NP-complete Problems: Reductions

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set $\rightarrow$ Vertex Cover
4. 3-SAT $\rightarrow$ Independent Set
5. SAT $\rightarrow$ 3-SAT
6. All of **NP** $\rightarrow$ SAT
7. Using SAT-solvers

Informally

We say that a search problem A is reduced to a search problem B and write $A \to B$, if a polynomial time algorithm for B can be used (as a black box) to solve A in polynomial time.

Reduction: $A \to B$

instance I of A

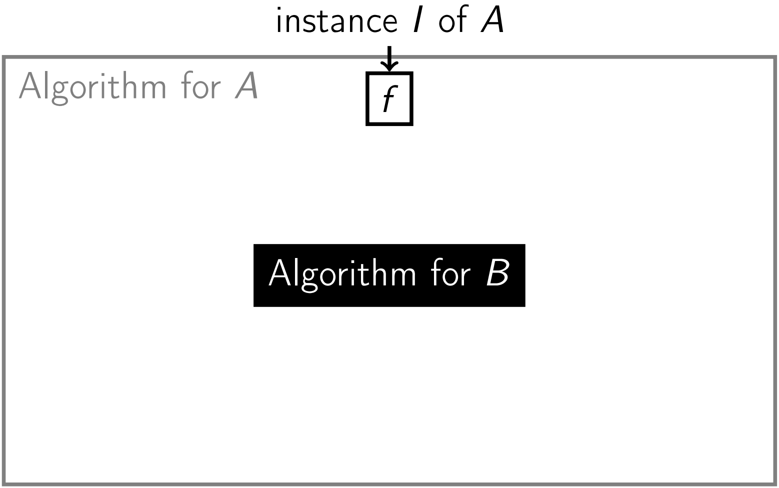
Reduction: $A \to B$

instance I of A

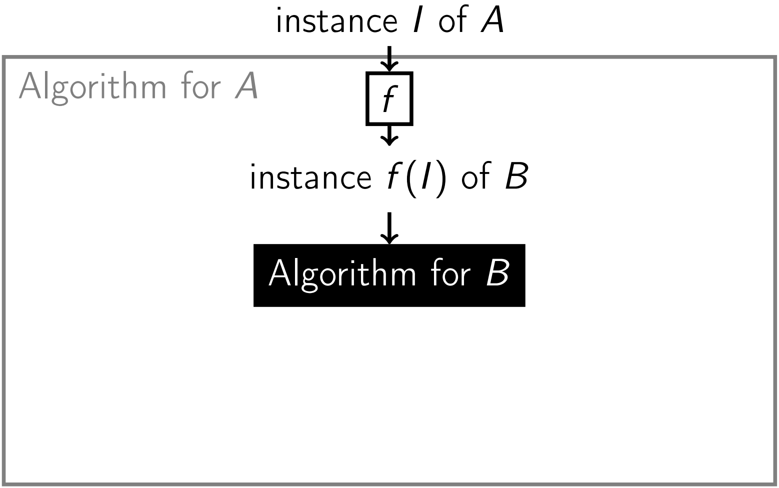
Algorithm for A

Algorithm for B

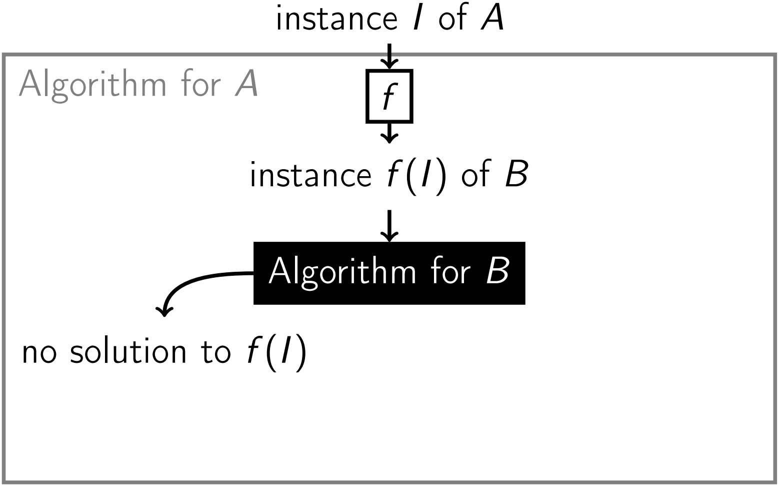
Reduction: $A \to B$



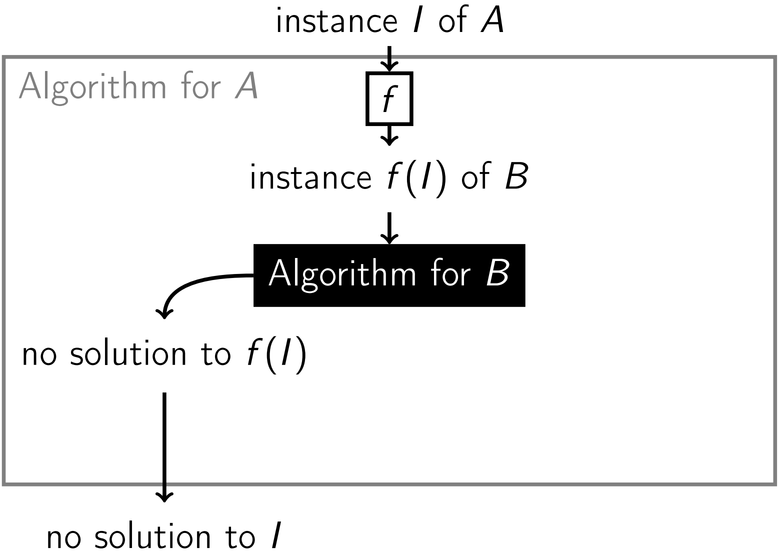
Reduction: $A \to B$



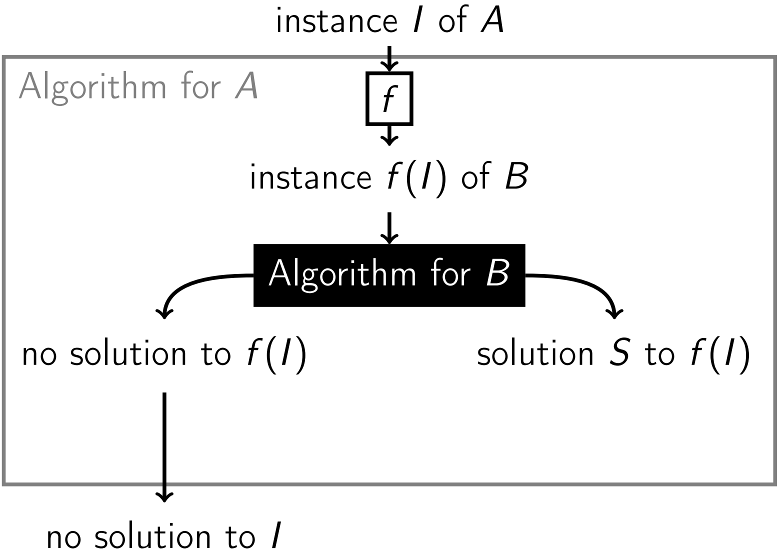
Reduction: $A \to B$



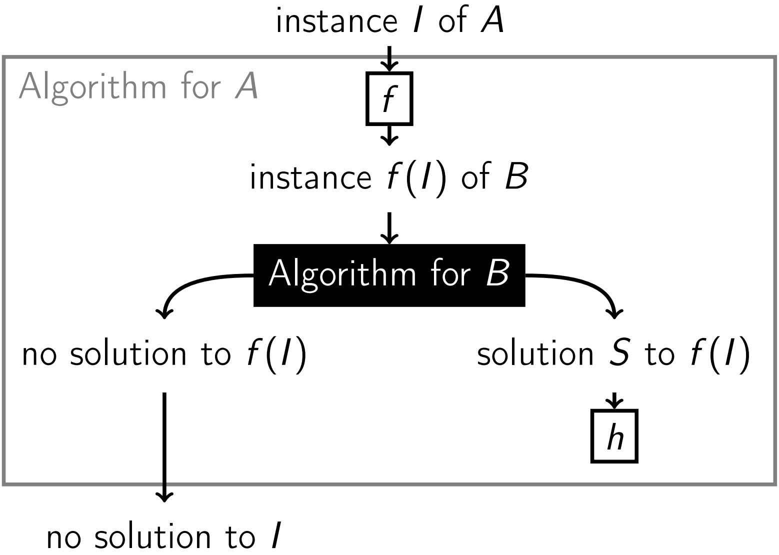
Reduction: $A \to B$



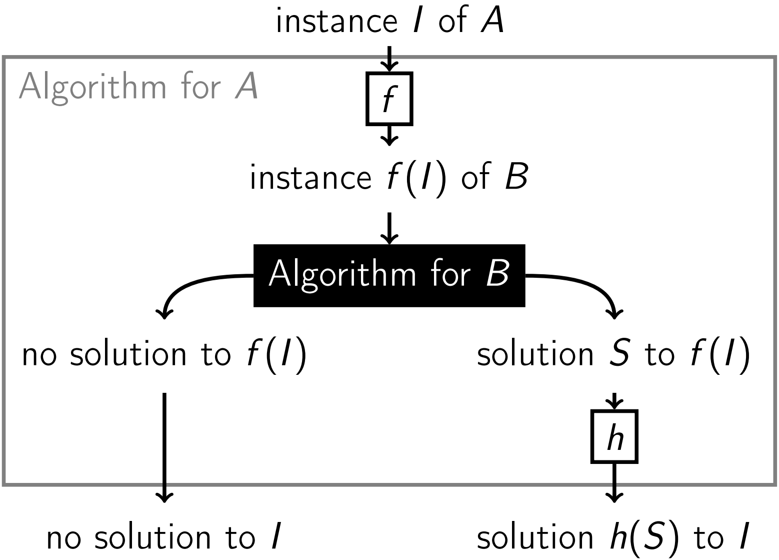
Reduction: $A \to B$



Reduction: $A \to B$



Reduction: $A \to B$

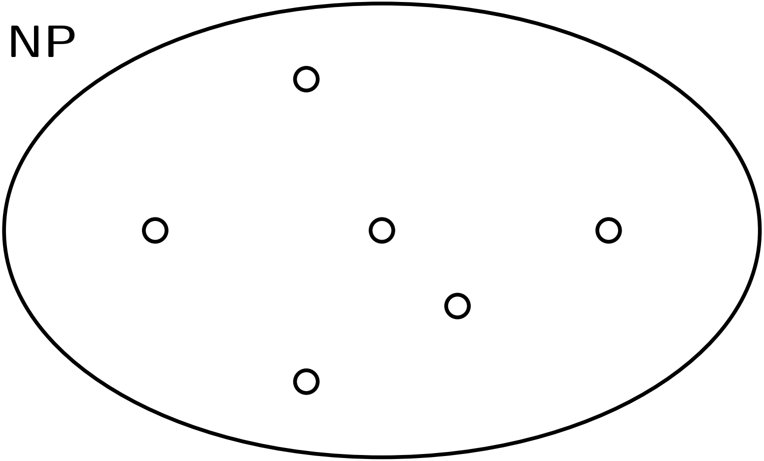


Formally

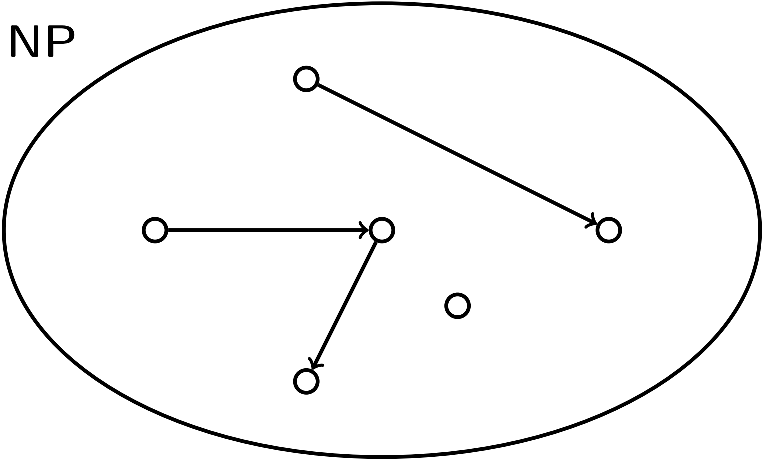
Definition

We say that a search problem A is reduced to a search problem B and write $A \to B$, if there exists a polynomial time algorithm f that converts any instance I of A into an instance $f(I)$ of B, together with a polynomial time algorithm h that converts any solution S to $f(I)$ back to a solution $h(S)$ to A. If there is no solution to $f(I)$, then there is no solution to I.

Graph of Search Problems



Graph of Search Problems



NP-complete Problems

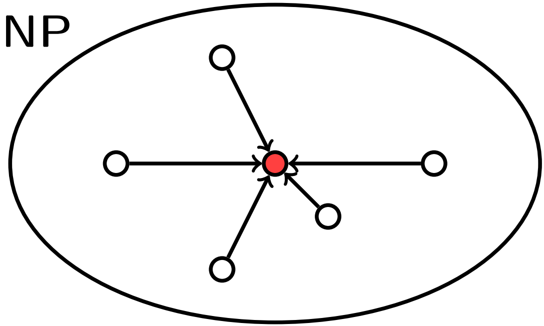
Definition

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

NP-complete Problems

Definition

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



Do they exist?

It's not at all immediate that **NP**-complete problems even exist. We'll see later that all hard problems that we've seen in the previous part are in fact **NP**-complete!

Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set → Vertex Cover
4. 3-SAT → Independent Set
5. SAT → 3-SAT
6. All of **NP** → SAT
7. Using SAT-solvers

Two ways of using $A \to B$:

1. if B is easy (can be solved in polynomial time), then so is A
2. if A is hard (cannot be solved in polynomial time), then so is B

Reductions Compose

Lemma

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

Proof

- The reductions $A \to B$ and $B \to C$ are given by pairs of polytime algorithms (f_{AB}, h_{AB}) and (f_{BC}, h_{BC}).

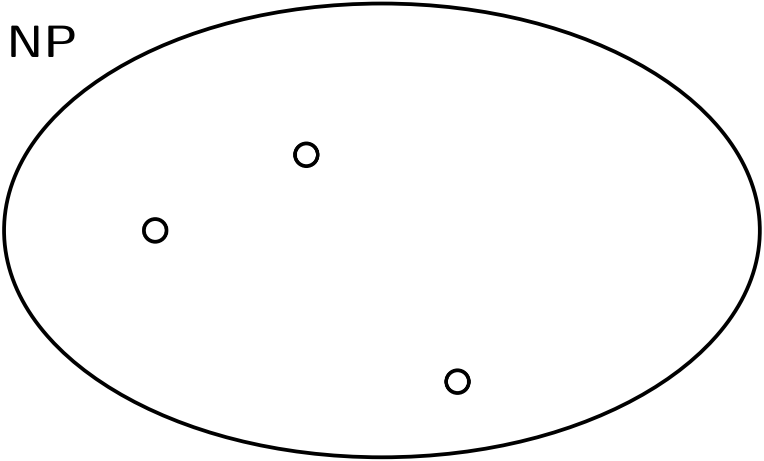
Proof

- The reductions $A \to B$ and $B \to C$ are given by pairs of polytime algorithms (f_{AB}, h_{AB}) and (f_{BC}, h_{BC}).
- To transform an instance I_A of A to an instance I_C of C we apply a polytime algorithm $f_{BC} \circ f_{AB}$: $I_C = f_{BC}(f_{AB}(I_A))$.

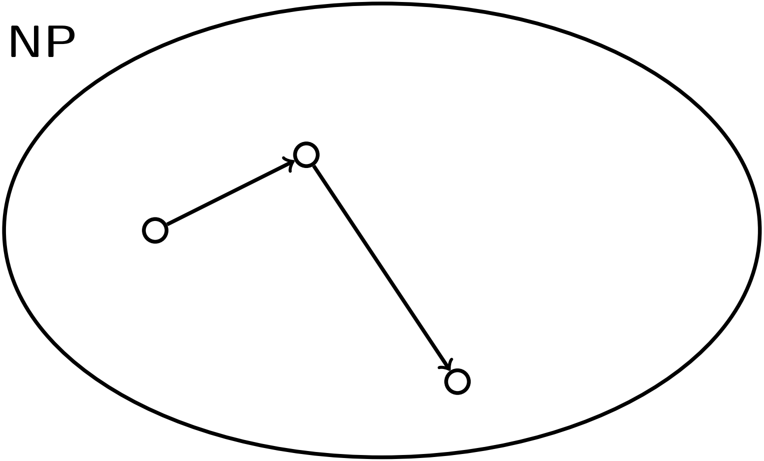
Proof

- The reductions $A \to B$ and $B \to C$ are given by pairs of polytime algorithms (f_{AB}, h_{AB}) and (f_{BC}, h_{BC}).
- To transform an instance I_A of A to an instance I_C of C we apply a polytime algorithm $f_{BC} \circ f_{AB}$: $I_C = f_{BC}(f_{AB}(I_A))$.
- To transform a solution S_C to I_C to a solution S_A to I_A we apply a polytime algorithm $h_{AB} \circ h_{BC}$: $S_A = h_{AB}(h_{BC}(S_C))$. $\square$

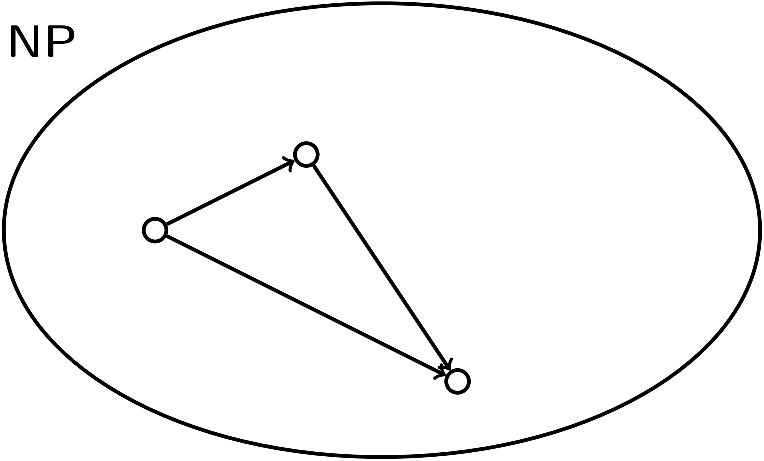
Pictorially



Pictorially



Pictorially



Showing **NP**-completeness

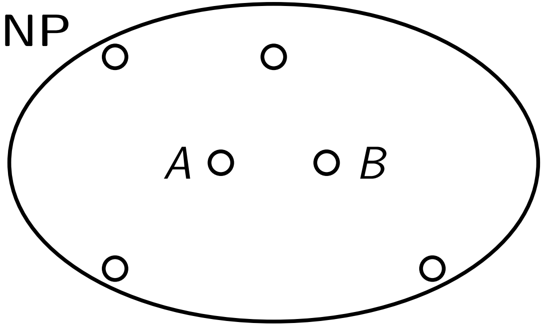
Corollary

If $A \to B$ and A is **NP**-complete, then so is B.

Showing **NP**-completeness

Corollary

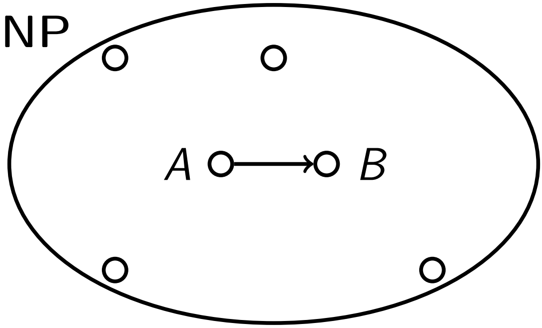
If $A \to B$ and A is **NP**-complete, then so is B.



Showing **NP**-completeness

Corollary

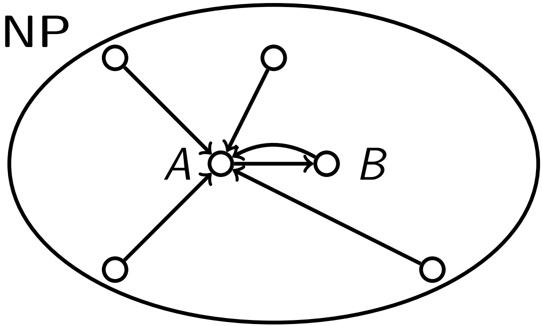
If $A \to B$ and A is **NP**-complete, then so is B.



Showing **NP**-completeness

Corollary

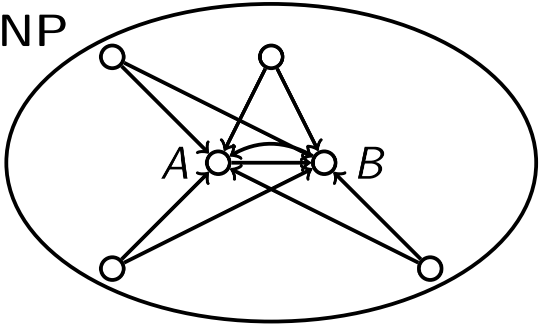
If $A \rightarrow B$ and A is **NP**-complete, then so is B.



Showing **NP**-completeness

Corollary

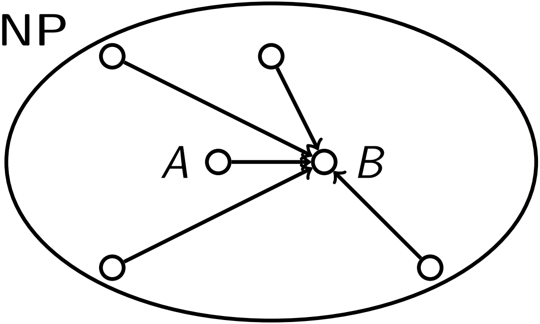
If $A \to B$ and A is **NP**-complete, then so is B.



Showing **NP**-completeness

Corollary

If $A \to B$ and A is **NP**-complete, then so is B.

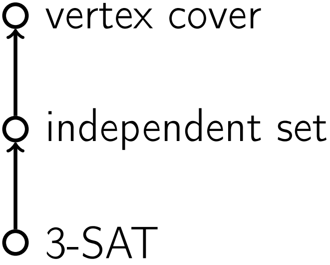


Plan

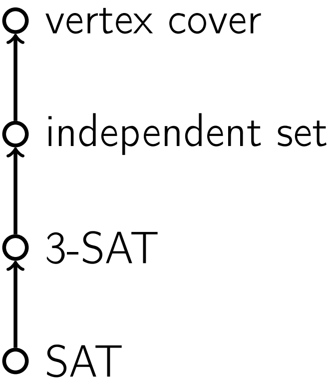
vertex cover

independent set

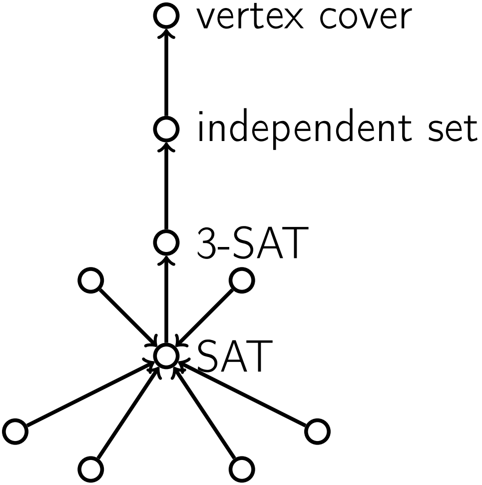
Plan



Plan



Plan



Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set $\rightarrow$ Vertex Cover
4. 3-SAT $\rightarrow$ Independent Set
5. SAT $\rightarrow$ 3-SAT
6. All of **NP** $\rightarrow$ SAT
7. Using SAT-solvers

Independent set

Input: A graph and a budget b.

Output: A subset of at least b vertices such that no two of them are adjacent.

Independent set

Input: A graph and a budget b.

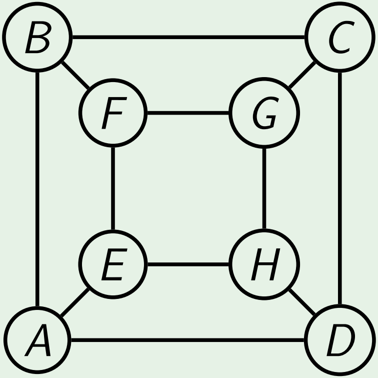
Output: A subset of at least b vertices such that no two of them are adjacent.

Vertex cover

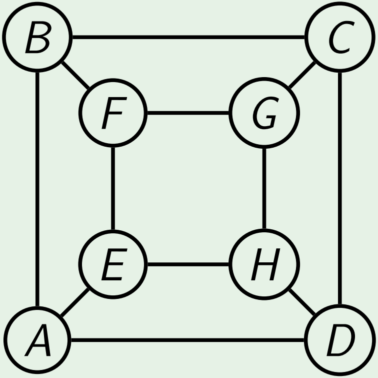
Input: A graph and a budget b.

Output: A subset of at most b vertices that touches every edge.

Example

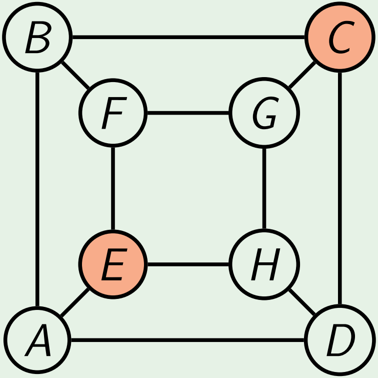


Example



Independent sets:

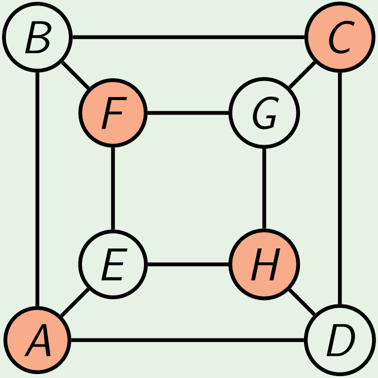
Example



Independent sets:

$\{E, C\}$

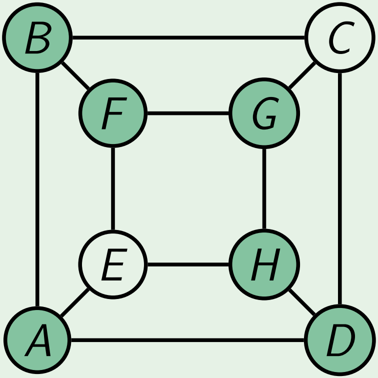
Example



Independent sets:

$\{E, C\}$ $\{A, C, F, H\}$

Example



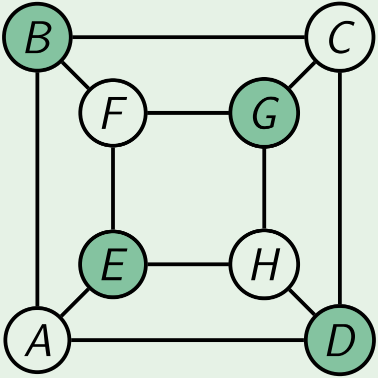
Independent sets:

$\{E, C\}$ $\{A, C, F, H\}$

Vertex covers:

$\{A, B, D, F, G, H\}$

Example



Independent sets:

$\{E, C\}$ $\{A, C, F, H\}$

Vertex covers:

$\{A, B, D, F, G, H\}$

$\{B, D, E, G\}$

Lemma

I is an independent set of $G(V, E)$, if and only if $V - I$ is a vertex cover of G.

Proof

- $\Rightarrow$ If I is an independent set, then there is no edge with both endpoints in I. Hence $V - I$ touches every edge.
- $\Leftarrow$ If $V - I$ touches every edge, then each edge has at least one endpoint in $V - I$. Hence I is an independent set. $\square$

Reduction

Independent set $\rightarrow$ vertex cover: to check whether $G(V, E)$ has an independent set of size at least b, check whether G has a vertex cover of size at most $|V| - b$:

- $f(G(V, E), b) = (G(V, E), |V| - b)$
- $h(S) = V - S$

Outline

3-SAT

Input: Formula F in 3-CNF (a collection of clauses each having at most three literals).

Output: An assignment of Boolean values to the variables of F satisfying all clauses, if exists.

Goal

Design a polynomial time algorithm that, given a 3-CNF formula F, outputs a graph G and an integer b, such that:

> *F is satisfiable, if and only if G has an independent set of size at least b.*

We need to find an assignment of Boolean values to variables, such that each clause contains at least one satisfied literal.

We need to find an assignment of Boolean values to variables, such that each clause contains at least one satisfied literal.

Example

- Setting $x = 1, y = 1, z = 1$ satisfies a formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$.

We need to find an assignment of Boolean values to variables, such that each clause contains at least one satisfied literal.

Example

- Setting $x = 1, y = 1, z = 1$ satisfies a formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$.
- Setting $x = 1, y = 0, z = 0$ also satisfies it: $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$.

Alternatively, we need to select at least one literal from each clause, such that the set of selected literals is consistent: it does not contain a literal ℓ together with its negation $\overline{\ell}$.

Alternatively, we need to select at least one literal from each clause, such that the set of selected literals is consistent: it does not contain a literal ℓ together with its negation $\overline{\ell}$.

Example: $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$

- Consistent: $\{x, x, \overline{z}\}$, $\{x, x, y\}$, $\{x, \overline{y}, \overline{z}\}$.

Alternatively, we need to select at least one literal from each clause, such that the set of selected literals is consistent: it does not contain a literal ℓ together with its negation $\overline{\ell}$.

Example: $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$

- Consistent: $\{x, x, \overline{z}\}$, $\{x, x, y\}$, $\{x, \overline{y}, \overline{z}\}$.
- Inconsistent: $\{y, \overline{y}, \overline{z}\}$, $\{z, x, \overline{z}\}$.

Using Alternative Statement

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

Using Alternative Statement

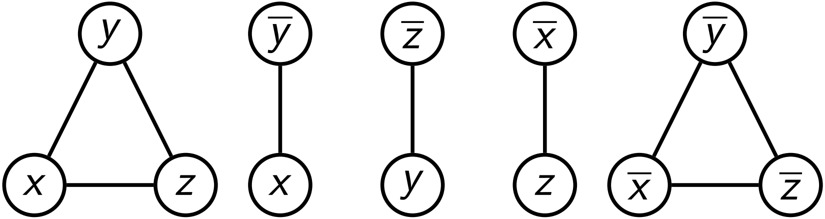
$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

y $\overline{y}$ $\overline{z}$ $\overline{x}$ $\overline{y}$

x z x y z $\overline{x}$ $\overline{z}$

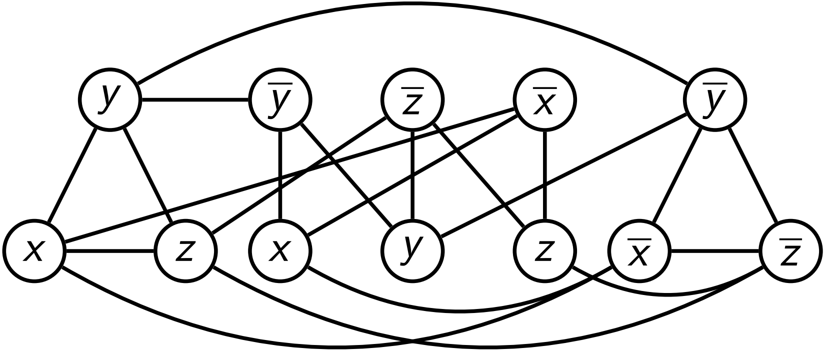
Using Alternative Statement

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



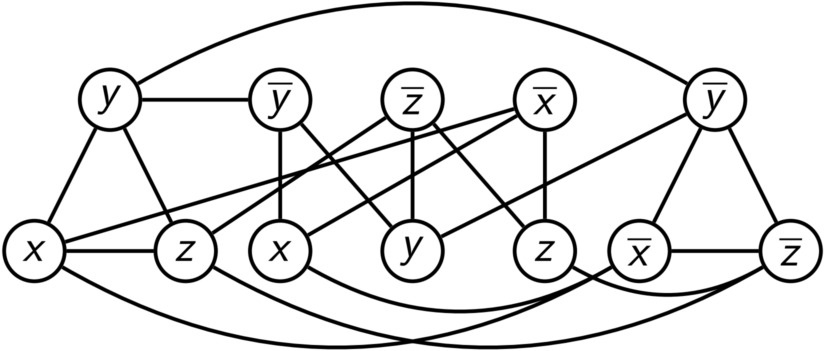
Using Alternative Statement

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



Using Alternative Statement

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



the formula is satisfiable iff the resulting graph has independent set of size 5

Transforming an Instance

- For each clause of the input formula F, introduce three (or two, or one) vertices in G labeled with the literals of this clause. Join every two of them.

Transforming an Instance

- For each clause of the input formula F, introduce three (or two, or one) vertices in G labeled with the literals of this clause. Join every two of them.
- Join every pair of vertices labeled with complementary literals.

Transforming an Instance

- For each clause of the input formula F, introduce three (or two, or one) vertices in G labeled with the literals of this clause. Join every two of them.
- Join every pair of vertices labeled with complementary literals.
- F is satisfiable if and only if G has independent set of size equal to the number of clauses in F.

Transforming an Instance

- For each clause of the input formula F, introduce three (or two, or one) vertices in G labeled with the literals of this clause. Join every two of them.
- Join every pair of vertices labeled with complementary literals.
- F is satisfiable if and only if G has independent set of size equal to the number of clauses in F.
- Transformation takes polynomial time.

Transforming a Solution

- Given a solution S for G, just read the labels of the vertices from S to get a satisfying assignment of F (takes polynomial time).

Transforming a Solution

- Given a solution S for G, just read the labels of the vertices from S to get a satisfying assignment of F (takes polynomial time).
- If there is no solution for G, then F is unsatisfiable: indeed, a satisfying assignment for F would give a required independent set in G.

Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set → Vertex Cover
4. 3-SAT → Independent Set
5. SAT → 3-SAT
6. All of **NP** → SAT
7. Using SAT-solvers

Goal

Transform a CNF formula into an equisatisfiable 3-CNF formula. That is, reduce a problem to its special case.

Transforming an Instance

- We need to get rid of clauses of length more than 3 in an input formula

Transforming an Instance

- We need to get rid of clauses of length more than 3 in an input formula
- Consider such a clause:
  $C = (\ell_1 \vee \ell_2 \vee A)$, where A is an OR of at least two literals.

Transforming an Instance

- We need to get rid of clauses of length more than 3 in an input formula
- Consider such a clause: $C = (\ell_1 \vee \ell_2 \vee A)$, where A is an OR of at least two literals.
- Introduce a fresh variable y and replace C with the following two clauses: $(\ell_1 \vee \ell_2 \vee y)$, $(\overline{y} \vee A)$

Transforming an Instance

- We need to get rid of clauses of length more than 3 in an input formula
- Consider such a clause:
  $C = (\ell_1 \vee \ell_2 \vee A)$, where A is an OR of at least two literals.
- Introduce a fresh variable y and replace C with the following two clauses:
  $(\ell_1 \vee \ell_2 \vee y)$, $(\overline{y} \vee A)$
- The second clause is shorter than C

Transforming an Instance

- We need to get rid of clauses of length more than 3 in an input formula
- Consider such a clause: $C = (\ell_1 \vee \ell_2 \vee A)$, where A is an OR of at least two literals.
- Introduce a fresh variable y and replace C with the following two clauses: $(\ell_1 \vee \ell_2 \vee y)$, $(\overline{y} \vee A)$
- The second clause is shorter than C
- Repeat while there is a long clause

Running time

The running time of the transformation is polynomial: at each iteration we replace a clause with a shorter clause and a 3-clause. Hence the total number of iterations is at most the total number of literals of the initial formula.

Correctness

Lemma

The formulas $F = (\ell_1 \vee \ell_2 \vee A)\ldots$ and $F' = (\ell_1 \vee \ell_2 \vee y)(\overline{y} \vee A)\ldots$ are equisatisfiable.

Proof

$F = (\ell_1 \vee \ell_2 \vee A) \dots$

$F' = (\ell_1 \vee \ell_2 \vee y)(\overline{y} \vee A) \dots$

- $\Rightarrow$ If either ℓ_1 or ℓ_2 is satisfied, set $y = 0$. Otherwise A must be satisfied. Then set $y = 1$.
- $\Leftarrow$ If $(\ell_1 \vee \ell_2 \vee y)(\overline{y} \vee A)$ are satisfied, then so is $(\ell_1 \vee \ell_2 \vee A)$. $\square$

Transforming a Solution

Given a satisfying assignment for F', just throw away the values of all new variables (y's) to get a satisfying assignment of the initial formula.

Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set $\rightarrow$ Vertex Cover
4. 3-SAT $\rightarrow$ Independent Set
5. SAT $\rightarrow$ 3-SAT
6. All of **NP** $\rightarrow$ SAT
7. Using SAT-solvers

Goal

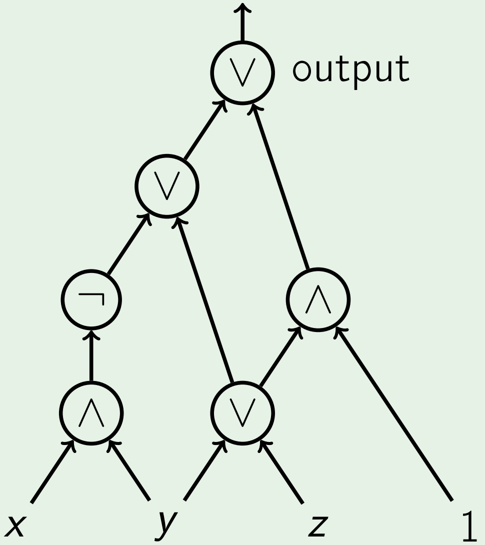
Show that every search problem reduces to SAT.

Goal

Show that every search problem reduces to SAT.

Instead, we show that any problem reduces to Circuit SAT problem, which, in turn, reduces to SAT.

Circuit



Definition

A circuit is a directed acyclic graph of in-degree at most 2. Nodes of in-degree 0 are called inputs and are marked by Boolean variables and constants. Nodes of in-degree 1 and 2 are called gates: gates of in-degree 1 are labeled with NOT, gates of in-degree 2 are labeled with AND or OR. One of the sinks is marked as output.

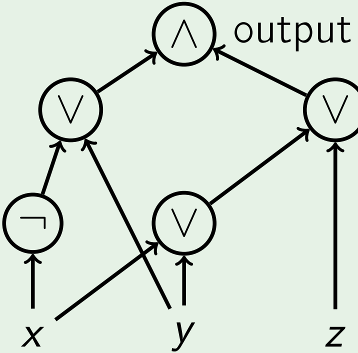
Circuit-SAT

Input: A circuit.

Output: An assignment of Boolean values to the input variables of the circuit that makes the output true.

SAT is a special case of Circuit-SAT as a CNF formula can be represented as a circuit:

Example: $(x \vee y \vee z)(y \vee \overline{x})$



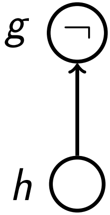
Circuit-SAT $\to$ SAT

To reduce Circuit-SAT to SAT, we need to design a polynomial time algorithm that for a given circuit outputs a CNF formula which is satisfiable, if and only if the circuit is satisfiable

Idea

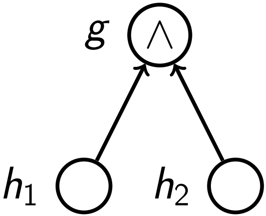
- Introduce a Boolean variable for each gate
- For each gate, write down a few clauses that describe the relationship between this gate and its direct predecessors

NOT Gates



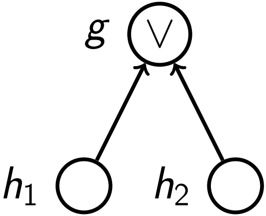
$$(h \vee g)(\overline{h} \vee \overline{g})$$

AND Gates



$$(h_1 \vee \overline{g})(h_2 \vee \overline{g})(\overline{h}_1 \vee \overline{h}_2 \vee g)$$

OR Gates



$$(\overline{h}_1 \vee g)(\overline{h}_2 \vee g)(h_1 \vee h_2 \vee \overline{g})$$

Output Gate

$$g \bigcirc \text{output} \qquad (g)$$

- The resulting CNF formula is consistent with the initial circuit: in any satisfying assignment of the formula, the value of g is equal to the value of the gate labeled with g in the circuit

- The resulting CNF formula is consistent with the initial circuit: in any satisfying assignment of the formula, the value of g is equal to the value of the gate labeled with g in the circuit
- Therefore, the CNF formula is equisatisfiable to the circuit

- The resulting CNF formula is consistent with the initial circuit: in any satisfying assignment of the formula, the value of g is equal to the value of the gate labeled with g in the circuit
- Therefore, the CNF formula is equisatisfiable to the circuit
- The reduction takes polynomial time

Goal

Reduce every search problem to Circuit-SAT.

Goal

Reduce every search problem to Circuit-SAT.

- Let A be a search problem

Goal

Reduce every search problem to Circuit-SAT.

- Let A be a search problem
- We know that there exists an algorithm $\mathcal{C}$ that takes an instance I of A and a candidate solution S and checks whether S is a solution for I in time polynomial in $|I|$

Goal

Reduce every search problem to Circuit-SAT.

- Let A be a search problem
- We know that there exists an algorithm $\mathcal{C}$ that takes an instance I of A and a candidate solution S and checks whether S is a solution for I in time polynomial in $|I|$
- In particular, $|S|$ is polynomial in $|I|$

Turn an Algorithm into a Circuit

- Note that a computer is in fact a circuit (of constant size!) implemented on a chip

Turn an Algorithm into a Circuit

- Note that a computer is in fact a circuit (of constant size!) implemented on a chip
- Each step of the algorithm $\mathcal{C}(I, S)$ is performed by this computer's circuit

Turn an Algorithm into a Circuit

- Note that a computer is in fact a circuit (of constant size!) implemented on a chip
- Each step of the algorithm $\mathcal{C}(I, S)$ is performed by this computer's circuit
- This gives a circuit of size polynomial in $|I|$ that has input bits for I and S and outputs whether S is a solution for I (a separate circuit for each input length)

Reduction

To solve an instance I of the problem A:

- take a circuit corresponding to $\mathcal{C}(I, \cdot)$

Reduction

To solve an instance I of the problem A:

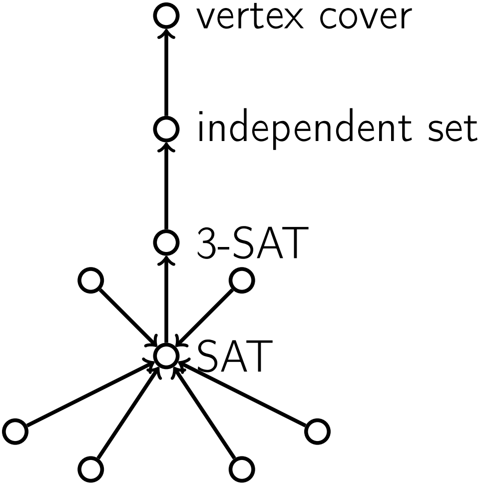
- take a circuit corresponding to $\mathcal{C}(I, \cdot)$
- the inputs to this circuit encode candidate solutions

Reduction

To solve an instance I of the problem A:

- take a circuit corresponding to $\mathcal{C}(I, \cdot)$
- the inputs to this circuit encode candidate solutions
- use a Circuit-SAT algorithm for this circuit to find a solution (if exists)

Summary



Outline

1. Reductions
2. Showing **NP**-completeness
3. Independent Set $\rightarrow$ Vertex Cover
4. 3-SAT $\rightarrow$ Independent Set
5. SAT $\rightarrow$ 3-SAT
6. All of **NP** $\rightarrow$ SAT
7. Using SAT-solvers

Sudoku Puzzle

This part

A simple and efficient Sudoku solver

SAT: Theory and Practice

Theory: we have no algorithm checking the satisfiability of a CNF formula F with n variables in time $\mathrm{poly}(|F|) \cdot 1.99^n$

SAT: Theory and Practice

Theory: we have no algorithm checking the satisfiability of a CNF formula F with n variables in time $\mathrm{poly}(|F|) \cdot 1.99^n$

Practice: SAT-solvers routinely solve instances with thousands of variables

Solving Hard Problems in Practice

An easy way to solve a hard combinatorial problem in practice:

- Reduce the problem to SAT (many problems are reduced to SAT in a natural way)

Solving Hard Problems in Practice

An easy way to solve a hard combinatorial problem in practice:

- Reduce the problem to SAT (many problems are reduced to SAT in a natural way)
- Use a SAT solver

Sudoku Puzzle

Goal: fill in with digits the partially completed 9×9 grid so that each row, each column, and each of the nine 3×3 subgrids contains all the digits from 1 to 9.

Example

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Example

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Variables

There will be $9 \times 9 \times 9 = 729$ Boolean variables: for $1 \leq i, j, k \leq 9$, $x_{ijk} = 1$, if and only if the cell $[i, j]$ contains the digit k

Exactly One Is True

Clauses expressing the fact that exactly one of the literals ℓ_1, ℓ_2, ℓ_3 is equal to 1:

$$(\ell_1 \lor \ell_2 \lor \ell_3)(\overline{\ell}_1 \lor \overline{\ell}_2)(\overline{\ell}_1 \lor \overline{\ell}_3)(\overline{\ell}_2 \lor \overline{\ell}_3)$$

Constraints

- Cell $[i, j]$ contains exactly one digit: $\texttt{ExactlyOneOf}(x_{ij1}, x_{ij2}, \ldots, x_{ij9})$

Constraints

- Cell $[i, j]$ contains exactly one digit: $\texttt{ExactlyOneOf}(x_{ij1}, x_{ij2}, \ldots, x_{ij9})$
- k appears exactly once in row i: $\texttt{ExactlyOneOf}(x_{i1k}, x_{i2k}, \ldots, x_{i9k})$

Constraints

- Cell $[i, j]$ contains exactly one digit:
  $\texttt{ExactlyOneOf}(x_{ij1}, x_{ij2}, \ldots, x_{ij9})$
- k appears exactly once in row i:
  $\texttt{ExactlyOneOf}(x_{i1k}, x_{i2k}, \ldots, x_{i9k})$
- k appears exactly once in column j:
  $\texttt{ExactlyOneOf}(x_{1jk}, x_{2jk}, \ldots, x_{9jk})$

Constraints

- Cell $[i, j]$ contains exactly one digit:
  $\texttt{ExactlyOneOf}(x_{ij1}, x_{ij2}, \ldots, x_{ij9})$
- k appears exactly once in row i:
  $\texttt{ExactlyOneOf}(x_{i1k}, x_{i2k}, \ldots, x_{i9k})$
- k appears exactly once in column j:
  $\texttt{ExactlyOneOf}(x_{1jk}, x_{2jk}, \ldots, x_{9jk})$
- k appears exactly once in a 3×3 block:
  $\texttt{ExactlyOneOf}(x_{11k}, x_{12k}, \ldots, x_{33k})$

Constraints

- Cell $[i, j]$ contains exactly one digit: $\texttt{ExactlyOneOf}(x_{ij1}, x_{ij2}, \ldots, x_{ij9})$
- k appears exactly once in row i: $\texttt{ExactlyOneOf}(x_{i1k}, x_{i2k}, \ldots, x_{i9k})$
- k appears exactly once in column j: $\texttt{ExactlyOneOf}(x_{1jk}, x_{2jk}, \ldots, x_{9jk})$
- k appears exactly once in a 3×3 block: $\texttt{ExactlyOneOf}(x_{11k}, x_{12k}, \ldots, x_{33k})$
- $[i, j]$ already contains k: (x_{ijk})

Resulting Formula

State-of-the-art SAT-solvers find a satisfying assignment for the resulting formula in blink of an eye, though the corresponding search space has size about $2^{729} \approx 10^{220}$