Contents

Problem Reductions: Models and Transformations

Technical Documentation

github.com/CodingThrust/problem-reductions

Abstract. We present formal definitions for computational problems and polynomial-time reductions implemented in the `problemreductions` library. For each reduction, we state theorems with constructive proofs that preserve solution structure.

1 Introduction

A *reduction* from problem A to problem B, denoted $A \longrightarrow B$, is a polynomial-time transformation of A-instances into B-instances such that: (1) the transformation runs in polynomial time, (2) solutions to B can be efficiently mapped back to solutions of A, and (3) optimal solutions are preserved. Figure 1 shows the 14 reductions connecting 33 problem types.

1.1 Notation

We use the following notation throughout. An *undirected graph* $G = (V, E)$ consists of a vertex set V and edge set $E \subseteq \binom{V}{2}$. For a set S, $\overline{S}$ or $V \setminus S$ denotes its complement. We write $|S|$ for cardinality. For Boolean variables, $\overline{x}$ denotes negation ($\neg x$). A *literal* is a variable x or its negation $\overline{x}$. A *clause* is a disjunction of literals. A formula in *conjunctive normal form* (CNF) is a conjunction of clauses. We abbreviate Independent Set as IS, Vertex Cover as VC, and use n for problem size, m for number of clauses, and $k_j = |C_j|$ for clause size.

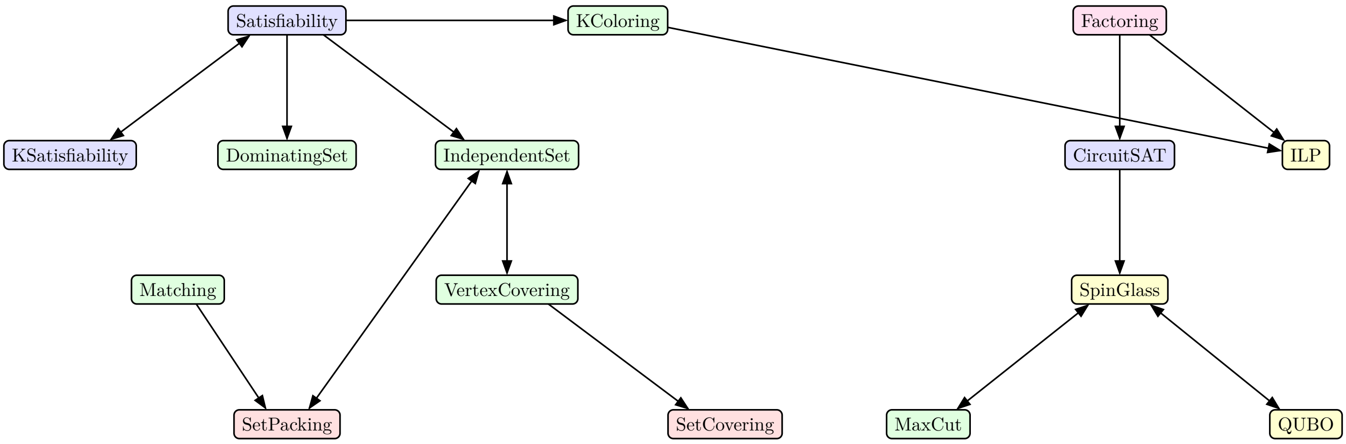


Figure 1: Reduction graph. Colors: green (graph), red (set), yellow (optimization), blue (satisfiability), pink (specialized).

2 Problem Definitions

2.1 Graph Problems

In all graph problems below, $G = (V, E)$ denotes an undirected graph with $|V| = n$ vertices and $|E|$ edges.

Definition 2.1 (Independent Set (IS)): Given $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ maximizing $\sum_{v \in S} w(v)$ such that no two vertices in S are adjacent: $\forall u, v \in S : (u, v) \notin E$.

Reduces to: Set Packing (Definition 2.8).

Reduces from: Vertex Cover (Definition 2.2), SAT (Definition 2.13), Set Packing (Definition 2.8).

```
pub struct IndependentSet<W = i32> {
    graph: UnGraph<(), ()>,  // The underlying graph
    weights: Vec<W>,         // Weights for each vertex
}

impl<W: 'static> Problem for IndependentSet<W> {
    const NAME: &'static str = "IndependentSet";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.2 (Vertex Cover (VC)): Given $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ minimizing $\sum_{v \in S} w(v)$ such that every edge has at least one endpoint in S: $\forall (u, v) \in E : u \in S \vee v \in S$.

Reduces to: Independent Set (Definition 2.1), Set Covering (Definition 2.9).

Reduces from: Independent Set (Definition 2.1).

```
pub struct VertexCovering<W = i32> {
    graph: UnGraph<(), ()>,  // The underlying graph
    weights: Vec<W>,         // Weights for each vertex
}

impl<W: 'static> Problem for VertexCovering<W> {
    const NAME: &'static str = "VertexCovering";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.3 (Max-Cut): Given $G = (V, E)$ with weights $w : E \to \mathbb{R}$, find partition $\left(S, \overline{S}\right)$ maximizing $\sum_{(u,v) \in E : u \in S, v \in \overline{S}} w(u, v)$.

Reduces to: Spin Glass (Definition 2.10).

Reduces from: Spin Glass (Definition 2.10).

```
pub struct MaxCut<W = i32> {
    graph: UnGraph<(), W>,  // Weighted graph (edge weights)
}

impl<W: 'static> Problem for MaxCut<W> {
    const NAME: &'static str = "MaxCut";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.4 (Graph Coloring): Given $G = (V, E)$ and k colors, find $c : V \to \{1, ..., k\}$ minimizing $|\{(u, v) \in E : c(u) = c(v)\}|$.

Reduces to: ILP (Definition 2.12).

Reduces from: SAT (Definition 2.13).

```
pub struct Coloring {
    num_colors: usize,       // Number of available colors (K)
    graph: UnGraph<(), ()>,  // The underlying graph
}

impl Problem for Coloring {
    const NAME: &'static str = "Coloring";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", "Unweighted")]
    }
    // ...
}
```

Definition 2.5 (Dominating Set): Given $G = (V, E)$ with weights $w : V \to \mathbb{R}$, find $S \subseteq V$ minimizing $\sum_{v \in S} w(v)$ s.t. $\forall v \in V : v \in S \vee \exists u \in S : (u, v) \in E$.

Reduces from: SAT (Definition 2.13).

```
pub struct DominatingSet<W = i32> {
    graph: UnGraph<(), ()>,  // The underlying graph
    weights: Vec<W>,         // Weights for each vertex
}

impl<W: 'static> Problem for DominatingSet<W> {
    const NAME: &'static str = "DominatingSet";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.6 (Matching): Given $G = (V, E)$ with weights $w : E \to \mathbb{R}$, find $M \subseteq E$ maximizing $\sum_{e \in M} w(e)$ s.t. $\forall e_1, e_2 \in M : e_1 \cap e_2 = \emptyset$.

Reduces to: Set Packing (Definition 2.8).

```
pub struct Matching<W = i32> {
    num_vertices: usize,     // Number of vertices
    graph: UnGraph<(), W>,   // Weighted graph
    edge_weights: Vec<W>,    // Weights for each edge
}

impl<W: 'static> Problem for Matching<W> {
    const NAME: &'static str = "Matching";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.7 (Unit Disk Graph (Grid Graph)): A graph $G = (V, E)$ where vertices V are points on a 2D lattice and $(u, v) \in E$ iff the Euclidean distance $d(u, v) \leq r$ for some radius r. A *King's subgraph* uses the King's graph lattice (8-connectivity square grid) with $r \approx 1.5$.

2.2 Set Problems

Definition 2.8 (Set Packing): Given universe U, collection $\mathcal{S} = \{S_1, ..., S_m\}$ with $S_i \subseteq U$, weights $w : \mathcal{S} \to \mathbb{R}$, find $\mathcal{P} \subseteq \mathcal{S}$ maximizing $\sum_{S \in \mathcal{P}} w(S)$ s.t. $\forall S_i, S_j \in \mathcal{P} : S_i \cap S_j = \emptyset$.

Reduces to: Independent Set (Definition 2.1).

Reduces from: Independent Set (Definition 2.1), Matching (Definition 2.6).

```
pub struct SetPacking<W = i32> {
    sets: Vec<Vec<usize>>,  // Collection of sets
    weights: Vec<W>,        // Weights for each set
}

impl<W: 'static> Problem for SetPacking<W> {
    const NAME: &'static str = "SetPacking";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.9 (Set Covering): Given universe U, collection $\mathcal{S}$ with weights $w : \mathcal{S} \to \mathbb{R}$, find $\mathcal{C} \subseteq \mathcal{S}$ minimizing $\sum_{S \in \mathcal{C}} w(S)$ s.t. $\bigcup_{S \in \mathcal{C}} S = U$.

Reduces from: Vertex Cover (Definition 2.2).

```
pub struct SetCovering<W = i32> {
    universe_size: usize,   // Size of the universe
    sets: Vec<Vec<usize>>,  // Collection of sets
    weights: Vec<W>,        // Weights for each set
}

impl<W: 'static> Problem for SetCovering<W> {
    const NAME: &'static str = "SetCovering";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

2.3 Optimization Problems

Definition 2.10 (Spin Glass (Ising Model)): Given n spin variables $s_i \in \{-1, +1\}$, pairwise couplings $J_{ij} \in \mathbb{R}$, and external fields $h_i \in \mathbb{R}$, minimize the Hamiltonian (energy function): $H(\boldsymbol{s}) = -\sum_{(i,j)} J_{ij} s_i s_j - \sum_i h_i s_i$.

Reduces to: Max-Cut (Definition 2.3), QUBO (Definition 2.11).

Reduces from: Circuit-SAT (Definition 2.15), Max-Cut (Definition 2.3), QUBO (Definition 2.11).

```
pub struct SpinGlass<W = f64> {
    num_spins: usize,                    // Number of spins
    interactions: Vec<((usize, usize), W)>,  // J_ij couplings
    fields: Vec<W>,                      // h_i on-site fields
}

impl<W: 'static> Problem for SpinGlass<W> {
    const NAME: &'static str = "SpinGlass";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.11 (QUBO): Given n binary variables $x_i \in \{0, 1\}$, matrix $Q \in \mathbb{R}^{n \times n}$, minimize $f(\boldsymbol{x}) = \boldsymbol{x}^\top Q \boldsymbol{x}$.

Reduces to: Spin Glass (Definition 2.10).

Reduces from: Spin Glass (Definition 2.10).

```
pub struct QUBO<W = f64> {
    num_vars: usize,          // Number of variables
    matrix: Vec<Vec<W>>,      // Q matrix (upper triangular)
}

impl<W: 'static> Problem for QUBO<W> {
    const NAME: &'static str = "QUBO";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.12 (Integer Linear Programming (ILP)): Given n integer variables $\boldsymbol{x} \in \mathbb{Z}^n$, constraint matrix $A \in \mathbb{R}^{m \times n}$, bounds $\boldsymbol{b} \in \mathbb{R}^m$, and objective $\boldsymbol{c} \in \mathbb{R}^n$, find $\boldsymbol{x}$ minimizing $\boldsymbol{c}^\top \boldsymbol{x}$ subject to $A\boldsymbol{x} \leq \boldsymbol{b}$ and variable bounds.

Reduces from: Graph Coloring (Definition 2.4), Factoring (Definition 2.16).

```
pub struct ILP {
    num_vars: usize,                  // Number of variables
    bounds: Vec<VarBounds>,           // Bounds per variable
    constraints: Vec<LinearConstraint>, // Linear constraints
    objective: Vec<(usize, f64)>,     // Sparse objective
    sense: ObjectiveSense,            // Maximize or Minimize
}

pub struct VarBounds { lower: Option<i64>, upper: Option<i64> }
pub struct LinearConstraint {
    terms: Vec<(usize, f64)>,  // (var_index, coefficient)
    cmp: Comparison,           // Le, Ge, or Eq
    rhs: f64,
}

impl Problem for ILP {
    const NAME: &'static str = "ILP";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", "f64")]
    }
    // ...
}
```

2.4 Satisfiability Problems

Definition 2.13 (SAT): Given a CNF formula $\varphi = \bigwedge_{j=1}^{m} C_j$ with m clauses over n Boolean variables, where each clause $C_j = \bigvee_i \ell_{ji}$ is a disjunction of literals, find an assignment $\boldsymbol{x} \in \{0,1\}^n$ such that $\varphi(\boldsymbol{x}) = 1$ (all clauses satisfied).

Reduces to: Independent Set (Definition 2.1), Graph Coloring (Definition 2.4), Dominating Set (Definition 2.5), k-SAT (Definition 2.14).

Reduces from: k-SAT (Definition 2.14).

```
pub struct Satisfiability<W = i32> {
    num_vars: usize,           // Number of variables
    clauses: Vec<CNFClause>,   // Clauses in CNF
    weights: Vec<W>,           // Weights per clause (MAX-SAT)
}

pub struct CNFClause {
    literals: Vec<i32>,  // Signed: +i for x_i, -i for NOT x_i
}

impl<W: 'static> Problem for Satisfiability<W> {
    const NAME: &'static str = "Satisfiability";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.14 (k-SAT): SAT with exactly k literals per clause.

Reduces to: SAT (Definition 2.13).

Reduces from: SAT (Definition 2.13).

```
pub struct KSatisfiability<const K: usize, W = i32> {
    num_vars: usize,           // Number of variables
    clauses: Vec<CNFClause>,   // Each clause has exactly K literals
    weights: Vec<W>,           // Weights per clause
}

impl<const K: usize, W: 'static> Problem for KSatisfiability<K, W> {
    const NAME: &'static str = "KSatisfiability";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.15 (Circuit-SAT): Given a Boolean circuit C composed of logic gates (AND, OR, NOT, XOR) with n input variables, find an input assignment $\boldsymbol{x} \in \{0,1\}^n$ such that $C(\boldsymbol{x}) = 1$.

Reduces to: Spin Glass (Definition 2.10).

Reduces from: Factoring (Definition 2.16).

```
pub struct CircuitSAT<W = i32> {
    circuit: Circuit,          // The boolean circuit
    variables: Vec<String>,    // Variable names in order
    weights: Vec<W>,           // Weights per assignment
}

pub struct Circuit { assignments: Vec<Assignment> }
pub struct Assignment { outputs: Vec<String>, expr: BooleanExpr }
pub enum BooleanOp { Var(String), Const(bool), Not(..), And(..), Or(..), Xor(..) }

impl<W: 'static> Problem for CircuitSAT<W> {
    const NAME: &'static str = "CircuitSAT";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", short_type_name::<W>())]
    }
    // ...
}
```

Definition 2.16 (Factoring): Given a composite integer N and bit sizes m, n, find integers $p \in [2, 2^m - 1]$ and $q \in [2, 2^n - 1]$ such that $p \times q = N$. Here p has m bits and q has n bits.

Reduces to: Circuit-SAT (Definition 2.15), ILP (Definition 2.12).

```
pub struct Factoring {
    m: usize,      // Bits for first factor
    n: usize,      // Bits for second factor
    target: u64,   // The number to factor
}

impl Problem for Factoring {
    const NAME: &'static str = "Factoring";
    fn variant() -> Vec<(&'static str, &'static str)> {
        vec![("graph", "SimpleGraph"), ("weight", "i32")]
    }
    // ...
}
```

3 Reductions

3.1 Trivial Reductions

Theorem: **(IS ↔ VC)** $S \subseteq V$ is independent iff $V \setminus S$ is a vertex cover, with $|\text{IS}| + |\text{VC}| = |V|$. [*Problems:* Definition 2.1, Definition 2.2.]

Proof: ($\Rightarrow$) If S is independent, for any $(u, v) \in E$, at most one endpoint lies in S, so $V \setminus S$ covers all edges. ($\Leftarrow$) If C is a cover, for any $u, v \in V \setminus C$, $(u, v) \notin E$, so $V \setminus C$ is independent. □

```
// Minimal example: IS -> VC -> extract solution
let is_problem = IndependentSet::<i32>::new(3, vec![(0, 1), (1, 2), (0, 2)]);
let result = ReduceTo::<VertexCovering<i32>>::reduce_to(&is_problem);
let vc_problem = result.target_problem();

let solver = BruteForce::new();
let vc_solutions = solver.find_best(vc_problem);
let is_solution = result.extract_solution(&vc_solutions[0]);
assert!(is_problem.solution_size(&is_solution).is_valid);
```

Theorem: **(IS → Set Packing)** Construct $U = E$, $S_v = \{e \in E : v \in e\}$, $w(S_v) = w(v)$. Then I is independent iff $\{S_v : v \in I\}$ is a packing. [*Problems:* Definition 2.1, Definition 2.8.]

Proof: Independence implies disjoint incident edge sets; conversely, disjoint edge sets imply no shared edges. □

```
// Minimal example: IS -> SetPacking -> extract solution
let is_problem = IndependentSet::<i32>::new(3, vec![(0, 1), (1, 2), (0, 2)]);
let result = ReduceTo::<SetPacking<i32>>::reduce_to(&is_problem);
let sp_problem = result.target_problem();

let solver = BruteForce::new();
let sp_solutions = solver.find_best(sp_problem);
let is_solution = result.extract_solution(&sp_solutions[0]);
assert!(is_problem.solution_size(&is_solution).is_valid);
```

Theorem: **(VC → Set Covering)** Construct $U = \{0, ..., |E| - 1\}$, $S_v = \{i : e_i \text{ incident to } v\}$, $w(S_v) = w(v)$. Then C is a cover iff $\{S_v : v \in C\}$ covers U. [*Problems:* Definition 2.2, Definition 2.9.]

Theorem: **(Matching → Set Packing)** Construct $U = V$, $S_e = \{u, v\}$ for $e = (u, v)$, $w(S_e) = w(e)$. Then M is a matching iff $\{S_e : e \in M\}$ is a packing. [*Problems:* Definition 2.6, Definition 2.8.]

Theorem: **(Spin Glass ↔ QUBO)** The substitution $s_i = 2x_i - 1$ yields $H_{\mathrm{SG}(\boldsymbol{s})} = H_{\mathrm{QUBO}(\boldsymbol{x})} + \mathrm{const}$. [*Problems:* Definition 2.10, Definition 2.11.]

Proof: Expanding $-\sum_{i,j} J_{ij}(2x_i - 1)(2x_j - 1) - \sum_i h_i(2x_i - 1)$ gives $Q_{ij} = -4J_{ij}$, $Q_{ii} = 2\sum_j J_{ij} - 2h_i$. □

```
// Minimal example: SpinGlass -> QUBO -> extract solution
let sg = SpinGlass::new(2, vec![((0, 1), -1.0)], vec![0.5, -0.5]);
let result = ReduceTo::<QUBO>::reduce_to(&sg);
let qubo = result.target_problem();

let solver = BruteForce::new();
let qubo_solutions = solver.find_best(qubo);
let sg_solution = result.extract_solution(&qubo_solutions[0]);
assert_eq!(sg_solution.len(), 2);
```

3.2 Non-Trivial Reductions

Theorem: **(SAT → IS)** [1] Given CNF φ with m clauses, construct graph G such that φ is satisfiable iff G has an IS of size m. [*Problems:* Definition 2.13, Definition 2.1.]

Proof: *Construction.* For $\varphi = \bigwedge_{j=1}^{m} C_j$ with $C_j = (\ell_{j,1} \lor \ldots \lor \ell_{j,k_j})$:

Vertices: For each literal $\ell_{j,i}$ in clause C_j, create $v_{j,i}$. Total: $|V| = \sum_j k_j$.

Edges: (1) Intra-clause cliques: $E_{\text{clause}} = \{(v_{j,i}, v_{j,i'}) : i \neq i'\}$. (2) Conflict edges: $E_{\text{conflict}} = \{(v_{j,i}, v_{j',i'}) : j \neq j', \ell_{j,i} = \overline{\ell_{j',i'}}\}$.

Correctness. (⇒) A satisfying assignment selects one true literal per clause; these vertices form an IS of size m (no clause edges by selection, no conflict edges by consistency). (⇐) An IS of size m must contain exactly one vertex per clause (by clause cliques); the corresponding literals are consistent (by conflict edges) and satisfy φ.

Solution extraction. For $v_{j,i} \in S$ with literal x_k: set $x_k = 1$; for $\overline{x_k}$: set $x_k = 0$. □

Theorem: **(SAT → 3-Coloring)** [2] Given CNF φ, construct graph G such that φ is satisfiable iff G is 3-colorable. [*Problems:* Definition 2.13, Definition 2.4.]

Proof: *Construction.* (1) Base triangle: TRUE, FALSE, AUX vertices with all pairs connected. (2) Variable gadget for x_i: vertices pos_i, neg_i connected to each other and to AUX. (3) Clause gadget: for $(\ell_1 \lor \ldots \lor \ell_k)$, apply OR-gadgets iteratively producing output o, then connect o to FALSE and AUX.

OR-gadget$(a, b) \mapsto o$*:* Five vertices encoding $o = a \lor b$: if both a, b have FALSE color, o cannot have TRUE color.

Solution extraction. Set $x_i = 1$ iff $\mathrm{color}(\mathrm{pos}_i) = \mathrm{color}(\mathrm{TRUE})$. □

Theorem: **(SAT → Dominating Set)** [2] Given CNF φ with n variables and m clauses, φ is satisfiable iff the constructed graph has a dominating set of size n. [*Problems:* Definition 2.13, Definition 2.5.]

Proof: *Construction.* (1) Variable triangle for x_i: vertices $\mathrm{pos}_i = 3i$, $\mathrm{neg}_i = 3i + 1$, $\mathrm{dum}_i = 3i + 2$ forming a triangle. (2) Clause vertex $c_j = 3n + j$ connected to pos_i if $x_i \in C_j$, to neg_i if $\overline{x_i} \in C_j$.

Correctness. Each triangle requires at least one vertex in any dominating set. Size-n set must take exactly one per triangle, which dominates clause vertices iff corresponding literals satisfy all clauses.

Solution extraction. Set $x_i = 1$ if pos_i selected; $x_i = 0$ if neg_i selected. □

Theorem: **(SAT ↔ *k*-SAT)** [2], [3] Any SAT formula converts to k-SAT ($k \geq 3$) preserving satisfiability. [*Problems:* Definition 2.13, Definition 2.14.]

Proof: *Small clauses ($|C| < k$):* Pad $(\ell_1 \lor \ldots \lor \ell_r)$ with auxiliary y: $(\ell_1 \lor \ldots \lor \ell_r \lor y \lor \overline{y} \lor \ldots)$ to length k.

Large clauses ($|C| > k$): Split $(\ell_1 \lor \ldots \lor \ell_r)$ with auxiliaries $y_1, \ldots, y_{r-k}$:

$$(\ell_1 \vee \ldots \vee \ell_{k-1} \vee y_1) \wedge (\overline{y_1} \vee \ell_k \vee \ldots \vee y_2) \wedge \ldots \wedge (\overline{y_{r-k}} \vee \ell_{r-k+2} \vee \ldots \vee \ell_r)$$

Correctness. Original clause true ↔ auxiliary chain can propagate truth through new clauses. □

Theorem: **(CircuitSAT → Spin Glass)** [4], [5] Each gate maps to a gadget whose ground states encode valid I/O. [*Problems:* Definition 2.15, Definition 2.10.]

Proof: *Spin mapping:* $\sigma \in \{0,1\} \mapsto s = 2\sigma - 1 \in \{-1,+1\}$.

Gate gadgets (inputs 0,1; output 2; auxiliary 3 for XOR) are shown in Table 1. Allocate spins per variable, instantiate gadgets, sum Hamiltonians. Ground states correspond to satisfying assignments. □

Gate	**Couplings J**	**Fields h**
AND	$J_{01} = 1, J_{02} = J_{12} = -2$	$h_0 = h_1 = -1, h_2 = 2$
OR	$J_{01} = 1, J_{02} = J_{12} = -2$	$h_0 = h_1 = 1, h_2 = -2$
NOT	$J_{01} = 1$	$h_0 = h_1 = 0$
XOR	$J_{01} = 1, J_{02} = J_{12} = -1, J_{03} = J_{13} = -2, J_{23} = 2$	$h_0 = h_1 = -1, h_2 = 1, h_3 = 2$

Table 1: Ising gadgets for logic gates. Ground states match truth tables.

Theorem: **(Factoring → Circuit-SAT)** An array multiplier with output constrained to N is satisfiable iff N factors within bit bounds. *(Folklore; no canonical reference.)* [*Problems:* Definition 2.16, Definition 2.15.]

Proof: *Construction.* Build $m \times n$ array multiplier for $p \times q$:

Full adder (i,j): $s_{i,j} + 2c_{i,j} = (p_i \wedge q_j) + s_{\text{prev}} + c_{\text{prev}}$ via:

$$a := p_i \wedge q_j, \quad t_1 := a \oplus s_{\text{prev}}, \quad s_{i,j} := t_1 \oplus c_{\text{prev}}$$

$$t_2 := t_1 \wedge c_{\text{prev}}, \quad t_3 := a \wedge s_{\text{prev}}, \quad c_{i,j} := t_2 \vee t_3$$

Output constraint: $M_k := \text{bit}_{k(N)}$ for $k = 1, \ldots, m+n$.

Solution extraction. $p = \sum_i p_i 2^{i-1}$, $q = \sum_j q_j 2^{j-1}$. □

Theorem: **(Spin Glass ↔ Max-Cut)** [5], [6] Ground states of Ising models correspond to maximum cuts. [*Problems:* Definition 2.10, Definition 2.3.]

Proof: *MaxCut → SpinGlass:* Set $J_{ij} = w_{ij}$, $h_i = 0$. Maximizing cut equals minimizing $-\sum J_{ij} s_i s_j$ since $s_i s_j = -1$ when $s_i \neq s_j$.

SpinGlass → MaxCut: If $h_i = 0$: direct mapping $w_{ij} = J_{ij}$. Otherwise, add ancilla a with $w_{i,a} = h_i$.

Solution extraction. Without ancilla: identity. With ancilla: if $\sigma_a = 1$, flip all spins before removing ancilla. □

```rust
// Minimal example: SpinGlass -> MaxCut -> extract solution
let sg = SpinGlass::new(3, vec![((0, 1), 1), ((1, 2), 1), ((0, 2), 1)], vec![0, 0, 0]);
let result = ReduceTo::<MaxCut<i32>>::reduce_to(&sg);
let maxcut = result.target_problem();

let solver = BruteForce::new();
let maxcut_solutions = solver.find_best(maxcut);
let sg_solution = result.extract_solution(&maxcut_solutions[0]);
assert_eq!(sg_solution.len(), 3);
```

Theorem: **(Coloring → ILP)** The k-coloring problem reduces to binary ILP with $|V| \cdot k$ variables and $|V| + |E| \cdot k$ constraints. [*Problems:* Definition 2.4, Definition 2.12.]

Proof: *Construction.* For graph $G = (V, E)$ with k colors:

Variables: Binary $x_{v,c} \in \{0,1\}$ for each vertex $v \in V$ and color $c \in \{1, \ldots, k\}$. Interpretation: $x_{v,c} = 1$ iff vertex v has color c.

Constraints: (1) Each vertex has exactly one color: $\sum_{c=1}^{k} x_{v,c} = 1$ for all $v \in V$. (2) Adjacent vertices have different colors: $x_{u,c} + x_{v,c} \leq 1$ for all $(u, v) \in E$ and $c \in \{1, ..., k\}$.

Objective: Feasibility problem (minimize 0).

Correctness. (⇒) A valid k-coloring assigns exactly one color per vertex with different colors on adjacent vertices; setting $x_{v,c} = 1$ for the assigned color satisfies all constraints. (⇐) Any feasible ILP solution has exactly one $x_{v,c} = 1$ per vertex; this defines a coloring, and constraint (2) ensures adjacent vertices differ.

Solution extraction. For each vertex v, find c with $x_{v,c} = 1$; assign color c to v. □

Theorem: **(Factoring → ILP)** Integer factorization reduces to binary ILP using McCormick linearization with $O(mn)$ variables and constraints. [*Problems:* Definition 2.16, Definition 2.12.]

Proof: *Construction.* For target N with m-bit factor p and n-bit factor q:

Variables: Binary $p_i, q_j \in \{0, 1\}$ for factor bits; binary $z_{ij} \in \{0, 1\}$ for products $p_i \cdot q_j$; integer $c_k \geq 0$ for carries at each bit position.

Product linearization (McCormick): For each $z_{ij} = p_i \cdot q_j$:

$$z_{ij} \leq p_i, \quad z_{ij} \leq q_j, \quad z_{ij} \geq p_i + q_j - 1$$

Bit-position equations: For each bit position k:

$$\sum_{i+j=k} z_{ij} + c_{k-1} = N_k + 2c_k$$

where N_k is the k-th bit of N and $c_{-1} = 0$.

No overflow: $c_{m+n-1} = 0$.

Correctness. The McCormick constraints enforce $z_{ij} = p_i \cdot q_j$ for binary variables. The bit equations encode $p \times q = N$ via carry propagation, matching array multiplier semantics.

Solution extraction. Read $p = \sum_i p_i 2^i$ and $q = \sum_j q_j 2^j$ from the binary variables. □

Example: Factoring 15. The following Rust code demonstrates the closed-loop reduction (requires `ilp` feature: `cargo add problemreductions --features ilp`):

```rust
use problemreductions::prelude::*;

// 1. Create factoring instance: find p (4-bit) × q (4-bit) = 15
let problem = Factoring::new(4, 4, 15);

// 2. Reduce to ILP
let reduction = ReduceTo::<ILP>::reduce_to(&problem);
let ilp = reduction.target_problem();

// 3. Solve ILP
let solver = ILPSolver::new();
let ilp_solution = solver.solve(ilp).unwrap();

// 4. Extract factoring solution
let extracted = reduction.extract_solution(&ilp_solution);

// 5. Verify: reads factors and confirms p × q = 15
let (p, q) = problem.read_factors(&extracted);
assert_eq!(p * q, 15); // e.g., (3, 5) or (5, 3)
```

3.3 Unit Disk Mapping

Theorem: **(IS → GridGraph IS)** [7] Any MIS problem on a general graph G can be reduced to MIS on a unit disk graph (King's subgraph) with at most quadratic overhead in the number of vertices. [*Problem:* Definition 2.1.]

Proof: *Construction (Copy-Line Method).* Given $G = (V, E)$ with $n = |V|$:

1. *Vertex ordering:* Compute a path decomposition of G to obtain vertex order $(v_1, ..., v_n)$. The pathwidth determines the grid height.
2. *Copy lines:* For each vertex v_i, create an L-shaped "copy line" on the grid:

$$\text{CopyLine}(v_i) = \{(r, c_i) : r \in [r_{\text{start}}, r_{\text{stop}}]\} \cup \{(r_i, c) : c \in [c_i, c_{\text{stop}}]\}$$

where positions are determined by the vertex order and edge structure.

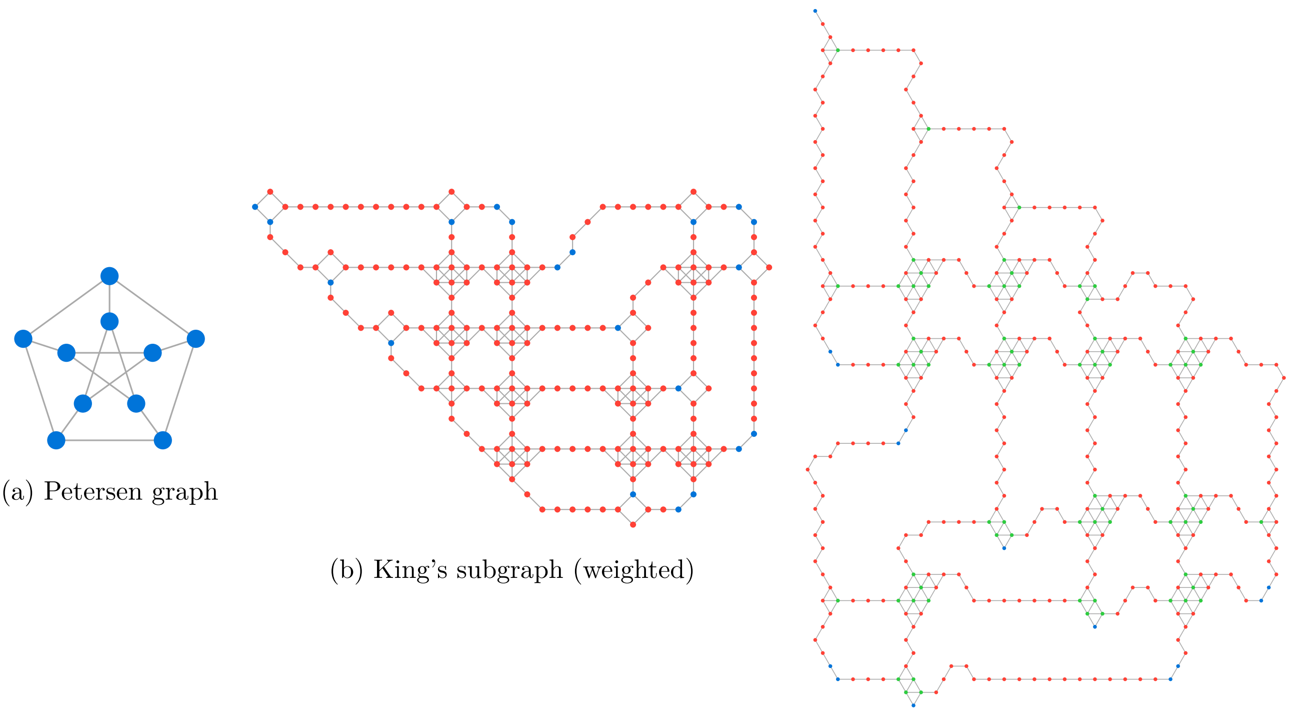
3. *Crossing gadgets:* When two copy lines cross (corresponding to an edge $(v_i, v_j) \in E$), insert a crossing gadget that enforces: at most one of the two lines can be "active" (all vertices selected).
4. *MIS correspondence:* Each copy line has MIS contribution $\approx |\text{line}|/2$. The gadgets add overhead Δ such that:

$$\text{MIS}(G_{\text{grid}}) = \text{MIS}(G) + \Delta$$

Solution extraction. For each copy line, check if the majority of its vertices are in the grid MIS. Map back: $v_i \in S$ iff copy line i is active.

Correctness. ($\Rightarrow$) An IS in G maps to selecting all copy line vertices for included vertices; crossing gadgets ensure no conflicts. ($\Leftarrow$) A grid MIS maps back to an IS by the copy line activity rule. □

Example: Petersen Graph.[1] The Petersen graph ($n = 10$, MIS$= 4$) maps to a 30×42 King's subgraph with 219 nodes and overhead $\Delta = 89$. Solving MIS on the grid yields $\text{MIS}(G_{\text{grid}}) = 4 + 89 = 93$. The weighted and unweighted KSG mappings share identical grid topology (same node positions and edges); only the vertex weights differ. With triangular lattice encoding [7], the same graph maps to a 42×60 grid with 395 nodes and overhead $\Delta = 375$, giving $\text{MIS}(G_{\text{tri}}) = 4 + 375 = 379$.



(a) Petersen graph

(b) King's subgraph (weighted)

(c) Triangular lattice (weighted)

Figure 2: Unit disk mappings of the Petersen graph. Blue: weight 1, red: weight 2, green: weight 3.

Weighted Extension. For MWIS, copy lines use weighted vertices (weights 1, 2, or 3). Source weights < 1 are added to designated "pin" vertices.

[1] Generated using `cargo run --example export_petersen_mapping` from the accompanying code repository.

QUBO Mapping. A QUBO problem $\min \boldsymbol{x}^\top Q \boldsymbol{x}$ maps to weighted MIS on a grid by:

1. Creating copy lines for each variable
2. Using XOR gadgets for couplings: $x_{\text{out}} = \neg(x_1 \oplus x_2)$
3. Adding weights for linear and quadratic terms

4 Summary

Reduction	**Overhead**	**Reference**
IS $\leftrightarrow$ VC	$O(\|V\|)$	—
IS $\rightarrow$ SetPacking	$O(\|V\| + \|E\|)$	—
Matching $\rightarrow$ SetPacking	$O(\|E\|)$	—
VC $\rightarrow$ SetCovering	$O(\|V\| + \|E\|)$	—
QUBO $\leftrightarrow$ SpinGlass	$O(n^2)$	—
SAT $\rightarrow$ IS	$O\left(\sum_j \|C_j\|^2\right)$	[1]
SAT $\rightarrow$ 3-Coloring	$O\left(n + \sum_j \|C_j\|\right)$	[2]
SAT $\rightarrow$ DominatingSet	$O(3n + m)$	[2]
SAT $\leftrightarrow$ k-SAT	$O\left(\sum_j \|C_j\|\right)$	[2], [3]
CircuitSAT $\rightarrow$ SpinGlass	$O(\|\text{gates}\|)$	[4], [5]
Factoring $\rightarrow$ CircuitSAT	$O(mn)$	Folklore
SpinGlass $\leftrightarrow$ MaxCut	$O(n + \|J\|)$	[5], [6]
Coloring $\rightarrow$ ILP	$O(\|V\| \cdot k + \|E\| \cdot k)$	—
Factoring $\rightarrow$ ILP	$O(mn)$	—
IS $\rightarrow$ GridGraph IS	$O(n^2)$	[7]

Table 2: Summary of reductions. Gray rows indicate trivial reductions.

Bibliography

[1] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, Plenum Press, 1972, pp. 85–103.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[3] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[4] J. D. Whitfield, M. Faccin, and J. D. Biamonte, "Ground-state spin logic," *EPL (Europhysics Letters)*, vol. 99, no. 5, p. 57004, 2012.

[5] A. Lucas, "Ising formulations of many NP problems," *Frontiers in Physics*, vol. 2, no. 5, 2014.

[6] F. Barahona, "On the computational complexity of Ising spin glass models," *Journal of Physics A: Mathematical and General*, vol. 15, no. 10, pp. 3241–3253, 1982.

[7] M.-T. Nguyen, J.-G. Liu, J. Wurtz, M. D. Lukin, S.-T. Wang, and H. Pichler, "Quantum Optimization with Arbitrary Connectivity Using Rydberg Atom Arrays," *PRX Quantum*, vol. 4, p. 10316, 2023, doi: 10.1103/PRXQuantum.4.010316.