NP-completeness

Slides by Carl Kingsford

The class **P**

P is the set of decision problems whose memberships are decidable by a Turing Machine that makes a polynomial number of steps.

By the Church-Turing thesis, this is the "same" as:

P is the set of decision problems that can be decided by a computer in a polynomial time.

You can just think of your normal computer as a Turing Machine — and we won't worry too much about that formalism.

Efficient Certification

Def. An algorithm V is an efficient certifier for decision problem X if:

1. V is a polynomial time algorithm that takes two input strings I (instance of X) and C (a certificate).
2. V outputs either `yes` or `no`.
3. There is a polynomial $p(n)$ such that for every string I:

   $I \in X$ if and only if there exists string C of length $\leq p(|I|)$ such that $V(I, C) =$ `yes`.

V is an algorithm that can decide whether an instance I is a `yes` instance if it is given some "help" in the form of a polynomially long certificate.

The class NP

NP is the set of languages for which there exists an efficient certifier.

The class NP

NP is the set of languages for which there exists an efficient certifier.

P is the set of languages for which there exists an efficient certifier that ignores the certificate.

A problem is in **P** if we can decided it in polynomial time. It is in **NP** if we can decide it in polynomial time, if we are given the right certificate.

$\mathbf{P} \subseteq \mathbf{NP}$

Theorem. $\mathbf{P} \subseteq \mathbf{NP}$

Proof. Suppose $X \in \mathbf{P}$. Then there is a polynomial-time algorithm A for X.

To show that $X \in \mathbf{NP}$, we need to design an efficient certifier $B(I, C)$.

Just take $B(I, C) = A(I)$. $\square$

Every problem with a polynomial time algorithm is in **NP**.

$\mathbf{P} \neq \mathbf{NP}$?

The big question:

$\mathbf{P} = \mathbf{NP}$?

We know $\mathbf{P} \subseteq \mathbf{NP}$. So the question is:

Is there some problem in **NP** that is not in **P**?

Seems like the power of the certificate would help a lot.
But no one knows. . . .

Reductions

Def. Problem X is polynomial-time reducible to problem Y if

- there is a polynomial-time algorithm A
- that converts instances of X into instances of Y such that
- for all I:

$$A(I) = \texttt{yes} \iff I = \texttt{yes}$$

We denote this by $X \leq_P Y$.

Reductions for Hardness

Theorem. *If $Y \leq_P X$ and Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.*

Why? If we *could* solve X in polynomial time, then we'd be able to solve Y in polynomial time using the reduction, contradicting the assumption.

So: If we could find one hard problem Y, we could prove that another problem X is hard by reducing Y to X.

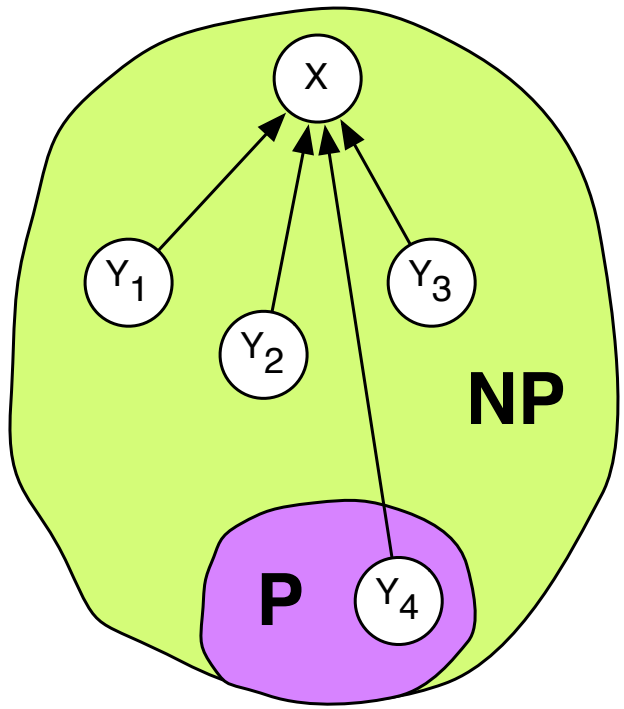
NP-completeness

Def. We say X is NP-complete if:

- $X \in \mathbf{NP}$
- for all $Y \in \mathbf{NP}$, $Y \leq_P X$.

If these hold, then X can be used to solve every problem in **NP**.

Therefore, X is definitely at least as hard as every problem in **NP**.



NP-completeness and P=NP

Theorem. *If X is NP-complete, then X is solvable in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof. If $\mathbf{P} = \mathbf{NP}$, then X can be solved in polytime.

Suppose X is solvable in polytime, and let Y be any problem in **NP**. We can solve Y in polynomial time: reduce it to X.

Therefore, every problem in **NP** has a polytime algorithm and $\mathbf{P} = \mathbf{NP}$.

Reductions and NP-completeness

Theorem. *If Y is NP-complete, and*

1. *X is in NP*
2. *$Y \leq_P X$*

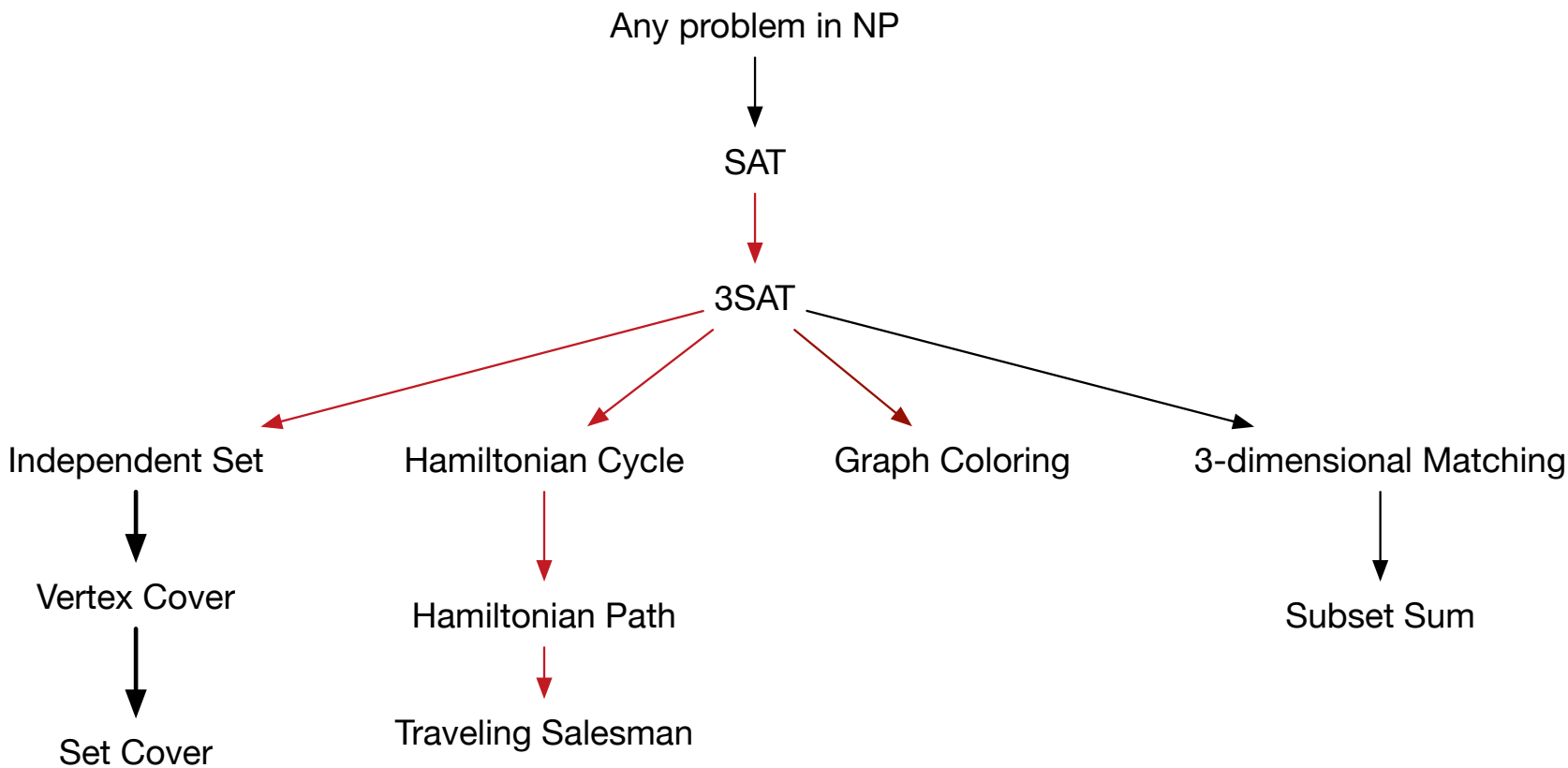
then X is NP-complete.

In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

Proof. Let Z be any problem in **NP**. Since Y is NP-complete, $Z \leq_P Y$. By assumption, $Y \leq_P X$. Therefore: $Z \leq_P Y \leq_P X$. $\square$

Chain of Reductions

Cook-Levin Theorem: The problem SAT is NP-complete.



Boolean Formulas

Boolean Formulas:

Variables: x_1, x_2, x_3 (can be either **true** or **false**)

Terms: $t_1, t_2, \ldots, t_\ell$: t_j is either x_i or $\bar{x}_i$
(meaning either x_i or **not** x_i).

Clauses: $t_1 \vee t_2 \vee \cdots \vee t_\ell$ ($\vee$ stands for "OR")
A clause is **true** if any term in it is **true**.

Example 1: $(x_1 \vee \bar{x_2}), (\bar{x_1} \vee \bar{x_3}), (x_2 \vee \bar{v_3})$

Example 2: $(x_1 \vee x_2 \vee \bar{x_3}), (\bar{x_2} \vee x_1)$

Boolean Formulas

Def. A truth assignment is a choice of **true** or **false** for each variable, ie, a function $v : X \to \{\textbf{true}, \textbf{false}\}$.

Def. A CNF formula is a conjunction of clauses:

$$C_1 \wedge C_2, \wedge \cdots \wedge C_k$$

Example: $(x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_2 \vee \bar{v_3})$

Def. A truth assignment is a satisfying assignment for such a formula if it makes every clause **true**.

SAT and 3-SAT

Problem (Satisfiability (SAT)). *Given a set of clauses $C_1, \ldots, C_k$ over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?*

Problem (Satisfiability (3-SAT)). *Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?*

Cook-Levin Theorem

Theorem (Cook-Levin). *SAT is NP-complete.*

Proven in early 1970s by Cook. Slightly different proof by Levin independently.

Idea of the original proof: encode the workings of a Nondeterministic Turing machine for an instance I of problem $X \in \mathbf{NP}$ as a SAT formula so that the formula is satisfiable if and only if the nondeterministic Turing machine would accept instance I.

Another intuition why this is true: A computer is just a circuit, and SAT encodes a kind circuit.

Reducing 3-SAT to Independent Set

Thm. 3-SAT $\leq_P$ Independent Set

Proof. Suppose we have an algorithm to solve Independent Set, how can we use it to solve 3-SAT?

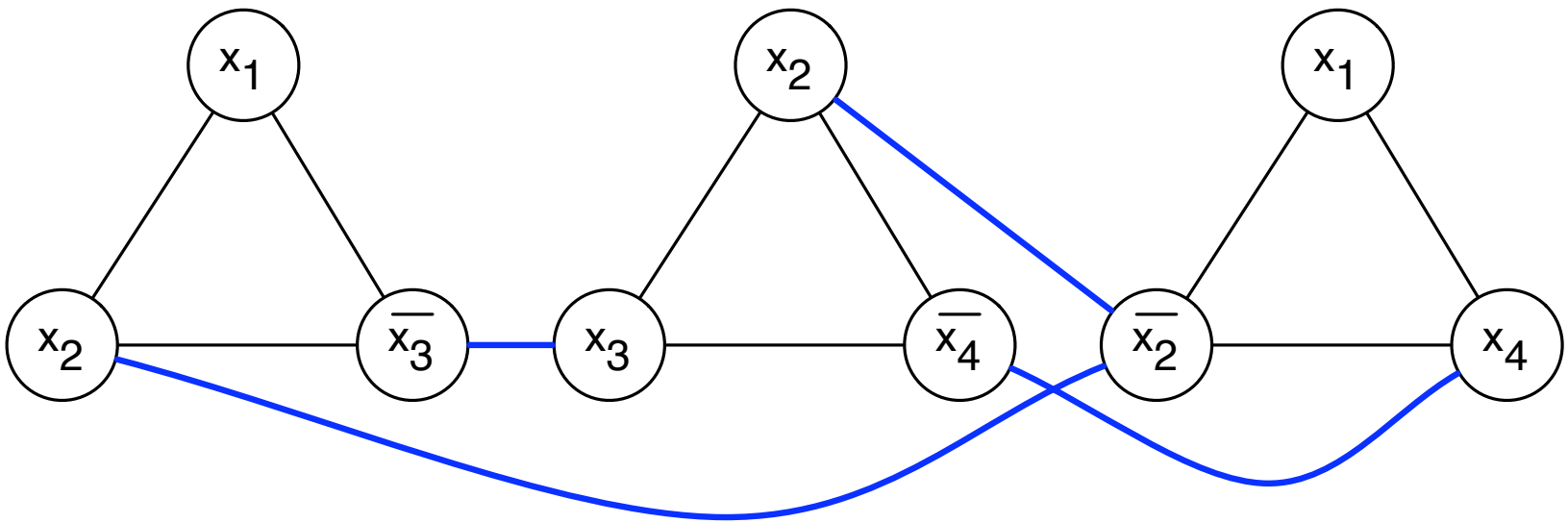
To solve 3-SAT:

- you have to choose a term from each clause to set to **true**,
- but you can't set both x_i and $\bar{x}_i$ to **true**.

How do we do the reduction?

3-SAT $\leq_P$ Independent Set

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



Proof

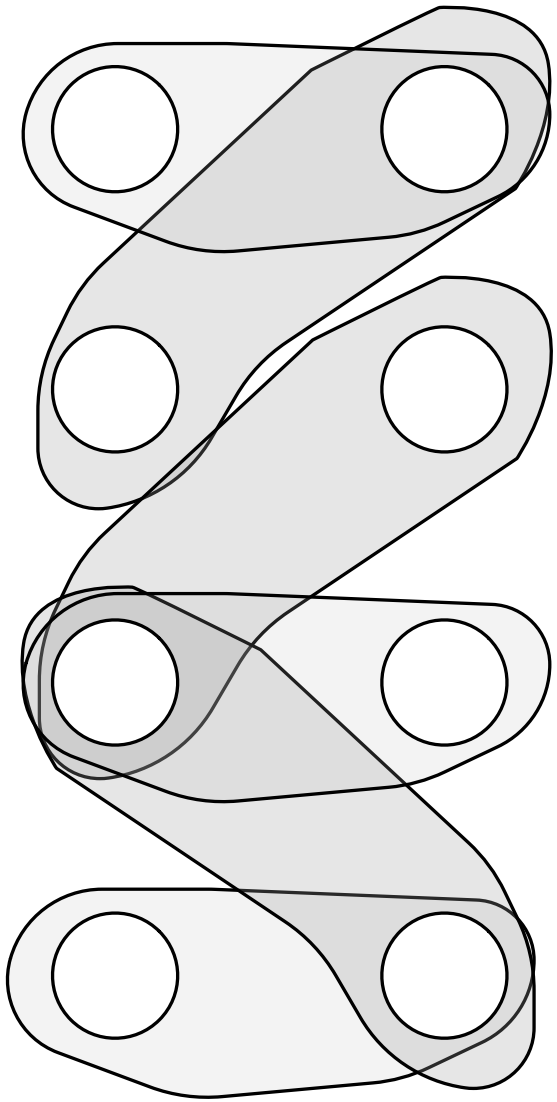
Theorem. *This graph has an independent set of size k iff the formula is satisfiable.*

Proof. $\Longrightarrow$ If the formula is satisfiable, there is at least one true literal in each clause. Let S be a set of one such true literal from each clause. $|S| = k$ and no two nodes in S are connected by an edge.

$\Longrightarrow$ If the graph has an independent set S of size k, we know that it has one node from each "clause triangle." Set those terms to **true**. This is possible because no 2 are negations of each other. $\square$

3-Dimensional Matching is NP-complete

Two-Dimensional Matching



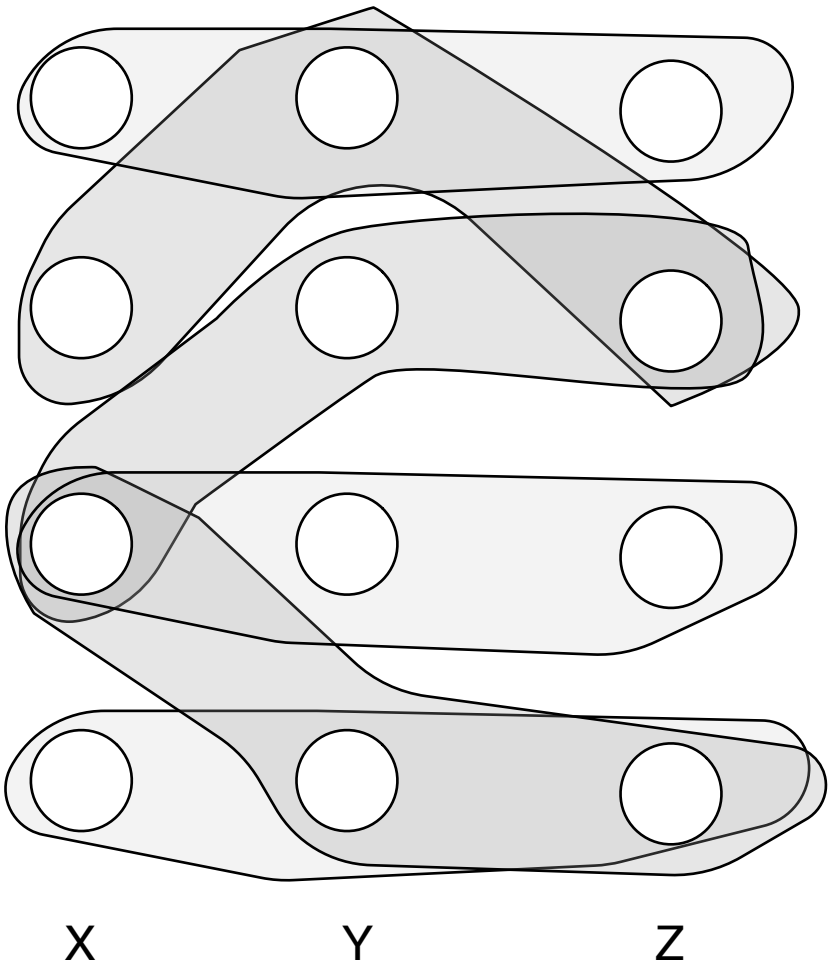
Recall '2-d matching':

Given sets X and Y, each with n elements, and a set E of pairs $\{x, y\}$,

Question: is there a choice of pairs such that every element in $X \cup Y$ is paired with some other element?

Usually, we thought of edges instead of pairs: $\{x, y\}$, but they are really the same thing.

Three-Dimensional Matching



Given: Sets X, Y, Z, each of size n, and a set $T \subset X \times Y \times Z$ of order triplets.

Question: is there a set of n triplets in T such that each element is contained in exactly one triplet?

3DM Is NP-Complete

Theorem. *Three-dimensional matching (aka 3DM) is NP-complete*

Proof. 3DM is in NP: a collection of n sets that cover every element exactly once is a certificate that can be checked in polynomial time.

Reduction from 3-SAT. We show that:

$$\text{3-SAT} \leq_P \text{3DM}$$

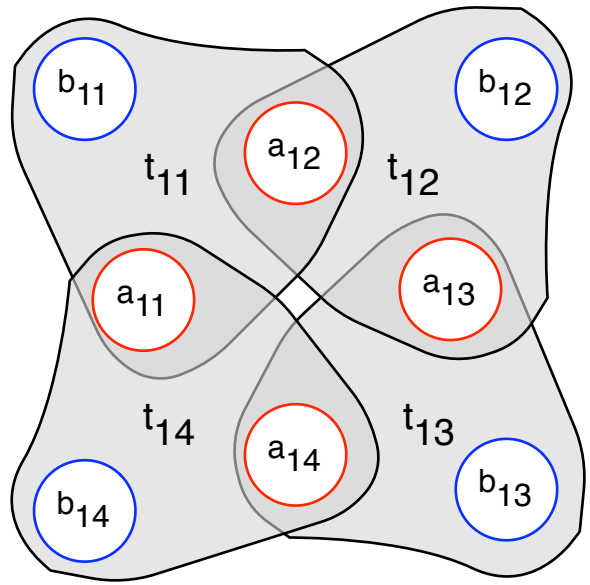
In other words, if we could solve 3DM, we could solve 3-SAT.

3-SAT $\leq_P$ 3DM

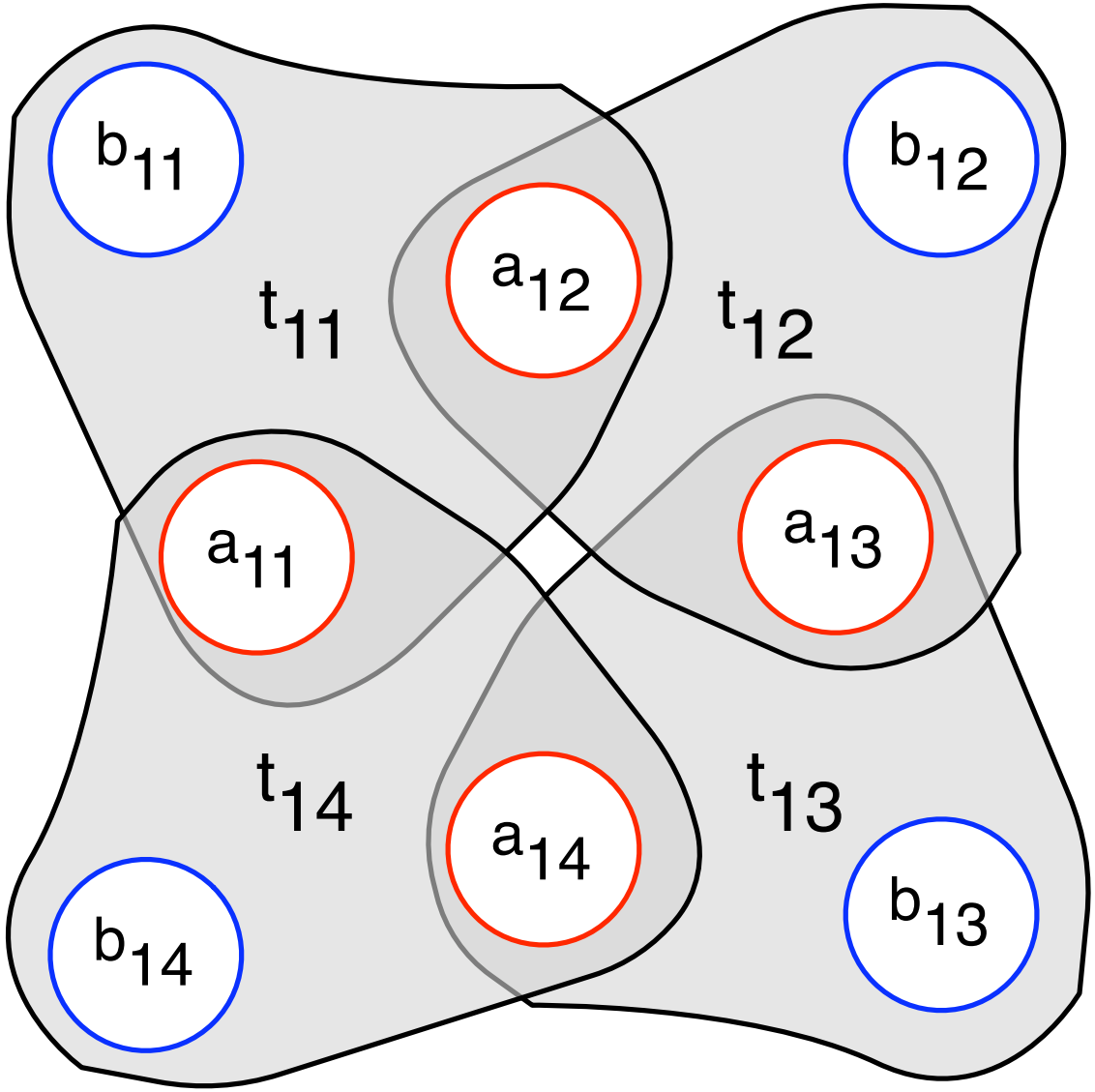
3SAT instance: $x_1, \ldots, x_n$ be n boolean variables, and $C_1, \ldots, C_k$ clauses.

We create a gadget for each variable x_i:

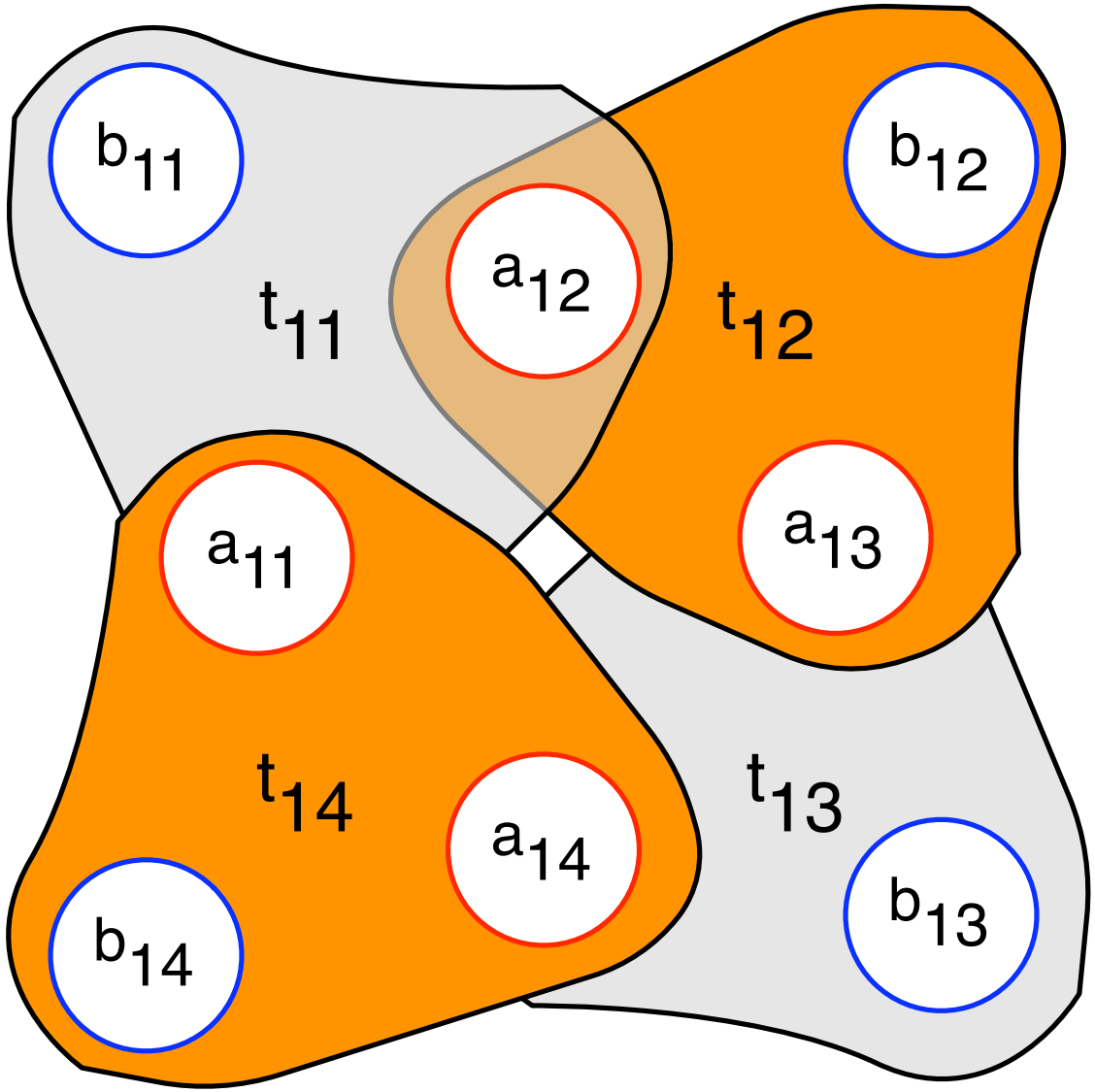
$$A_i = \{a_{i1}, \ldots, a_{i,2k}\} \quad \textit{core}$$
$$B_i = \{a_{i1}, \ldots, a_{i,2k}\} \quad \textit{tips}$$
$$t_{ij} = (a_{ij}, a_{i,j+1}, b_{ij}) \quad \textit{TF triples}$$



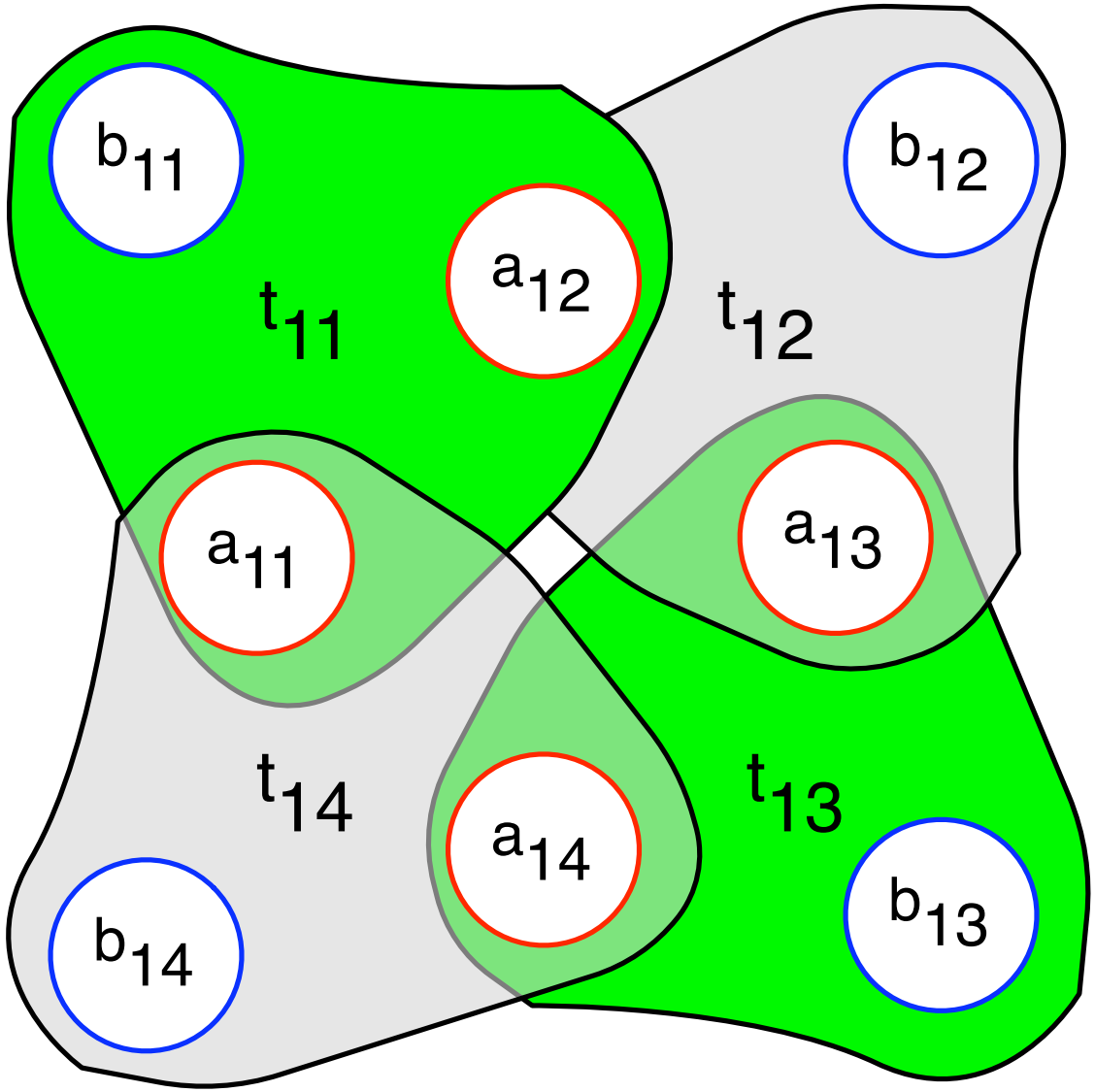
Gadget Encodes True and False



Gadget Encodes True and False

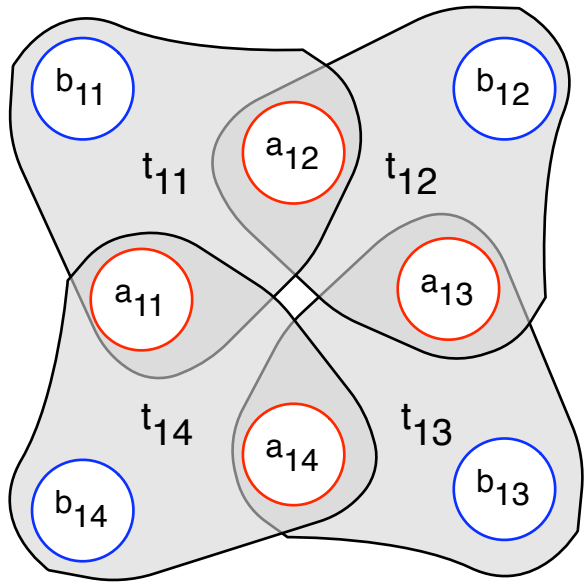


Gadget Encodes True and False



How "choice" is encoded

- We can only either use the even or odd "wings" of the gadget.
- In other words, if we use the even wings, we leave the odd tips uncovered (and vice versa).
- Leaving the odd tips free for gadget i means setting x_i to **false**.
- Leaving the odd tips free for gadget i means setting x_i to **true**.



Clause Gadgets

Need to encode constraints between the tips that ensure we satisfy all the clauses.

We create a gadget for each clause $C_j = \{t_1, t_2, t_3\}$

$$P_j = \{c_j, c'_j\} \quad \textit{Clause core}$$

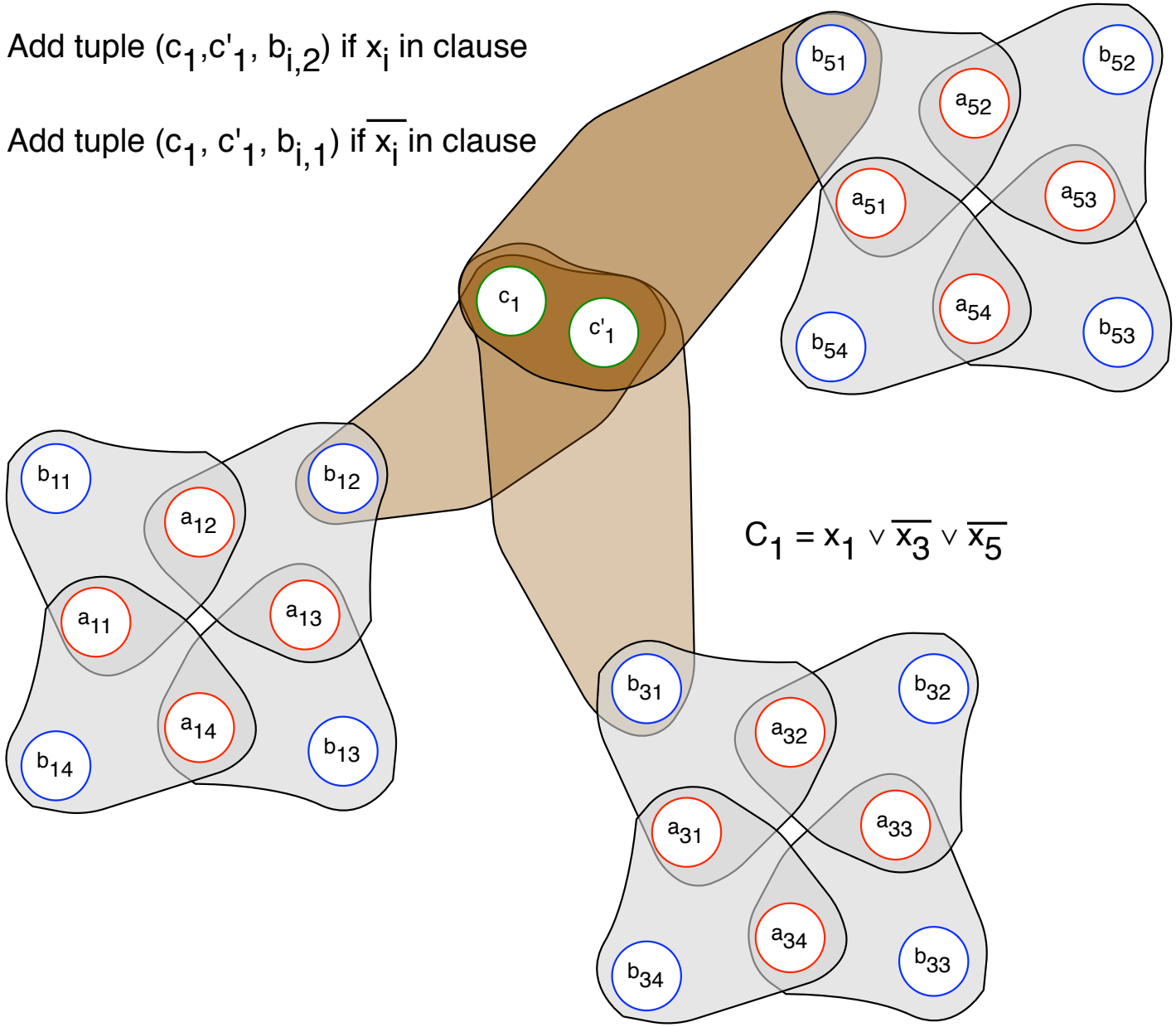
We will hook up these two clause core nodes with some tip nodes depending on whether the clause asks for a variable to be true or false.

See the next slide.

Clause Gadget Hookup

Add tuple $(c_1, c'_1, b_{i,2})$ if x_i in clause

Add tuple $(c_1, c'_1, b_{i,1})$ if $\overline{x_i}$ in clause



$C_1 = x_1 \vee \overline{x_3} \vee \overline{x_5}$

Clause Gadgets

Since only clause tuples (brown) cover c_j, c'_j, we have to choose exactly one of them for every clause.

We can only choose a clause tuple (c_j, c'_j, b_{ij}) if we haven't chosen a TF tuple that already covers b_{ij}.

Hence, we can satisfy (cover) the clause (c_j, c'_j) with the term represented by b_{ij} only if we "set" x_i to the appropriate value.

That's the basic idea. Two technical points left...

Details

Need to cover all the tips:

Even if we satisfy all the clauses, we might have extra tips left over. We add a clean up gadget (q_i, q_i', b) for every tip b.

Can we partition the sets?

$$X = \{a_{ij} : j \text{ even}\} \cup \{c_j\} \cup \{q_i\}$$
$$Y = \{a_{ij} : j \text{ odd}\} \cup \{c_j'\} \cup \{q_i'\}$$
$$Z = \{b_{ij}\}$$

Every set we defined uses 1 element from each of X, Y, Z.

Proof

If there is a satisfying assignment,

We choose the odd / even wings depending on whether we set a variable to **true** or **false**. At least 1 free tip for a term will be available to use to cover each clause gadget. We then use the clean up gadgets to cover all the rest of the tips.

If there is a 3D matching,

We can set variable x_i to **true** or **false** depending on whether it's even or odd wings were chosen. Because $\{c_j, c'_j\}$ were covered, we must have correctly chosen one even/odd wing that will satisfy this clause.