm. There will be shenanigans. Possibly even tomfoolery.

But it would still be fascinating from a theory viewpoint. Why? Because it would say: as long as we can recognise a solution when we see it, we can **always** do better than naïve search. That's really counterintuitive!

What if P = NP?

If P = NP, then we're very unlikely to get a practical algorithm. There will be shenanigans. Possibly even tomfoolery.

But it would still be fascinating from a theory viewpoint. Why? Because it would say: as long as we can recognise a solution when we see it, we can **always** do better than naïve search. That's really counterintuitive!

For example, this problem is NP-complete: "Given a mathematical statement S and $k \geq 1$, is there a proof of S in k lines or fewer?"

If P = NP, so there's a polynomial-time algorithm for that problem, then what does that mean for the way we think about mathematics?

What if P = NP?

If P = NP, then we're very unlikely to get a practical algorithm. There will be shenanigans. Possibly even tomfoolery.

But it would still be fascinating from a theory viewpoint. Why? Because it would say: as long as we can recognise a solution when we see it, we can **always** do better than naïve search. That's really counterintuitive!

For example, this problem is NP-complete: "Given a mathematical statement S and $k \geq 1$, is there a proof of S in k lines or fewer?"

If P = NP, so there's a polynomial-time algorithm for that problem, then what does that mean for the way we think about mathematics?