Algorithms

NP-Complete Problems

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

Search Problems

We are searching for a solution (path, tree, matching, etc.) among an exponential population of possibilities:

- n persons and n jobs can be matching in $n!$ different ways.
- a graph with n nodes has n^{n-2} spanning trees.
- a typical undirected graph has exponential number of paths from s to t.

Can be solved in exponential time by checking all candidates one by one.
Too much time to be useful in practice.

We have seen polynomial-time algorithms for these problems.

- Greedy algorithms - $O(|E| \log |V|)$ time for minimum spanning trees.
- Dijkstra's algorithm - $O((|V| + |E|) \log |V|)$ time for $s - t$ shortest path.
- Dynamic programming - $O(n^3)$ time for chain matrix multiplication.
- Linear programming - $O(n^3)$ time for bipartite matching.

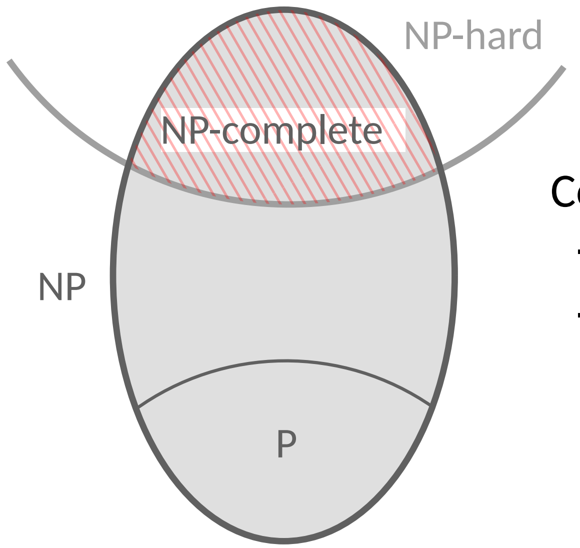
But there are search problems for which the fastest algorithms are all EXPONENTIAL!
- not substantially better than an exhaustive search.

P and NP

P stands for “polynomial” while NP stands for “nondeterministic polynomial time”.

> A solution to any search problem can be found and verified in polynomial time by a nondeterministic algorithm.

- A nondeterministic algorithm may exhibit different behaviors on different runs even for the same input.
- Nondeterministic poly-time means an algorithm may run in poly-time or exponential time depending on the choices it makes during execution.



Complexity classes include

- NP-complete : NP but $O(n^k)$-time algorithm unlikely.
- Undecidability : No algorithm that always gives the correct answer. Ex.The halting problem (determining whether a Turing machine halts for an arbitrary program and a finite input).

(1) Satisfiability (SAT)

An instance of SAT is a Boolean formula in CNF (conjunctive normal form):

$$(x \vee y \vee z)(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

A collection of *clauses* (the parentheses), each consisting of the disjuction of several *literals* (a Boolean variable or the negation of one.).

A satisfying truth assignment is an assignment of `true` or `false` to each variable so that every clause contains a literal whose value is `true`.

SAT problem. Given a Boolean formula in CNF, either find a satisfying true assignment or report “none exists”.

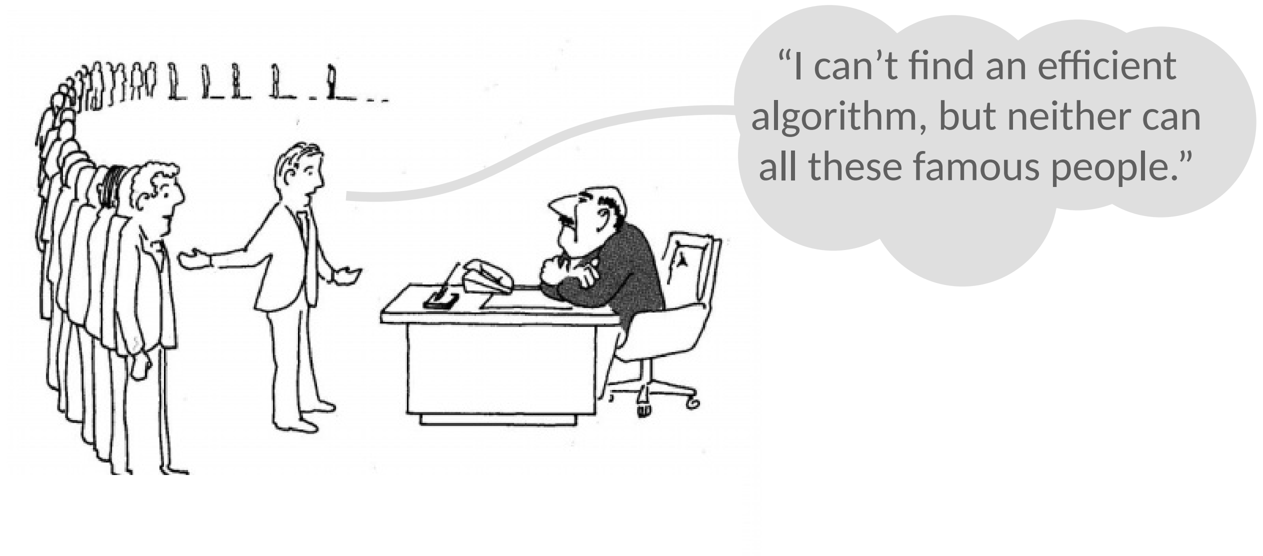
An exhaustive search on n variables - 2^n possible assignments!

SAT is a typical **search problem**. We are given an *instance* I, and we are asked to find a *solution* S.

Property of a search problem. Any proposed solution S to an instance I can be quickly checked for correctness. That is, there is a polynomial-time algorithm that takes as input I and S, and decides whether or not S is a solution for I.

(1) Satisfiability (SAT)

Researchers over the past 50 years have tried to find efficient ways to solve it, but without success. The fastest algorithms we have are still exponential on their worst-case inputs.


"I can't find an efficient algorithm, but neither can all these famous people."

But, interestingly, there are two natural variants of SAT for which we have good algorithms.

- **Horn formula.** All clauses contain at most one positive literal. A greedy algorithm can find a solution in linear time.
- **2SAT.** All clauses have only two literals. Can be solved in linear time by finding the strongly connected components.

(2) Traveling Salesman Problem (TSP)

An instance of TSP consists of n vertices and all $n(n-1)/2$ distances between them.

TSP problem. Given a set of n vertices and their pairwise distances, find a shortest tour (a permutation of vertices) that passes through every vertex exactly once.

TSP as a Search problem. Given a set of n vertices, their pairwise distances, and a budget b, find a tour of total cost b or less that passes through every vertex exactly once, or report "no such tour exists".

An optimization problem and a search problem of TSP reduce each other.

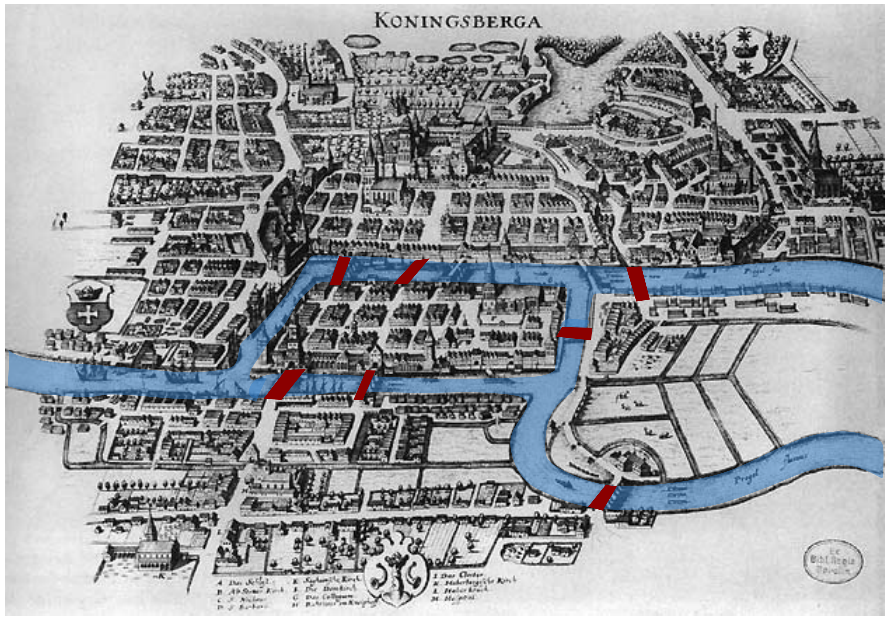
- Any algorithm for the optimization can also answer the search problem.
- We can answer the optimization problem by using binary search (on budget b) with an algorithm for the search problem.

For the search problem, any proposed solution (tour T) to an instance I can be quickly checked for correctness: "Is T a tour?" and "Does T has total length $\leqslant b$?"

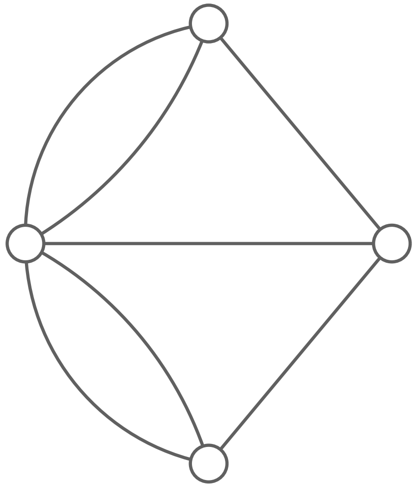
A dynamic programming algorithm takes $O(n^2 2^n)$ time, which is slightly better than $(n-1)!$. But no polynomial time algorithm is known yet.

(3) Rudrata (Hamilton) Cycle Problem

Euler Path problem. Given a graph, find a path that contains each edge exactly once.



Merian-Erben - 1652



When can a graph be drawn without lifting the pencil from the paper?
- Leonhard Euler, the famous Swiss mathematician, Summer of 1735.

Euler's intuition - If and only if (a) the graph is connected and (b) every vertex, without the possible exception of two vertices (start and end), has even degree.

It follows from Euler's observation, and a little more thinking, that this search problem can be solved in polynomial time.

Eulerian path

Theorem [Carl Hierholzer 1873] A connected undirected graph has an Eulerian cycle iff every vertex has even degree.

Proof. ($\rightarrow$) Each time the path passes through a vertex it contributes two to the vertex's degree, except the starting and ending vertices. If the path terminates where it started, it will contribute two to that degree as well.

($\leftarrow$) We start with any vertex and choose a next vertex from all adjacent vertices in random order. We continue in this manner. After finite number of steps, we will end in the vertex we started with. If we traverse all edges, we build an Euler cycle. Otherwise, consider a subgraph H obtained from the original graph by deleting that cycle. Each vertex of H has even degree and the original graph and H must have one vertex in common. We start with this vertex and repeat the procedure until all edges are traversed.

The proof above produces an algorithm immediately. Do you see? What is the running time?

Eulerian path

Theorem. A connected undirected graph has an Eulerian trail (or Eulerian path) iff at most two vertices have odd degree.

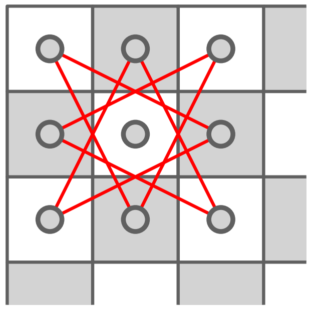
Fleury's algorithm (1883) starts at a vertex of odd degree. At each step it chooses the next edge to be one whose deletion would not disconnect the graph, unless there is no such edge, in which case it picks the remaining edge at the current vertex. At the end of the algorithm there are no edges left, and the sequence in which the edges was chosen forms an Eulerian trail if there are at most two vertices of odd degree.

It takes $O(|E|)$ iterations. In each iteration, we perform an operation of detecting bridges, which can be done in linear time (Tarjan), so the overall time complexity of $O(|E|^2)$. A dynamic bridge-finding algorithm of Thorup (2000) improved this to $O(|E| \log^3 |E| \log \log |E|)$ time.

However, an Eulerian path can be found in linear time using the algorithm for Eulerian cycle.

Theorem. A digraph has an Eulerian cycle iff it is strongly connected and $\text{indeg}(v_i) = \text{outdeg}(v_i)$ for all vertices.

(3) Rudrata (Hamilton) Cycle Problem



knight's tour

Can one visit all the sqaures of the chessboard, without repeating any square, in one long walk that ends at the starting square and at each step makes a legal knight move?
- Rudrata, a Kashmiri poet, Summer of 1735.

Rudrata Cycle problem. Given a graph, find a cycle that passes through every vertex exactly once.

There are two differences from the Euler problem.

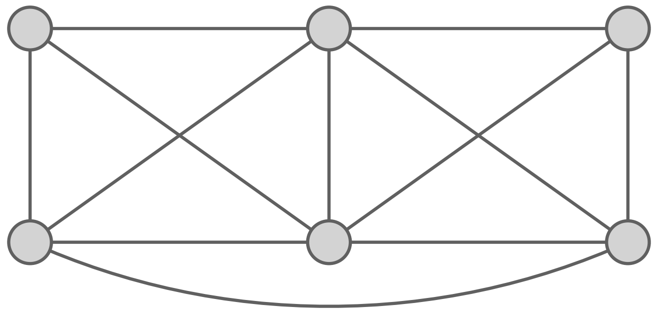
- Euler's problem visits all edges while Rudrata's visits all vertices.
- Euler's demands a path while Rudrata's requires a cycle.

This is a reminiscent of the TSP, and indeed no polynomial time algorithms is known for it. But a proposed solution can be quickly checked.

(4) Balanced Cut

A *cut* is a set of edges whose removal leaves a graph disconnected.

Minimum Cut problem. Given a graph and a budget b, find a cut with at most b edges.



Easy to solve: assign capacity 1 for each edge, run **max-flow** algorithm $n - 1$ times (from a fixed node to every single other node). This gives us the smallest such flow, from which we can get the smallest cut.

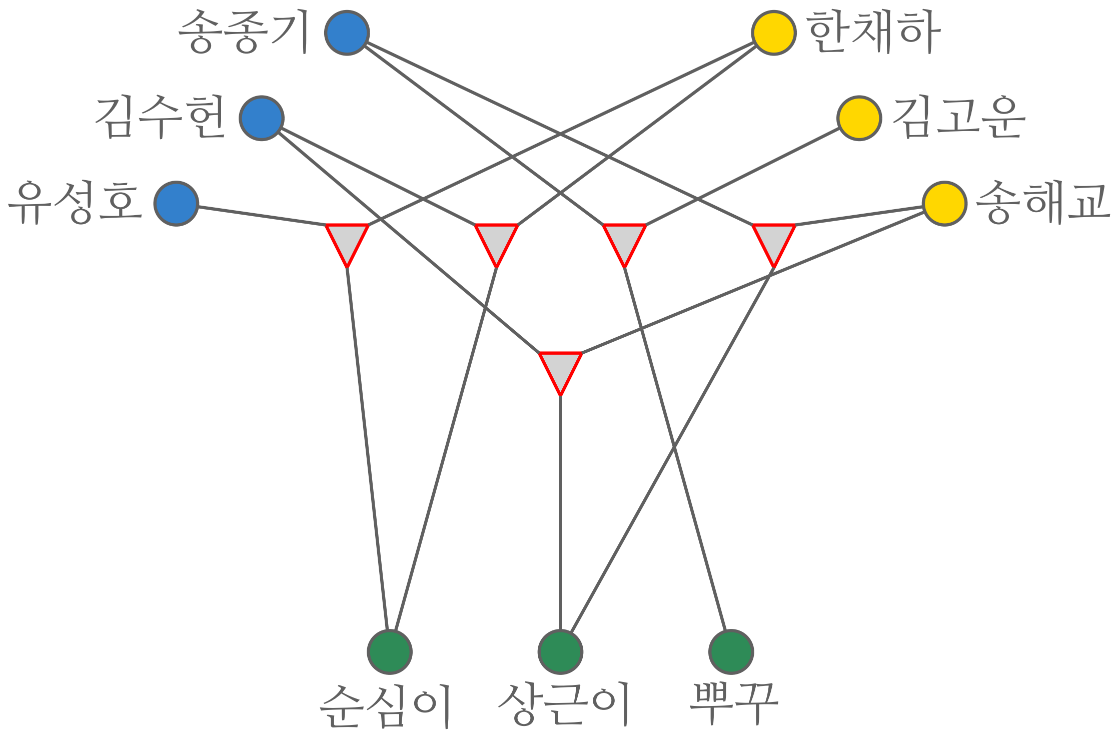
Balanced Cut problem. Given a graph with n vertices and a budget b, find a cut (A, B) such that $|A|, |B| \geqslant n/3$ and at most b edges in the cut.

Balanced cuts arise in a variety of important applications, such as *clustering* and and *image segmentation*. However, no polynomial-time algorithm is known for it. But a proposed solution can be quickly checked.

(5) 3D Matching

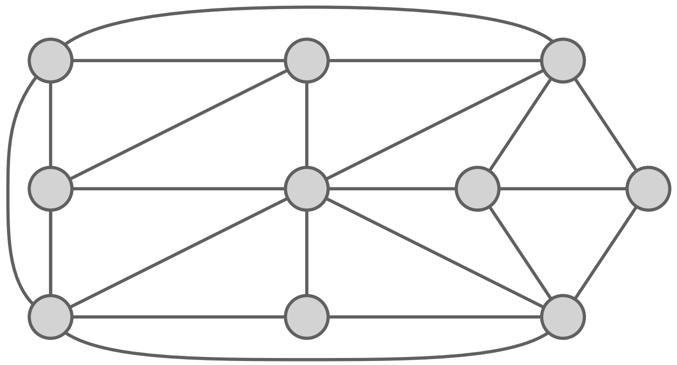
Bipartite Matching problem. Given n boys and n girls, and the compatibility between them, find as many disjoint couples as possible.

3D Matching problem. Given n boys and n girls but also n pets, and the compatibility among them, find n disjoint triples.



(6) Independent Set, Vertex Cover, Clique

Independent Set problem. Given a graph and a goal g, find g vertices that are independent, that is, no two of which have an edge between them.



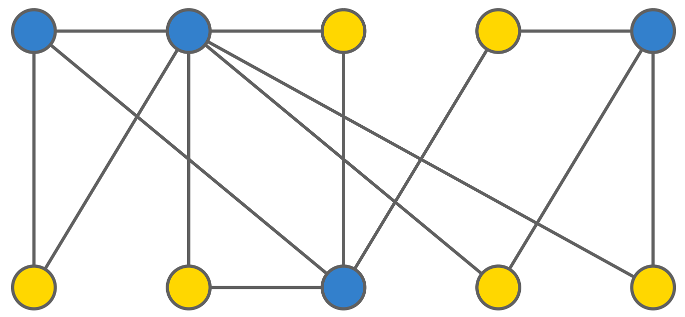
Vertex Cover problem. Given a graph and a budget b, find b vertices that cover (touch) every edge.

Vertex Cover is a special case of Set Cover, in which we are given a set E and several subsets of it, $S_1, \ldots, S_m$, along with a budget b, and we aim to select b of these subsets so that their union is E. Let E be the set of the edges of a graph, and S_i be the set of the edges adjacent to vertex v_i for each vertex v_i. 3D Matching is also a special case of Set Cover.

Clique problem. Given a graph with n vertices and a budget b, find a set of b vertices such that all possible edges between them are present.

(6) Independent Set, Vertex Cover, Clique

Claim. Vertex-Cover $\equiv_P$ Independent-Set.



🔵 vertex cover

🟡 independent set

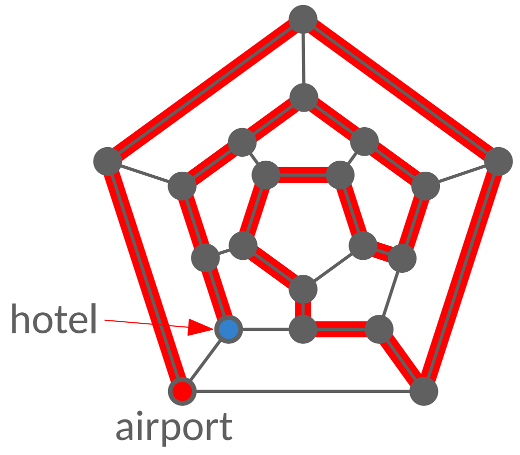
Proof. We show S is an independent set iff $V \setminus S$ is a vertex cover.

Let S be any independent set. Consider an arbitrary edge (u, v). Since S is an independent set, we have $u \notin S$ or $v \notin S$. This implies that $u \in V \setminus S$ or $v \in V \setminus S$. Thus, $V \setminus S$ covers (u, v).

Let $V \setminus S$ be any vertex cover. Consider two nodes $u \in S$ and $v \in S$. Since $V \setminus S$ is a vertex cover, the graph does not have edge (u, v). Thus, no two nodes in S are joined by an edge, which implies that S is an independent set.

(7) Longest Path, Subset Sum

Longest Path (aka Taxicab Rip-off) problem. Given a graph with nonnegative edge weights and two vertices *s* and *t*, along with a goal *g*, find a simple path from *s* to *t* with total weight at least *g*.



Subset Sum problem. Given a set of integers and a value *W*, find a subset of integers that adds up to exactly *W*.

This is a special case of Knapsack problem, so it cannot be any harder. But it turns out that Subset Sum is also very hard.

NP-Complete Problems

The world is full of search problems, some of which can be solved efficiently, while others seem to be very difficult, as depicted in the following table.

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
TSP	MST
Longest Path	Shortest Path
3D Matching	Bipartite Matching
Independent Set, Vertex Cover, Clique	Independent Set on trees
Rudrata (Hamiltonian) Path	Euler Path
Balanced Cut	Minimum Cut

The problems on the left are all difficult for the same reason!

P and NP

Property of a search problem. Any proposed solution S to an instance I can be quickly checked for correctness.
There is an efficient checking algorithm $C(I, S)$ that takes time polynomial in $|I|$.

NP : the class of all search problems.

P : the class of all search problems that can be solved in polynomial time. Most of search problems we have considered belong to P.

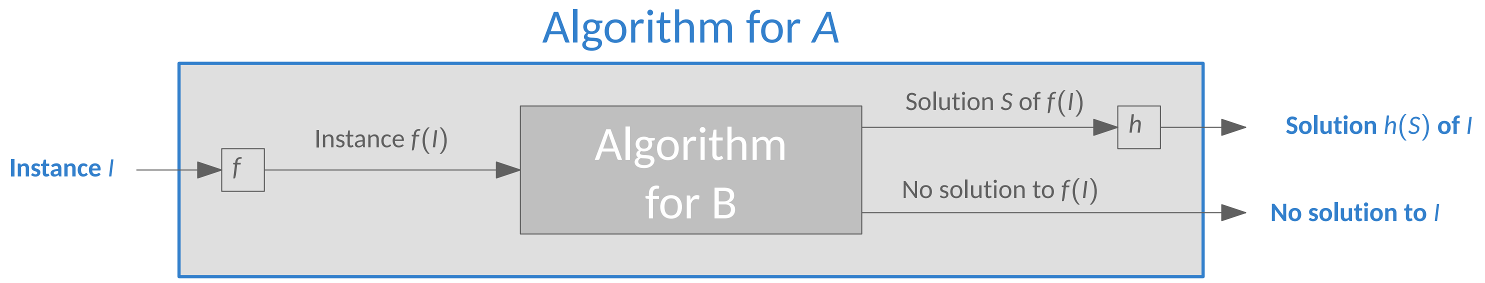
P $=$ NP ?
Most researchers believe P $\neq$ NP. But no proof. Then why they believe so?

There are hard problems, famously unsolved for decades and centuries. But on what evidence do we believe there's no efficient algorithms for them?

Reduction provides some evidence.

The hard problems in the left side of the table are the hardest search problems in NP, and all are the same problem (by reduction).

Reductions



Reduction, $A \rightarrow B$, has two purposes :

(1) We know how to solve B efficiently, and want to use this to solve A.

(2) We know A is hard, and use the reduction to prove B is hard as well.

Reduction provides some evidence.

The hard problems in the left side of the table are the hardest search problems in NP, and all are the same problem (by reduction).

The problems in the left side of the table reduce each other, and all other search problems reduce to them (we will see soon).

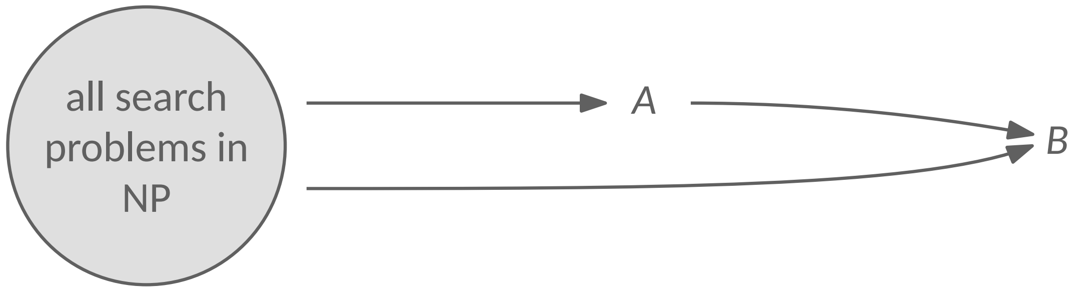
Reductions

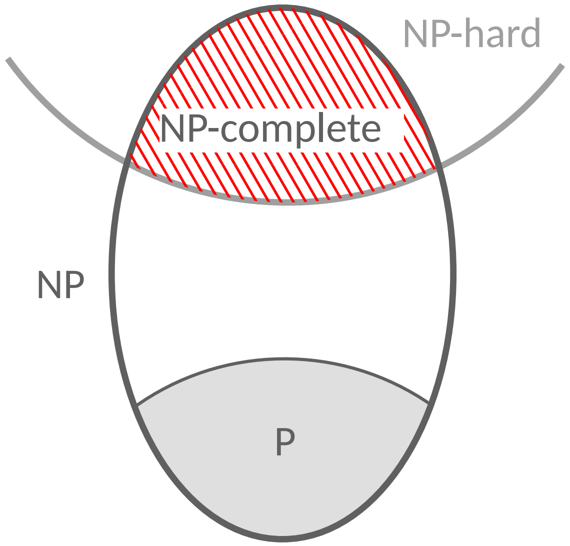
Reductions have the composition property:

If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

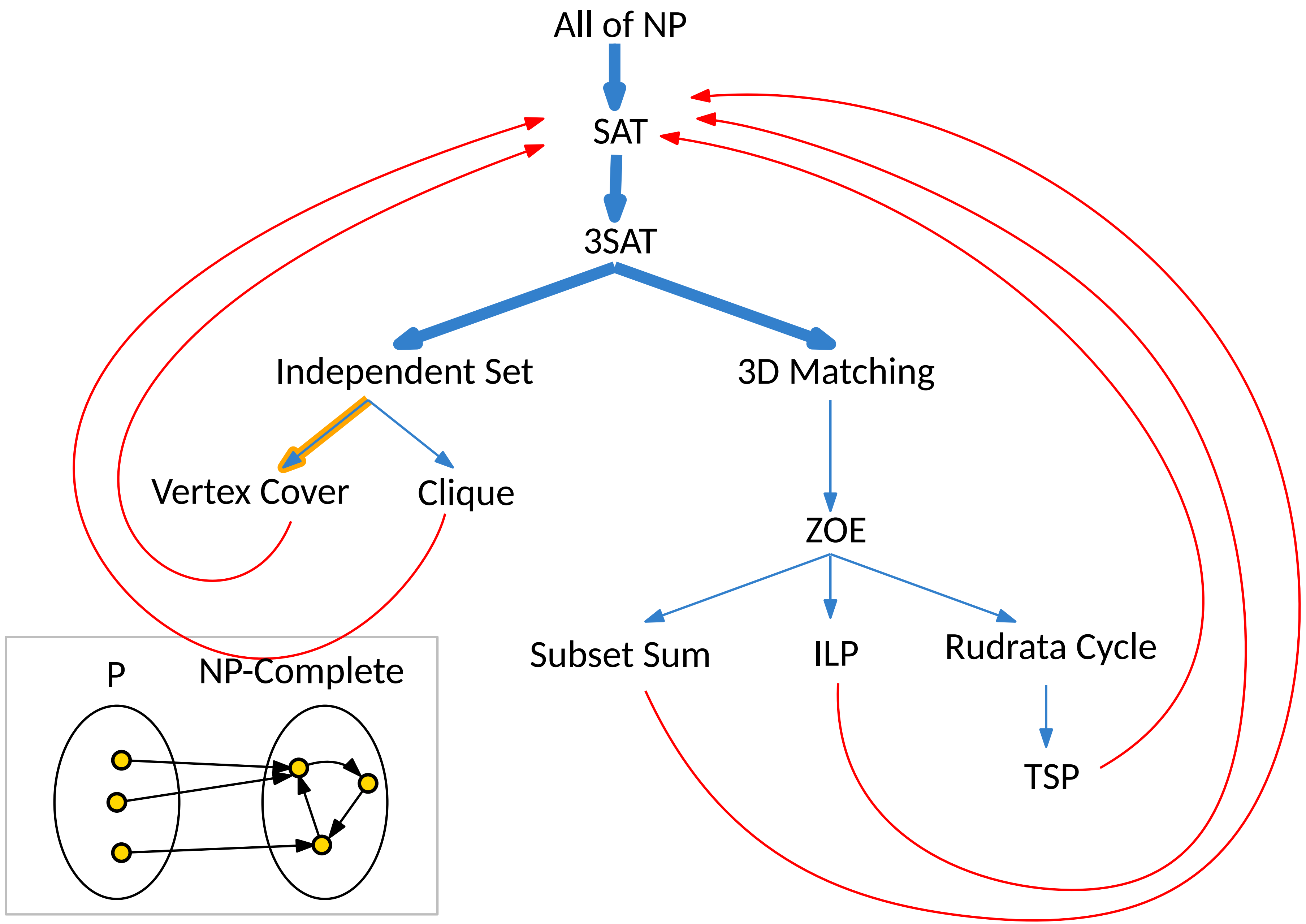
Definition. A search problem is NP-complete if all other search problems reduce to it.

Once we know A is NP-complete, we can use it to prove that a new search problem B is also NP-complete, simply by reducing A to B.
This means that all problems in NP reduces to B via A.





Reductions



All of NP
SAT
3SAT
Independent Set
3D Matching
Vertex Cover
Clique
ZOE
Subset Sum
ILP
Rudrata Cycle
TSP
P
NP-Complete

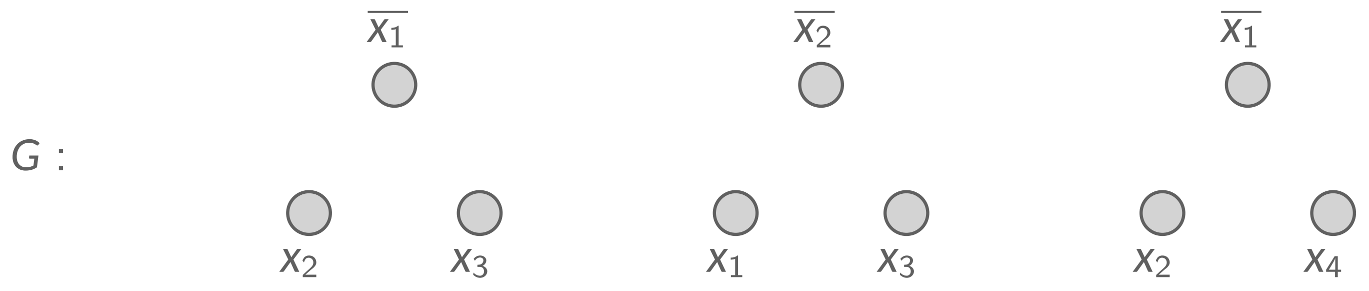
3SAT $\rightarrow$ Ind. Set

Claim. 3SAT $\leqslant_P$ Independent-Set.

Proof. Given an instance Φ of 3SAT, we construct an instance (G, k) of Independent-Set that has an independent set of size k iff Φ is satisfiable.

Construction of an instance of Independent-Set from an instance of 3SAT:

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



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

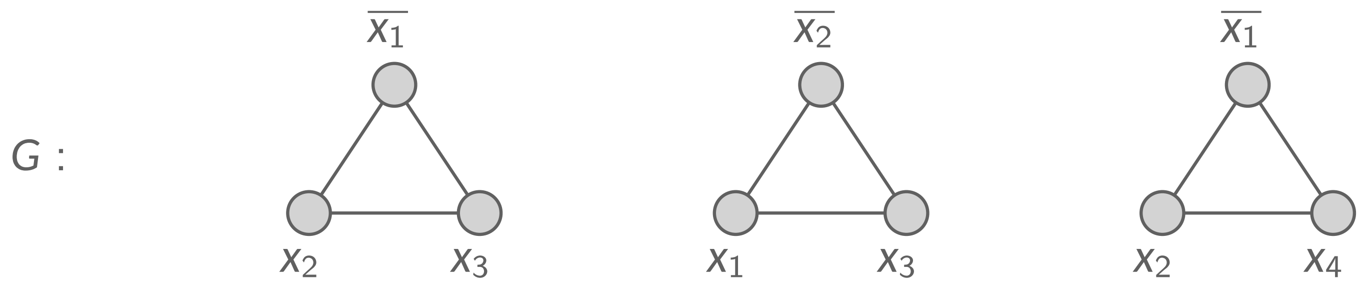
3SAT $\rightarrow$ Ind. Set

Claim. 3SAT $\leqslant_P$ Independent-Set.

Proof. Given an instance Φ of 3SAT, we construct an instance (G, k) of Independent-Set that has an independent set of size k iff Φ is satisfiable.

Construction of an instance of Independent-Set from an instance of 3SAT:

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



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

3SAT $\rightarrow$ Ind. Set

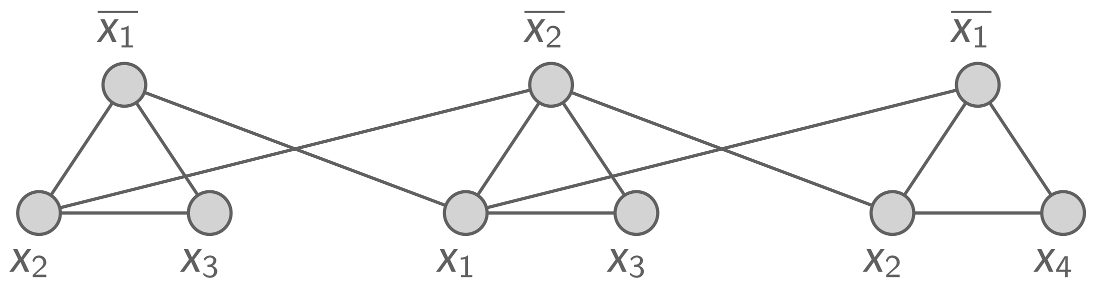
Claim. 3SAT $\leqslant_P$ Independent-Set.

Proof. Given an instance Φ of 3SAT, we construct an instance (G, k) of Independent-Set that has an independent set of size k iff Φ is satisfiable.

Construction of an instance of Independent-Set from an instance of 3SAT:

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

G :



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

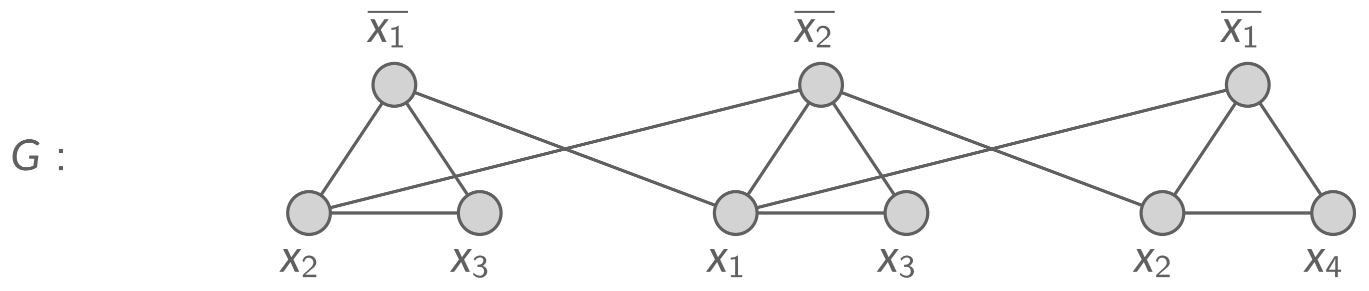
3SAT $\rightarrow$ Ind. Set

Claim. 3SAT $\leqslant_P$ Independent-Set.

Proof. Given an instance Φ of 3SAT, we construct an instance (G, k) of Independent-Set that has an independent set of size k iff Φ is satisfiable.

Let S be independent set of size k. S must contain exactly one vertex in each triangle. Set these literals to true and any other variables in a consistent way. Truth assignment is consistent and all clauses are satisfied.

Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k.



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

Clique : NP-completeness

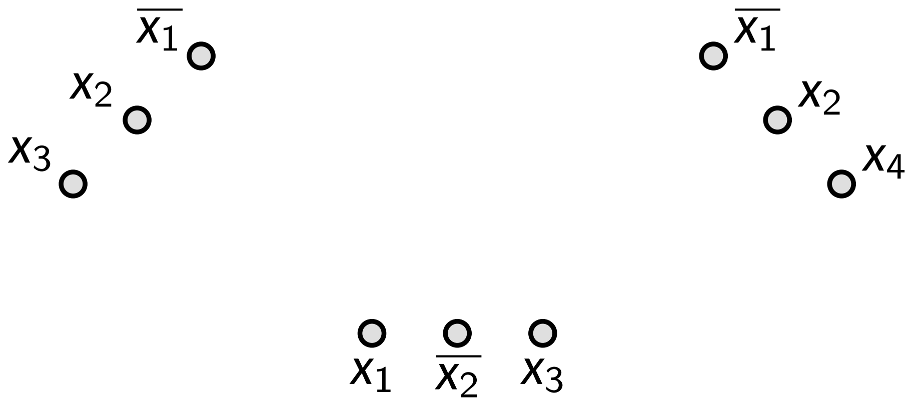
Clique (search problem). given a graph and an integer g, find g vertices s.t. all possible edges between them are present.

3SAT (search problem). given a Boolean formula (at most 3 literals in a clause) in CNF, either find a satisfying true assignment or report "none exists".

Claim. 3SAT $\leqslant_p$ Clique.

Proof. Given an instance Φ of 3SAT consisting of k clauses, we construct an instance (G, k) of Clique that has a clique of size k iff Φ is satisfiable.

- G contains 3 vertices for each clause, one for each literal.
- Connect a literal u to other literal v if they belong to different clauses and $u \neq \overline{v}$.



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

Clique : NP-completeness

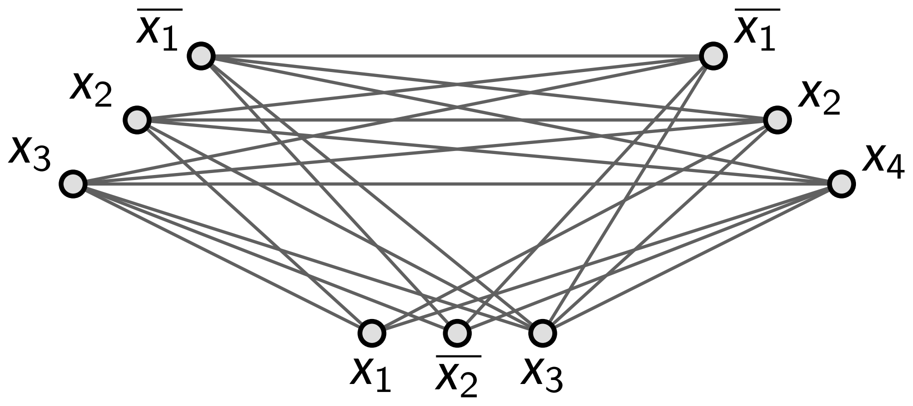
Clique (search problem). given a graph and an integer g, find g vertices s.t. all possible edges between them are present.

3SAT (search problem). given a Boolean formula (at most 3 literals in a clause) in CNF, either find a satisfying true assignment or report "none exists".

Claim. 3SAT $\leqslant_p$ Clique.

Proof. Given an instance Φ of 3SAT consisting of k clauses, we construct an instance (G, k) of Clique that has a clique of size k iff Φ is satisfiable.

- G contains 3 vertices for each clause, one for each literal.
- Connect a literal u to other literal v if they belong to different clauses and $u \neq \overline{v}$.



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

Clique : NP-completeness

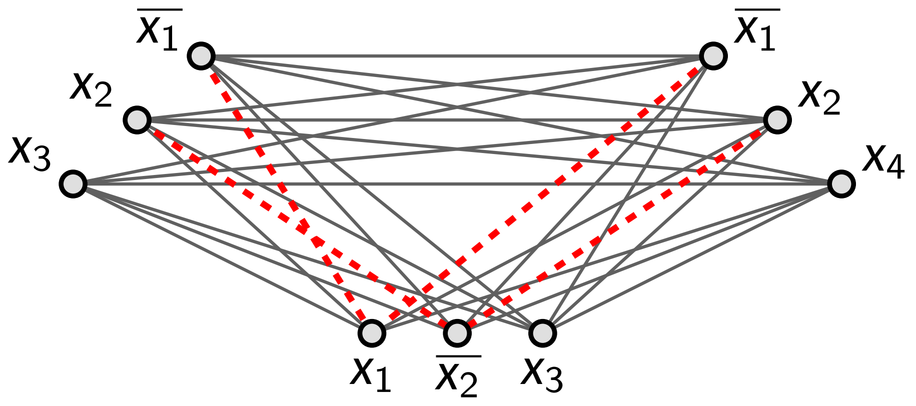
Clique (search problem). given a graph and an integer g, find g vertices s.t. all possible edges between them are present.

3SAT (search problem). given a Boolean formula (at most 3 literals in a clause) in CNF, either find a satisfying true assignment or report "none exists".

Claim. 3SAT $\leqslant_p$ Clique.

Proof. Given an instance Φ of 3SAT consisting of k clauses, we construct an instance (G, k) of Clique that has a clique of size k iff Φ is satisfiable.

- G contains 3 vertices for each clause, one for each literal.
- Connect a literal u to other literal v if they belong to different clauses and $u \neq \overline{v}$.



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

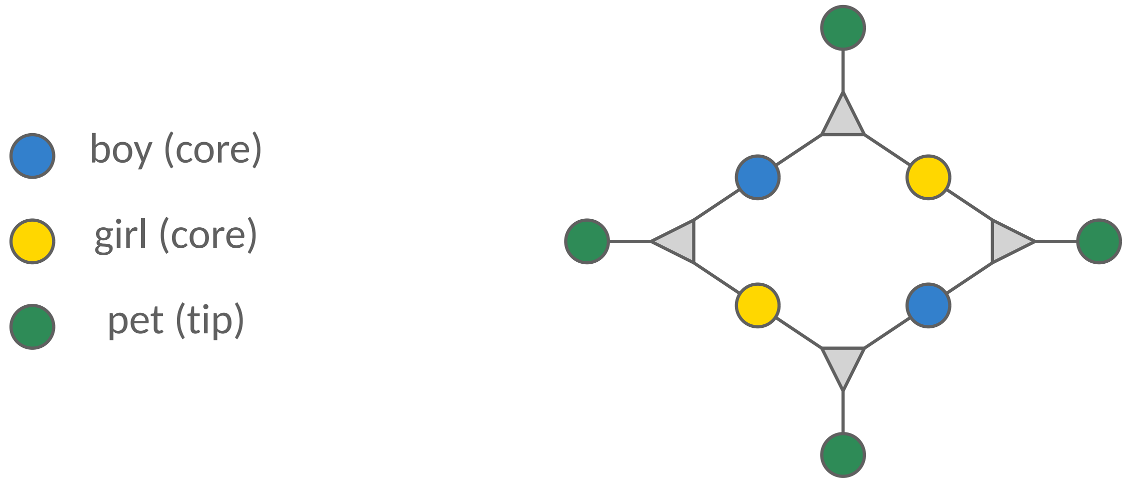
3SAT $\longrightarrow$ 3D Matching

Claim. 3SAT $\leqslant_P$ 3D Matching.

Proof. Given an instance Φ of 3SAT, we construct an instance of 3D Matching that has a perfect matching iff Φ is satisfiable.
We first apply the reduction from 3SAT to its constrained version to ensure that no literal appears more than twice.

Construction (part 1) of an instance of 3D Matching from an instance of 3SAT:

- Create a gadget for each variable x_i with 2 cores and 4 tip elements.
- No other triples will use core elements.
- In gadget i, 3D-matching must use either both grey triples or both blue ones.



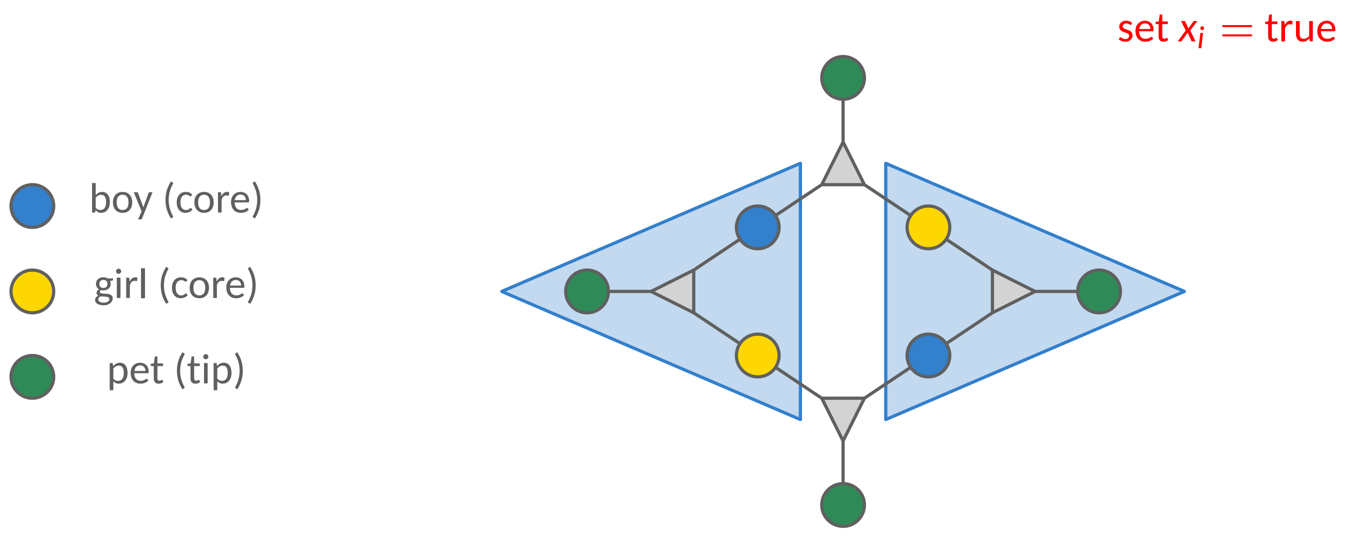
3SAT $\rightarrow$ 3D Matching

Claim. 3SAT $\leqslant_P$ 3D Matching.

Proof. Given an instance Φ of 3SAT, we construct an instance of 3D Matching that has a perfect matching iff Φ is satisfiable.
We first apply the reduction from 3SAT to its constrained version to ensure that no literal appears more than twice.

Construction (part 1) of an instance of 3D Matching from an instance of 3SAT:

- Create a gadget for each variable x_i with 2 cores and 4 tip elements.
- No other triples will use core elements.
- In gadget i, 3D-matching must use either both grey triples or both blue ones.



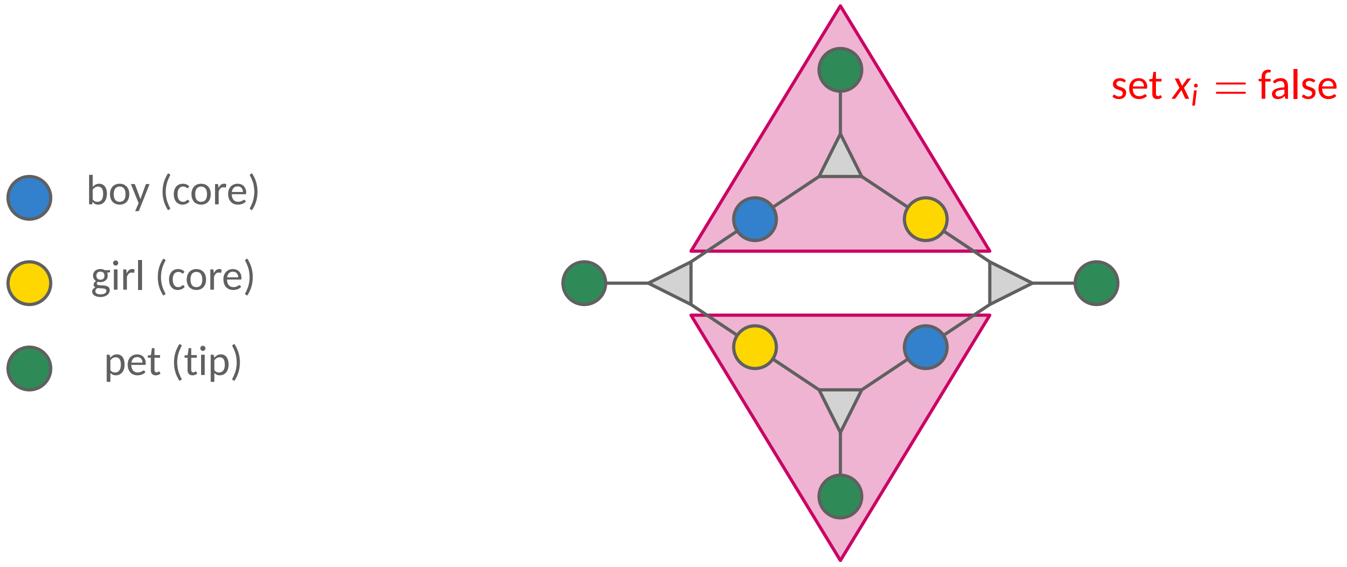
3SAT $\rightarrow$ 3D Matching

Claim. 3SAT $\leqslant_P$ 3D Matching.

Proof. Given an instance Φ of 3SAT, we construct an instance of 3D Matching that has a perfect matching iff Φ is satisfiable.
We first apply the reduction from 3SAT to its constrained version to ensure that no literal appears more than twice.

Construction (part 1) of an instance of 3D Matching from an instance of 3SAT:

- Create a gadget for each variable x_i with 2 cores and 4 tip elements.
- No other triples will use core elements.
- In gadget i, 3D-matching must use either both grey triples or both blue ones.

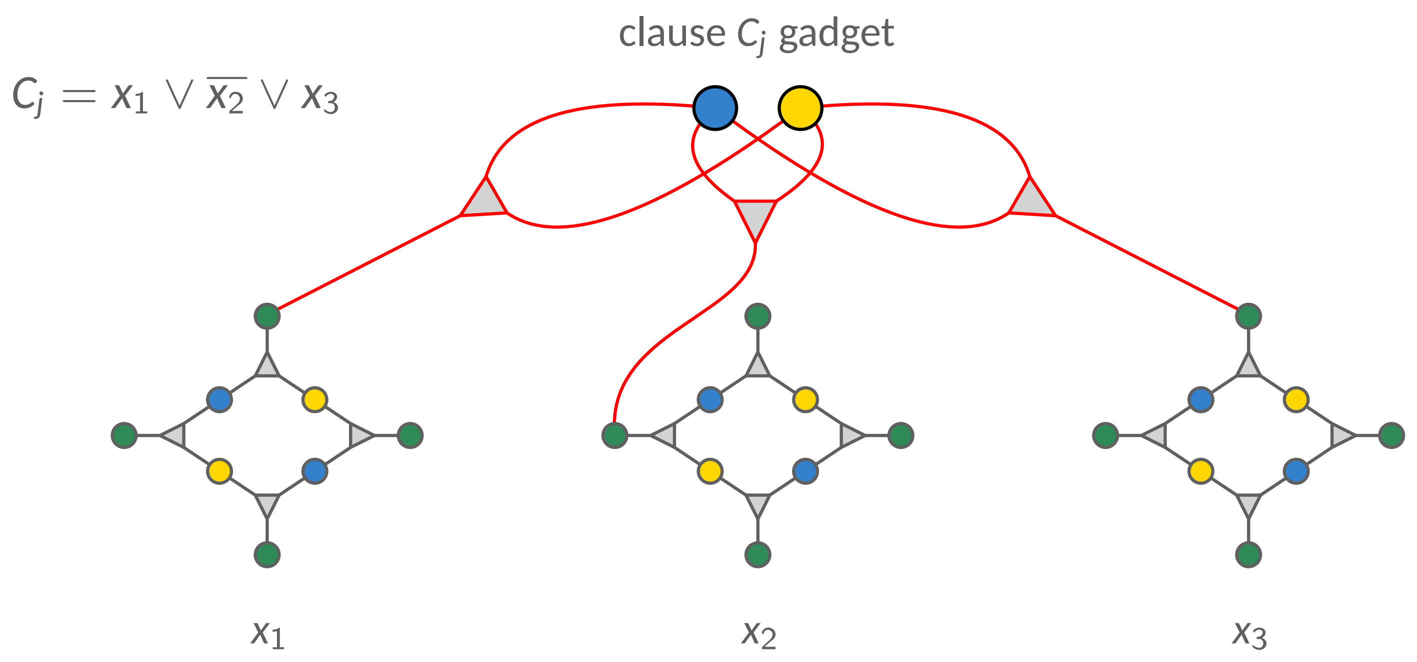


3SAT $\rightarrow$ 3D Matching

Claim. 3SAT $\leqslant_P$ 3D Matching.

Construction (part 2) of an instance of 3D Matching from an instance of 3SAT:

- For each clause C_j, create two elements (b-g couple) and three triples.
- Exactly one of these triples will be used in any 3D-matching.
- This ensures any 3D-matching uses either (i) grey core of x_1 or (ii) blue core of x_2 or (iii) grey core of x_3.
- Add $2n - m$ cleanup gadgets (b-g couples) and form triples with every tip.



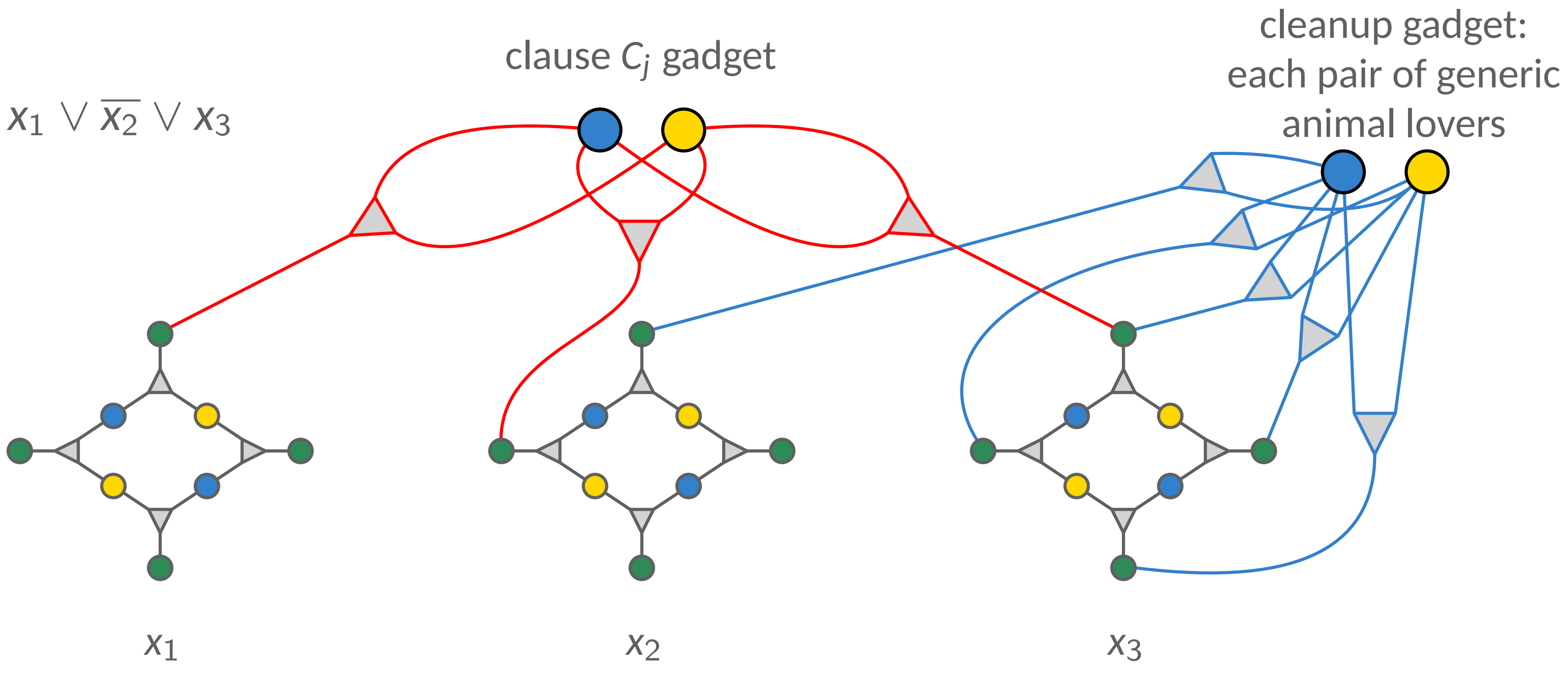
3SAT $\rightarrow$ 3D Matching

Claim. 3SAT $\leqslant_P$ 3D Matching.

Construction (part 2) of an instance of 3D Matching from an instance of 3SAT:

- For each clause C_j, create two elements (b-g couple) and three triples.
- Exactly one of these triples will be used in any 3D-matching.
- This ensures any 3D-matching uses either (i) grey core of x_1 or (ii) blue core of x_2 or (iii) grey core of x_3.
- Add $2n - m$ cleanup gadgets (b-g couples) and form triples with every tip.

$C_j = x_1 \vee \overline{x_2} \vee x_3$

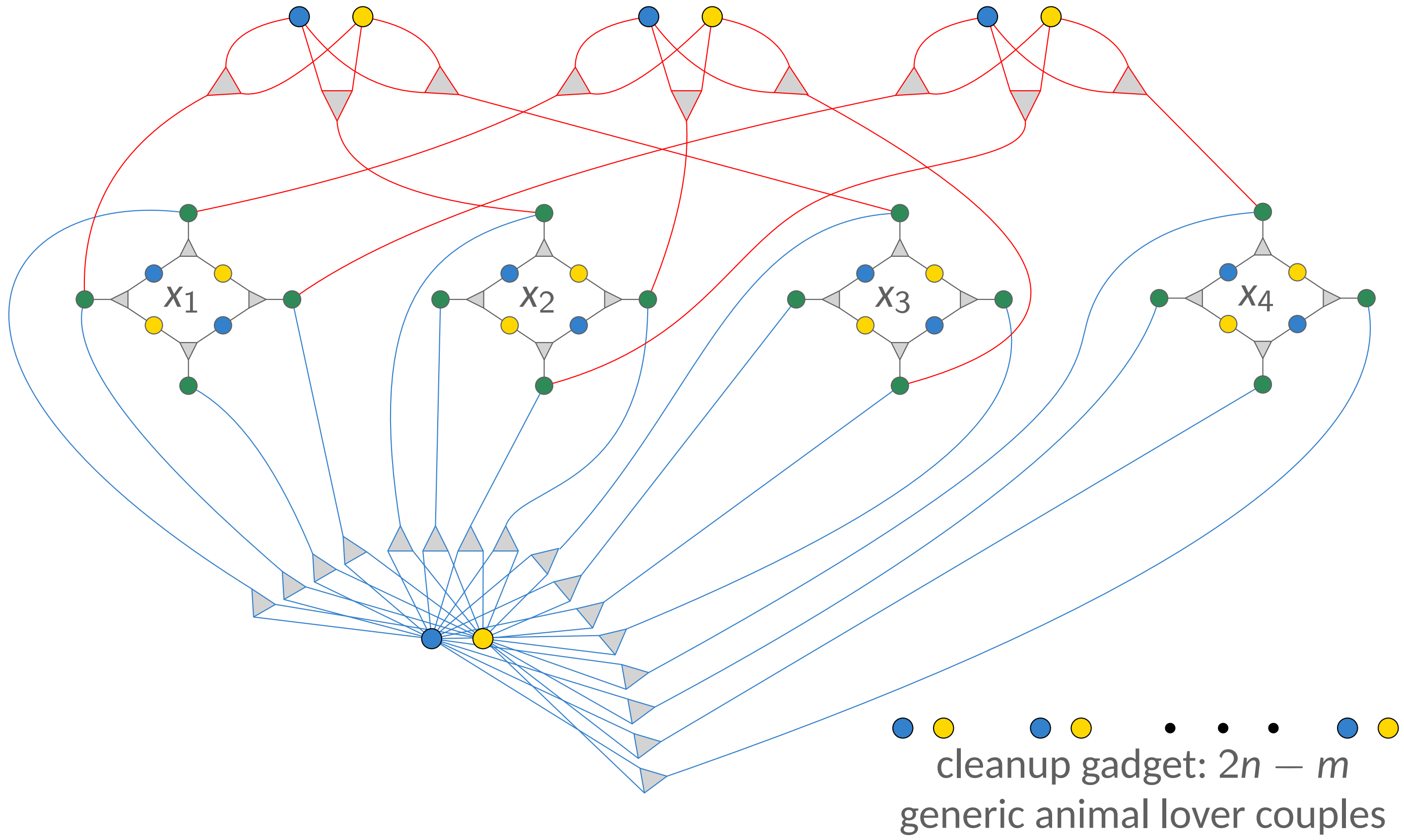


3SAT → 3D Matching

Claim. Instance has a 3D matching iff Φ is satisfiable.

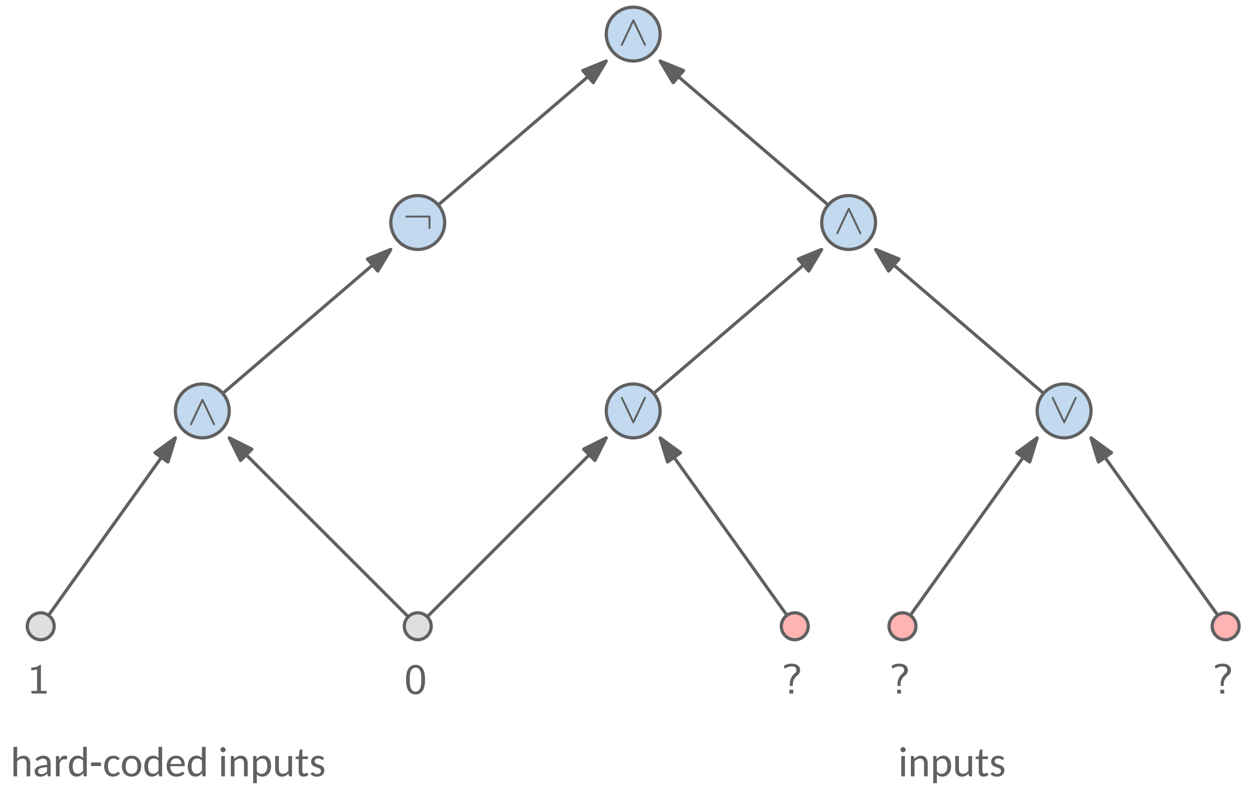
- From any perfect matching, a satisfying truth assignment can be recovered.
- From any satisfying truth assignment, all gadgets can be matched.

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



Circuit Satisfiability

Circuit-SAT. Given a combinatorial circuit built out of `AND`, `OR`, and `NOT` gates, is there a way to set the circuit inputs so that the output is 1?



Circuit-SAT is a generalization of SAT.

Circuit-SAT $\longrightarrow$ 3SAT

Theorem. 3SAT is NP-complete, that is, Circuit-SAT $\leqslant_P$ 3SAT.

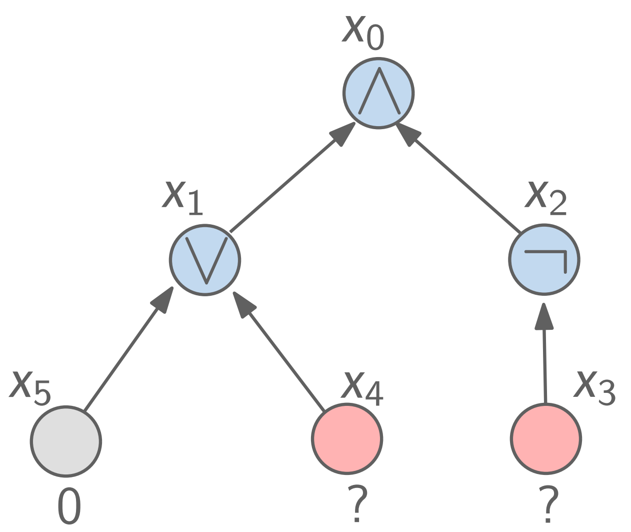
Proof. (Sketch) Let K be any circuit. Create a 3SAT variable x_i for each circuit element i. Make circuit compute correct values at each node:

- If $x_2 = \neg x_3$, add 2 clauses: $(x_2 \vee x_3)(\overline{x_2} \vee \overline{x_3})$.
- If $x_1 = x_4 \vee x_5$, add 3 clauses: $(x_1 \vee \overline{x_4})(x_1 \vee \overline{x_5})(\overline{x_1} \vee x_4 \vee x_5)$.
- If $x_0 = x_1 \wedge x_2$, add 3 clauses: $(\overline{x_0} \vee x_1)(\overline{x_0} \vee x_2)(x_0 \vee \overline{x_1} \vee \overline{x_2})$.

Hard-coded input values and output value.

- If $x_5 = 0$, add 1 clause: $(\overline{x_5})$
- If $x_5 = 1$, add 1 clause: (x_5).

Finally, turn clauses of length < 3 into clauses of length exactly 3.



Any NP problem $\rightarrow$ Circuit-SAT

Theorem. Circuit-SAT is NP-complete. [Cook 1971, Levin 1973]

Proof. (Sketch)
Any algorithm that takes a fixed number of bits n as input and produces a yes/no answer can be represented by such a circuit. Moreover, if algorithm takes poly-time, then circuit is of poly-size.

Consider some problem X in NP. It has a poly-time certifier $C(s, t)$. To determine whether s is in X, we need to know if there exists a certificate t of length $p(|s|)$ such that $C(s, t) = \texttt{yes}$. Now problem X is to find a solution that satisfies $C(s, t)$!

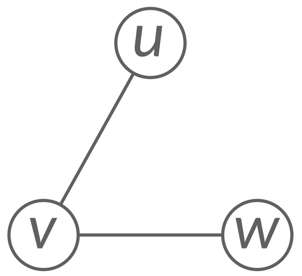
View $C(s, t)$ as an algorithm on $|s| + p(|s|)$ bits (input s, certificate t) and convert it into a poly-size circuit K.

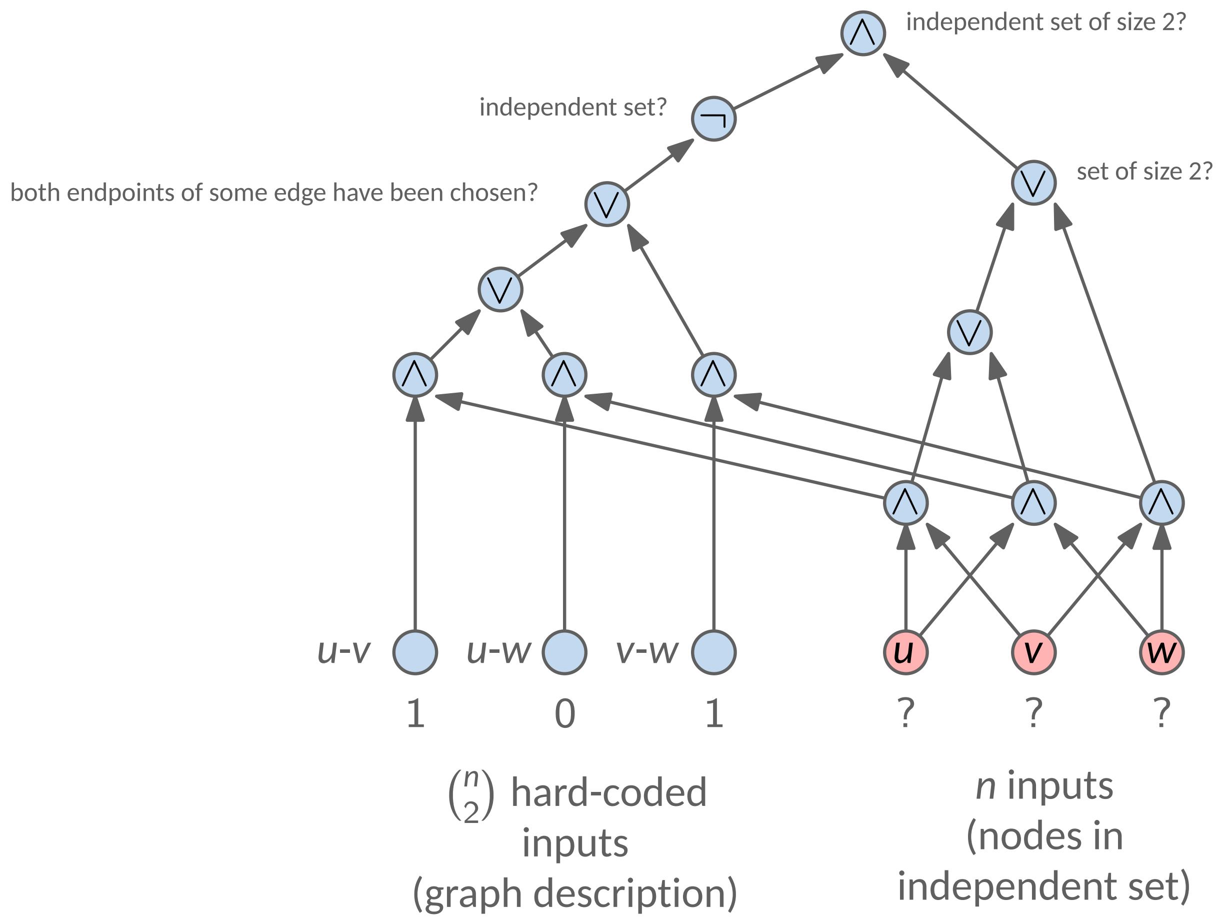
- first $|s|$ bits are hard-coded with s.
- remaining $p(|s|)$ bits represent bits of t.

Circuit K is satisfiable iff $C(s, t) = \texttt{yes}$.

Example

Construction below creates a circuit K whose inputs can be set so that K outputs true iff graph G has an independent set of size 2.




independent set of size 2?
independent set?
both endpoints of some edge have been chosen?
set of size 2?
u-v
u-w
v-w
u
v
w
1
0
1
?
?
?
$\binom{n}{2}$ hard-coded inputs (graph description)
n inputs (nodes in independent set)