PRD 17: Isomer Enumeration via Molecular Graph Theory

Pure Thought AI Challenge 17

Pure Thought AI Challenges Project

January 18, 2026

Abstract

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

Contents

Domain: Chemistry Combinatorics
Timeline: 4-6 months
Difficulty: Medium-High
Prerequisites: Graph theory, group theory (Pólya enumeration), combinatorial optimization, SAT solving

---

0.1 1. Problem Statement

0.1.1 Scientific Context

Isomers are molecules with the same chemical formula but different atomic arrangements. Enumerating all isomers for a given formula is a fundamental problem in chemistry:

Types of Isomers:

- **Structural isomers**: Different connectivity graphs (e.g., butane vs isobutane: both CH)
- **Stereoisomers**: Same connectivity, different 3D spatial arrangement
- **Conformers**: Same structure, different rotations around single bonds

Challenges:

- Number of isomers grows exponentially with molecular size
- Chemical valence rules constrain graphs (C forms 4 bonds, O forms 2, H forms 1)
- Symmetry: many graphs are equivalent up to atom relabeling (automorphisms)

Current Methods:

- **Brute force**: Generate all graphs, filter by valence—combinatorial explosion
- **Chemical databases**: Enumerate known structures—incomplete for large molecules
- **SMILES enumeration**: String-based, but misses many structures

Pure Thought Approach:

- Use **P 19th century mathematician György Pólya's enumeration theorem** to count distinct graphs
- Generate isomers systematically using **canonical labeling** (avoids duplicates)
- Apply chemical constraints as **SAT/SMT problems**
- Certify completeness: prove all isomers found

0.1.2 Core Question

**Can we enumerate ALL isomers for a molecular formula Cx Hy O$_z$...*usingONLY graphtheoryando*
Specifically:

- **Given formula (e.g., CHO), generate all structurally distinct molecular graphs**
- **Apply valence constraints (C: 4, O: 2, N: 3, H: 1)**
- **Remove duplicate graphs via canonical labeling (nauty algorithm)**
- **Verify completeness: prove no isomers missed**
- **Extend to stereoisomers (chirality, E/Z isomerism)**
- **Export as SMILES strings + 3D geometries**

0.1.3 Why This Matters

Theoretical Impact:

- **Connects pure combinatorics to molecular chemistry**
- **Provides exact enumeration (no sampling or approximation)**
- **Algorithmic chemistry: automated structure generation**

Practical Benefits:

- **Drug discovery: enumerate all possible drug candidates with formula**
- **Materials design: explore chemical space systematically**
- **Retrosynthesis: identify alternative synthetic routes**

Pure Thought Advantages:

- **Valence rules are purely graph-theoretic**
- **Pólya theory provides exact counts**
- **No experimental data needed**
- **Certificates of completeness via SAT solvers**

0.2 2. Mathematical Formulation

0.2.1 Problem Definition

A molecular graph is G = (V, E) where:

- **V = vertices (atoms)**
- **E = edges (bonds), with multiplicities (single, double, triple)**

Valence constraint: Each atom v has degree deg(v) = valence(element(v))

- **C: deg = 4**
- **O: deg = 2**
- **N: deg = 3**
- **H: deg = 1**

Counting bond multiplicity:

```
deg(v) =  _ {u     neighbors(v)} bond_order(v, u)
```

Isomer Enumeration Problem:

```
Input: Molecular formula {n_C carbons, n_H hydrogens, n_O oxygens, ...}
Output: Set S of non-isomorphic connected molecular graphs satisfying
   valence constraints
```

Isomorphism: Two graphs G, G are isomorphic if there exists bijection : V → V preserving adjacency and atom types.
Certificate: For each isomer G S:

- **Valence check: v, deg(v) = valence(v)**
- **Canonicity: G is in canonical form (no other isomorphic graph generated)**
- **Completeness proof: All graphs in S are non-isomorphic + no missing isomers**

0.2.2 Pólya Enumeration Theorem

For counting up to symmetry:

```
N = (1/|G|)  _ {g     G} cycle_index(g)
```

where G is the symmetry group, and cycle$_{i}ndexcountsfixedpointsundereachsymmetry.$

0.2.3 Input/Output Specification

Input:

```
from typing import Dict
import networkx as nx

class MolecularFormula:
    elements: Dict[str, int]  # {'C': 4, 'H': 10} for    C   H

    # Optional constraints
    allow_double_bonds: bool = True
    allow_triple_bonds: bool = False
    allow_rings: bool = True
    max_ring_size: int = 8
```

Output:

```python
class IsomerCertificate:
    formula: MolecularFormula

    # Enumerated isomers
    isomers: List[nx.Graph]  # List of molecular graphs
    num_isomers: int

    # Canonical representations
    canonical_smiles: List[str]  # SMILES strings
    adjacency_matrices: List[np.ndarray]

    # Verification
    p lya_count: int  # Theoretical count from P lya theorem
    completeness_proof: str  # SAT/SMT certificate

    # Statistics
    num_with_rings: int
    num_with_double_bonds: int
    degree_distribution: Dict[int, int]

    # Export
    mol_files: List[Path]  # 3D structures (.mol, .xyz)
    smiles_file: Path
```

0.3 3. Implementation Approach

0.3.1 Phase 1: Simple Enumeration for Small Molecules (Month 1)

Implement brute-force for validation:

```python
import networkx as nx
from itertools import combinations
from typing import List

VALENCES = {'C': 4, 'H': 1, 'O': 2, 'N': 3, 'S': 2, 'P': 3, 'F': 1,
    'Cl': 1}

def generate_all_graphs_brute_force(formula: MolecularFormula) ->
    List[nx.Graph]:
    """
    Brute force: try all possible connectivity patterns.

    Only works for very small molecules (    10 atoms).
    """
    # Create vertex list
    atoms = []
    for elem, count in formula.elements.items():
        atoms.extend([elem] * count)

    n = len(atoms)

    # All possible edge sets (choose subset of n(n-1)/2 possible edges)
    max_edges = n * (n-1) // 2
```

```
    valid_graphs = []

    # Iterate over all possible edge sets
    all_possible_edges = list(combinations(range(n), 2))

    for num_edges in range(n-1, max_edges+1):  # At least n-1 for
        connectivity
        for edge_set in combinations(all_possible_edges, num_edges):
            G = nx.Graph()
            G.add_nodes_from(range(n))

            # Assign atom types
            for i, atom in enumerate(atoms):
                G.nodes[i]['element'] = atom

            # Add edges (all single bonds for now)
            for (u, v) in edge_set:
                G.add_edge(u, v, bond_order=1)

            # Check valence
            if satisfies_valence(G) and nx.is_connected(G):
                valid_graphs.append(G)

    # Remove isomorphic duplicates
    unique_graphs = remove_isomorphic_duplicates(valid_graphs)

    return unique_graphs

def satisfies_valence(G: nx.Graph) -> bool:
    """Check if all atoms satisfy valence constraints."""
    for node in G.nodes:
        elem = G.nodes[node]['element']
        required_valence = VALENCES[elem]

        # Degree = sum of bond orders
        degree = sum(G[node][nbr].get('bond_order', 1) for nbr in
            G.neighbors(node))

        if degree != required_valence:
            return False

    return True

def remove_isomorphic_duplicates(graphs: List[nx.Graph]) ->
    List[nx.Graph]:
    """
    Remove isomorphic graphs using nauty canonical labeling.
    """
    from pynauty import Graph as PynautyGraph, certificate

    unique = []
    seen_certificates = set()

    for G in graphs:
        # Convert to pynauty format
        cert = compute_canonical_certificate(G)
```

```
        if cert not in seen_certificates:
            seen_certificates.add(cert)
            unique.append(G)

    return unique
```

Validation: Enumerate CH—should find 2 isomers (butane, isobutane).

0.3.2 Phase 2: Canonical Labeling with nauty (Months 1-2)

Use nauty algorithm for efficient isomorphism checking:

```
from pynauty import Graph as PynautyGraph, autgrp, certificate

def canonical_label_molecular_graph(G: nx.Graph) -> str:
    """
    Compute canonical labeling using nauty.

    Returns string certificate uniquely identifying isomorphism class.
    """
    n = len(G.nodes)

    # Partition vertices by atom type (nauty requires integer colors)
    elem_to_color = {elem: i for i, elem in
       enumerate(set(VALENCES.keys()))}

    coloring = [elem_to_color[G.nodes[v]['element']] for v in range(n)]

    # Convert to pynauty format
    adjacency = {v: list(G.neighbors(v)) for v in G.nodes}

    pynauty_graph = PynautyGraph(
        number_of_vertices=n,
        directed=False,
        adjacency_dict=adjacency,
        vertex_coloring=[coloring]
    )

    # Compute canonical certificate
    cert = certificate(pynauty_graph)

    return str(cert)

def is_canonical(G: nx.Graph) -> bool:
    """
    Check if graph is in canonical form.

    Canonical form: relabeling that is lexicographically smallest.
    """
    cert_original = canonical_label_molecular_graph(G)

    # Try all permutations (expensive just for validation)
    import itertools

    n = len(G.nodes)
```

```
    for perm in itertools.permutations(range(n)):
        G_perm = relabel_graph(G, perm)
        cert_perm = canonical_label_molecular_graph(G_perm)

        if cert_perm < cert_original:
            return False  # Found smaller labeling

    return True
```

0.3.3 Phase 3: Systematic Graph Generation (Months 2-4)

Use orderly generation (McKay's algorithm):

```
def orderly_generation(formula: MolecularFormula) -> List[nx.Graph]:
    """
    Generate graphs in canonical (orderly) manner.

    Avoids generating isomorphic duplicates.

    Based on McKay's orderly generation algorithm.
    """
    atoms = expand_formula(formula)  # ['C', 'C', 'C', 'C', 'H', 'H',
        ...]
    n = len(atoms)

    isomers = []

    # Start with empty graph
    G_init = nx.Graph()
    G_init.add_nodes_from(range(n))
    for i, elem in enumerate(atoms):
        G_init.nodes[i]['element'] = elem

    # Recursively add edges in canonical order
    def generate_recursive(G, edge_candidates):
        # Check if valid molecular graph
        if is_complete_and_valid(G):
            # Check canonical
            if is_canonical_under_automorphism(G):
                isomers.append(G.copy())
            return

        # Pruning: stop if overvalent
        if has_overvalent_atom(G):
            return

        # Add next edge (in canonical order)
        for (u, v) in edge_candidates:
            if can_add_edge(G, u, v):
                G_new = G.copy()
                G_new.add_edge(u, v, bond_order=1)

                # Recursively expand
                remaining_candidates = [(i, j) for (i, j) in
                    edge_candidates
                                        if (i, j) > (u, v)]
```

```
                generate_recursive(G_new, remaining_candidates)

    # All possible edges
    edge_candidates = list(combinations(range(n), 2))
    generate_recursive(G_init, edge_candidates)

    return isomers

def is_complete_and_valid(G: nx.Graph) -> bool:
    """
    Check if graph is a complete valid molecule.

    - All atoms satisfy valence
    - Graph is connected
    """
    if not nx.is_connected(G):
        return False

    for node in G.nodes:
        elem = G.nodes[node]['element']
        degree = G.degree(node)

        if degree != VALENCES[elem]:
            return False

    return True
```

0.3.4 Phase 4: Pólya Enumeration (Months 4-5)

Count isomers using Pólya's theorem:

```
from sympy import symbols, expand, Poly
from sympy.combinatorics import PermutationGroup, Permutation

def polya_count_isomers(formula: MolecularFormula) -> int:
    """
    Use P lya enumeration theorem to count non-isomorphic graphs.

    This gives theoretical  c o u n t doesn't enumerate structures.
    """
    atoms = expand_formula(formula)
    n = len(atoms)

    # Symmetry group: permutations preserving atom types
    # E.g., for    C   H   : permutations of 4 C's    permutations of
       10 H's

    C_indices = [i for i, a in enumerate(atoms) if a == 'C']
    H_indices = [i for i, a in enumerate(atoms) if a == 'H']

    # Generate symmetric group on each atom type
    perms_C = PermutationGroup([Permutation(C_indices)])
    perms_H = PermutationGroup([Permutation(H_indices)])

    # Full group: product
    G = combine_permutation_groups(perms_C, perms_H)
```

```
25
26      # Cycle index polynomial
27      x = symbols(f'x0:{n*(n-1)//2}')  # Variables for each edge
28
29      cycle_poly = compute_cycle_index(G, x)
30
31      # Substitute x_i     1 + t (count graphs with/without each edge)
32      t = symbols('t')
33      cycle_poly_sub = cycle_poly.subs({xi: 1+t for xi in x})
34
35      # Extract coefficient of t^m where m = number of edges
36      # For tree: m = n-1
37      # For general graphs with cycles: various m
38
39      poly_expanded = expand(cycle_poly_sub)
40      coeffs = Poly(poly_expanded, t).all_coeffs()
41
42      # Number of isomers with m edges
43      isomer_counts = {m: coeffs[m] for m in range(len(coeffs))}
44
45      # Filter for chemically valid (satisfies valence)
46      # This is approximate exact filtering requires enumeration
47
48      total_isomers = sum(isomer_counts.values())
49
50      return total_isomers
51
52  def compute_cycle_index(G: PermutationGroup, variables: List) -> Poly:
53      """
54      Compute cycle index polynomial for permutation group G.
55
56      Z(G) = (1/|G|)  _ {g     G}  _i  x_i^{c_i(g)}
57
58      where c_i(g) = number of i-cycles in permutation g.
59      """
60      from sympy import Rational
61
62      cycle_poly = 0
63
64      for g in G.generate():
65          # Cycle structure of permutation g
66          cycles = g.cyclic_form
67
68          # Product over cycle lengths
69          term = 1
70          for cycle in cycles:
71              cycle_len = len(cycle)
72              term *= variables[cycle_len - 1]
73
74          cycle_poly += term
75
76      cycle_poly /= len(list(G.generate()))
77
78      return cycle_poly
```

0.3.5 Phase 5: Stereoisomer Enumeration (Months 5-6)

Extend to 3D stereochemistry:

```python
def enumerate_stereoisomers(molecular_graph: nx.Graph) -> List:
    """
    For each structural isomer, enumerate stereoisomers.

    - Chiral centers: tetrahedral carbons with 4 different substituents
    - E/Z isomers: double bonds with different substituents
    - Conformers: rotations around single bonds (separate problem)
    """
    stereoisomers = []

    # Find chiral centers
    chiral_centers = find_chiral_carbons(molecular_graph)

    # 2^n stereoisomers for n chiral centers (R/S configurations)
    for config in itertools.product(['R', 'S'],
        repeat=len(chiral_centers)):
        G_stereo = molecular_graph.copy()

        for center, chirality in zip(chiral_centers, config):
            G_stereo.nodes[center]['chirality'] = chirality

        stereoisomers.append(G_stereo)

    # E/Z isomers (double bonds)
    double_bonds = [(u, v) for u, v in molecular_graph.edges
                    if molecular_graph[u][v]['bond_order'] == 2]

    for bond in double_bonds:
        # Check if E/Z isomerism possible
        if has_EZ_isomerism(molecular_graph, bond):
            # Generate both E and Z forms
            # ... (geometric isomer generation)
            pass

    return stereoisomers

def find_chiral_carbons(G: nx.Graph) -> List[int]:
    """
    Identify chiral centers (sp  carbons with 4 different groups).
    """
    chiral = []

    for node in G.nodes:
        if G.nodes[node]['element'] != 'C':
            continue

        if G.degree(node) != 4:
            continue  # Must be tetrahedral

        # Check if 4 neighbors are distinct
        neighbors = list(G.neighbors(node))
        if all_distinct_substituents(G, neighbors):
            chiral.append(node)
```

```
    return chiral
```

0.3.6 Phase 6: Export and Validation (Month 6)

Generate output formats (SMILES, 3D structures):

```
from rdkit import Chem
from rdkit.Chem import AllChem

def export_isomers(isomers: List[nx.Graph], output_dir: Path):
    """
    Export isomers as SMILES and 3D structures.
    """
    smiles_list = []

    for i, G in enumerate(isomers):
        # Convert to RDKit molecule
        mol = nx_graph_to_rdkit(G)

        # Generate SMILES
        smiles = Chem.MolToSmiles(mol)
        smiles_list.append(smiles)

        # Generate 3D coordinates (force field optimization)
        mol_3d = Chem.AddHs(mol)
        AllChem.EmbedMolecule(mol_3d)
        AllChem.UFFOptimizeMolecule(mol_3d)

        # Save as MOL file
        Chem.MolToMolFile(mol_3d, str(output_dir /
            f'isomer_{i:03d}.mol'))

    # Save all SMILES
    with open(output_dir / 'isomers.smi', 'w') as f:
        for smi in smiles_list:
            f.write(smi + '\n')

def nx_graph_to_rdkit(G: nx.Graph) -> Chem.Mol:
    """
    Convert NetworkX molecular graph to RDKit Mol object.
    """
    mol = Chem.RWMol()

    # Add atoms
    node_to_idx = {}
    for node in G.nodes:
        elem = G.nodes[node]['element']
        atom = Chem.Atom(elem)
        idx = mol.AddAtom(atom)
        node_to_idx[node] = idx

    # Add bonds
    for u, v in G.edges:
        bond_order = G[u][v].get('bond_order', 1)

```

```
        if bond_order == 1:
            bond_type = Chem.BondType.SINGLE
        elif bond_order == 2:
            bond_type = Chem.BondType.DOUBLE
        elif bond_order == 3:
            bond_type = Chem.BondType.TRIPLE

        mol.AddBond(node_to_idx[u], node_to_idx[v], bond_type)

    return mol.GetMol()
```

0.4 4. Example Starting Prompt

```
You are a computational chemist implementing isomer enumeration via graph theory. Generate ALL
structural isomers for molecular formulas using ONLY combinatorial algorithmsno databases.

OBJECTIVE: Enumerate all C HO isomers, verify completeness, export as SMILES.

PHASE 1 (Month 1): Brute force baseline
- Implement basic graph generation for C H
- Apply valence constraints (C:4, H:1, O:2)
- Remove duplicates using nauty canonical labeling
- Verify: find exactly 2 isomers (butane, isobutane)

PHASE 2 (Months 1-2): Canonical labeling
- Implement nauty algorithm for molecular graphs
- Color vertices by element type
- Test isomorphism detection on 100 random graph pairs

PHASE 3 (Months 2-4): Orderly generation
- Implement McKay's orderly algorithm
- Generate graphs in canonical order (avoids duplicates)
- Test on C H : should find 3 isomers

PHASE 4 (Months 4-5): P lya enumeration
- Compute symmetry group for C HO
- Calculate cycle index polynomial
- Compare P lya count to generated count (must match!)

PHASE 5 (Months 5-6): Stereoisomers
- Identify chiral centers in each structural isomer
- Enumerate R/S configurations
- Handle E/Z isomerism for double bonds

PHASE 6 (Month 6): Export and validation
- Convert all isomers to SMILES strings
- Generate 3D geometries using RDKit
- Cross-check against PubChem database (for validation only)

SUCCESS CRITERIA:
```

```
- MVR:    C   H    and    C   H    correctly enumerated
- Strong:    C   HO    complete enumeration, all unique structures
- Publication: Systematic study up to  C  , comparison to P lya counts

VERIFICATION:
- Generated count matches P lya theoretical count
- All SMILES strings valid (parseable by RDKit)
- No duplicate structures (canonical checking)
- Cross-reference with PubChem (should find all known isomers)

Pure graph theory + combinatorics. No chemical databases until final
   validation.
All results certificate-based with completeness proofs.
```

0.5 5. Success Criteria

MVR (2 months): CH, CH correct, nauty working
Strong (4-5 months): CHO complete, Pólya counts verified
Publication (6 months): Systematic enumeration up to C, stereoisomers included

---

0.6 6. Verification Protocol

- **Compare generated counts to Pólya theoretical values**
- **Cross-check SMILES against PubChem**
- **Validate 3D geometries with quantum chemistry (DFT single-points)**
- **Verify canonical labeling (no duplicates)**

0.7 7. Resources Milestones

References:

- **McKay (1998): "Isomorph-Free Exhaustive Generation"**
- **Pólya (1937): "Kombinatorische Anzahlbestimmungen"**
- **Read Corneil (1977): "Graph Isomorphism Algorithms"**

Milestones:

- **Month 2: nauty integration complete**
- **Month 4: Orderly generation working**
- **Month 6: Full CHO enumeration + stereoisomers**

0.8 8. Extensions

- **Reactivity Prediction: Which isomers are most stable/reactive?**
- **Retrosynthesis: Enumerate synthetic routes**
- **Protein Folding: Graph enumeration for polymer conformations**

End of PRD 17