Enumeration and Generation of Simple Unlabelled Graphs

Jonathan Glanfield

Supervisor: Levente Bodnár

University of Warwick

Warwick Mathematics Institute

Contents

0 Introduction

0.1 Context and Motivation

Enumerating unlabelled graphs is a classical and challenging problem in combinatorics. Unlike *labelled* graphs, where each vertex has a distinct identity, *unlabelled* graphs consider graphs equivalent under relabelling of vertices, making their enumeration significantly more complex.

This task is known to be #P-complete, meaning it is as computationally difficult as counting the solutions to NP problems, and no efficient algorithm is known for solving it in the general case [1]. Moreover, the total number of graphs on n vertices grows super-exponentially, rendering brute-force enumeration infeasible even for moderate n.

To tackle the enumeration of unlabelled graphs on n vertices, we apply combinatorial group action techniques. **Burnside's Lemma** reduces the problem to averaging fixed-point counts over the action of the symmetric group S_n on the set of all labelled graphs [2]. **Pólya's Enumeration Theorem** further refines this by encoding the cycle index polynomial of the action, producing a closed-form generating function for the number of non-isomorphic graphs [3].

Next, we address the related problem of *graph isomorphism testing*, focusing on **McKay's canonical labelling algorithm**. This method assigns a unique canonical form to each graph, allowing for efficient comparison of isomorphism classes in practice, despite a worst-case factorial time complexity. The algorithm's success stems from powerful symmetry breaking and pruning strategies. [4, 5].

Finally, we develop a **recursive enumeration algorithm** that generates all unlabelled simple graphs on n vertices. Starting from smaller graphs, the algorithm incrementally adds a new vertex in all non-isomorphic ways by selecting one representative from each orbit of the automorphism group of the current graph. At each step, canonical labelling is used to detect and discard duplicates.

0.2 Basic Definitions

Definition 0.1 (Simple Graph)**.** *A* **simple graph** G *is a pair* (V, E) *where* V *is a finite set of* **vertices** *and* $E \subseteq \big\{\{u, v\} \colon u, v \in V,\ u \neq v\big\}$ *is a set of unordered pairs called* **edges***.*

Definition 0.2 (Labelled vs. Unlabelled Graph)**.** *A* **labelled** *graph on the vertex set* $[n] = \{1, 2, \ldots, n\}$ *is a simple graph in which each vertex carries a distinct label in* $[n]$*.*

An **unlabelled** *graph is an isomorphism class of labelled graphs: two labelled graphs on* $[n]$ *represent the same unlabelled graph if there exists a bijection of* $[n]$ *taking edges to edges.*

Definition 0.3 (Graph Isomorphism)**.** *Two graphs* $G = (V, E)$ *and* $H = (W, F)$ *are* **isomorphic***, written* $G \cong H$*, if there exists a bijection* $\varphi : V \to W$ *such that:*

$$\{u, v\} \in E \quad \Longleftrightarrow \quad \{\varphi(u), \varphi(v)\} \in F.$$

This bijection preserves adjacency in the vertex set V*.*

Definition 0.4 (Automorphism Group)**.** *An* **automorphism** *of a graph* $G = (V, E)$ *is a permutation* $\sigma \in S_V$ *of the vertex-set* V *such that* σ *preserves adjacency. The set of all automorphisms forms a group under composition, denoted* $\mathbf{Aut}(\mathbf{G})$*.*

Definition 0.5 (Group Action, Orbit and Stabiliser)**.** *A* **group action** *of a group* G *on a set* X *is a map* $G \times X \to X$*,* $(g, x) \mapsto g \cdot x$*, satisfying* $e \cdot x = x$ *and* $(gh) \cdot x = g \cdot (h \cdot x)$*.*

- *For* $x \in X$*, the* **orbit** *of* x *is* $G \cdot x = \{g \cdot x : g \in G\}$*.*
- *The* **stabiliser** *of* x *is* $G_x = \{g \in G : g \cdot x = x\}$*.*

1 Graph Enumeration and Pólya's Enumeration Theorem

We begin with the problem of determining the number of simple unlabelled graphs on n vertices. This problem is known to be #P complete, meaning no known algorithm runs in polynomial time for all n.

An efficient approach is given through the use of Pólya's Enumeration Theorem, which uses the action of the symmetric group on edge labels to account for graph automorphisms, transforming the enumeration into a manageable sum over orbits of the group action.

This section also draws on insights from Chapters 2 and 4.1 of Harary and Palmer's *Graphical Enumeration* [6], which helped guide the development of these enumerative techniques.

1.1 Burnside's Lemma

Lemma 1.1 (Burnside)**.** *Let G be a finite group acting on a finite set X. The number of orbits of X under G is*

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|,$$

where $X^g = \{x \in X : g \cdot x = x\}$.

Proof. Consider the set

$$S = \{(g, x) \in G \times X : g \cdot x = x\}.$$

We can count $|S|$ two ways:

For the first, we consider for a fixed $g \in G$, the number of pairs (g, x) with $g \cdot x = x$ is exactly $|X^g|$.

Thus:

$$|S| = \sum_{g \in G} |X^g|.$$

Secondly, for a fixed $x \in X$, the number of $g \in G$ such that $g \cdot x = x$ is the size of the stabiliser subgroup G_x. By the Orbit–Stabiliser Theorem,

$$|G_x| = |G|/|G \cdot x|$$

Where $G \cdot x$ is the orbit of x. Summing over all x,

$$|S| = \sum_{x \in X} |G_x| = \sum_{x \in X} \frac{|G|}{|G \cdot x|}.$$

We can group the sum by orbits; each orbit O contributes $|O|$ terms, each equal to $|G|/|O|$, so the total is $|O| \cdot (|G|/|O|) = |G|$.

We see that for $|X/G|$ orbits

$$|S| = |G|\,|X/G|.$$

Equating the two counts gives

$$\sum_{g \in G} |X^g| = |G|\,|X/G|,$$

and hence the formula holds. □

1.2 Cycle Index Polynomial

Definition 1.2. *The cycle index of a permutation group G on the object set $X = \{1, 2, ..., n\}$ is the polynomial on variables $x_1, x_2, \ldots, x_n$ written as*

$$Z(G, x_1, x_2, \ldots, x_n) = \frac{1}{|G|} \sum_{g \in G} \prod_{j=1}^{m} x_j^{c_j(g)},$$

where $c_j(g)$ is the number of cycles of length j in the disjoint cycle decomposition of g.

It can be thought of as a way of encoding the cycle structure of all permutations of some finite permutation group.

1.3 Pólya's Enumeration Theorem

Theorem 1.3 (Pólya)**.** *Let G act on $X = \{x_1, \ldots, x_m\}$ and let A be a set of q colours. Then the number of distinct colourings $f : X \to A$ up to the action of G is*

$$\left|A^X/G\right| = Z(G, q, q, \ldots, q).$$

Proof. We know that for a colouring $f : X \to A$ to be fixed by g, we need $f(g \cdot x) = f(x)$ for all $x \in X$.

For $g \in G$, we denote $j_k(g)$ as the number of cycles of length k.

We are looking to find how many colourings exist which are fixed by g. By fixture, each cycle must be a single colour, of which there are q colours to choose from, so the total number of possible colourings which are fixed by g is precisely the sum $q^{\sum_k j_k(g)}$.

By Burnside's Lemma,

$$\left|A^X/G\right| = \frac{1}{|G|}\sum_{g\in G}|X^g| = \frac{1}{|G|}\sum_{g\in G} q^{j_1(g)+j_2(g)+\cdots+j_m(g)} = Z(G, q, q, \ldots, q).$$

□

1.4 Counting Unlabelled Graphs

Let $X = \{1, 2, \ldots, n\}$ and let $X^{(2)}$ be the set of unordered pairs (edges) $\big\{\{i, j\} \colon 1 \le i < j \le n\big\}$.

The symmetric group $G = S_n$ acts on X by permuting vertices, while $S_n^{(2)}$ is induced by S_n which acts on $X^{(2)}$. Specifically, each permutation $\sigma \in S_n$ induces a permutation $\sigma' \in S_n^{(2)}$ such that for every $\{i, j\} \in X^{(2)}$,

$$\sigma' \cdot \{i, j\} = \{\sigma(i), \sigma(j)\}.$$

Hence each 2-colouring of $X^{(2)}$ corresponds to a distinct graph of X where the colours represent the presence/absence of an edge, and as such a permutation on vertices is induced on each edge pair.

Using PET, we derive the formula counting the number of unlabelled graphs with n vertices as:

$$g(n) = \left|\{\text{graphs on } n \text{ vertices}\}/S_n^{(2)}\right| = Z\big(S_n^{(2)}, 2, 2\ldots, 2\big) = \frac{1}{n!}\sum_{\sigma\in S_n^{(2)}} 2^{c(\sigma)}.$$

Example: Enumerating Graphs on 4 Vertices

We begin by listing the elements of S_4 grouped by cycle type:

#	Permutations
1	(1)(2)(3)(4)
6	(1)(2)(34), (1)(3)(24), (1)(4)(23), (2)(3)(14), (1)(4)(13), (3)(4)(12)
8	(1)(234), (1)(243), (2)(134), (2)(143), (3)(124), (3)(142), (4)(123), (4)(132)
3	(12)(34), (13)(24), (14)(23)
6	(1234), (1243), (1324), (1342), (1423), (1432)

We now compute the cycle index of the induced action S_4. Table 1 summarises the contributions from each conjugacy class to the cycle index polynomial $Z\big(S_4\big)$.

$$Z(S_4) \;=\; \frac{1}{24}\Big(1 \cdot s_1^4 + 6 \cdot s_1^2 s_2 + 8 \cdot s_1 s_3 + 3 \cdot s_2^2 + 6 \cdot s_4\Big).$$

Next, consider the induced action on the set of $\binom{4}{2} = 6$ edges, namely $\{12, 13, 14, 23, 24, 34\}$, to form the cycle–index $Z\big(S_4^{(2)}\big)$.

Examining how the conjugacy classes act on the edge set:

- **Identity** $(a)(b)(c)(d)$: fixes all 6 pairs - s_1^6.
- **Transpositions** $(ab)(c)(d)$: fixes the edges (ab) and (cd), swaps the other four $(ac\ bc)$, $(ad\ bd)$ - $s_2^2 s_1^2$.
- 3**-Cycles** $(abc)(d)$: acts on the six edges as two disjoint 3-cycles $(ab\ bc\ ca)$ and $(ad\ bd\ cd)$ - s_3^2
- **Double-Transpositions** $(ab)(cd)$: same as single transposition, fixes the edges (ab) and (cd), swaps the other four $(ac\ bd)$, $(ad\ bc)$ - $s_2^2 s_1^2$.
- 4**-Cycles** $(abcd)$: acts on the six edges as a 4-cycle $(ab\ bc\ cd\ da)$ and a transposition $(ac\ bd)$ - $s_2 s_4$

To clarify this induced action, we provide an example based on a specific vertex relabelling:

Example. Let $\sigma = (1)(2)(34)$. Its action on the edges is:

$$\begin{aligned} 12 &\mapsto 12, \qquad 34 \mapsto 34, \\ 13 &\mapsto 14, \qquad 23 \mapsto 24, \\ 14 &\mapsto 13, \qquad 24 \mapsto 23. \end{aligned}$$

Thus the induced permutation on the six edges is:

$$(12)(34)\,(13\,14)(23\,24),$$

giving the monomial term $s_1^2 s_2^2$.

We now compute the cycle index of the induced action $S_4^{(2)}$. Table 2 summarises the contributions from each conjugacy class to the cycle index polynomial $Z\big(S_4^{(2)}\big)$.

$$Z\big(S_4^{(2)}\big) \;=\; \frac{1}{24}\Big(s_1^6 + 9\,s_1^2 s_2^2 + 8\,s_3^2 + 6\,s_2 s_4\Big).$$

Finally, each edge may be either *present* or *absent*, so we substitute

$$s_k \;\mapsto\; 2 \quad \text{for all } k.$$

Hence:

$$Z\big(S_4^{(2)}\big)\big|_{s_k=2} = \;\frac{1}{24}\big(2^6 + 9\cdot 2^4 + 8\cdot 2^2 + 6\cdot 2^2\big) \;=\; 11.$$

Thus there are exactly **eleven** simple, unlabelled graphs on four vertices.

Table 1: Conjugacy classes in $Z(S_4)$

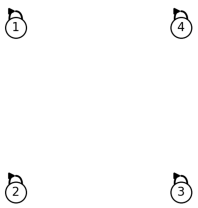
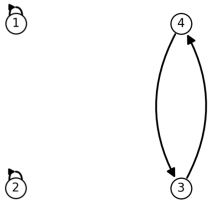
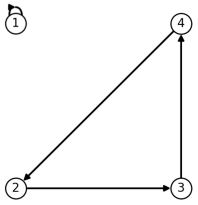
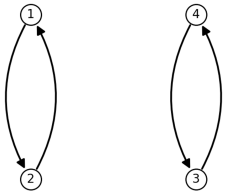
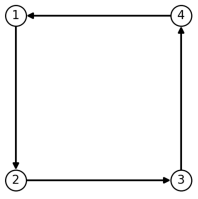
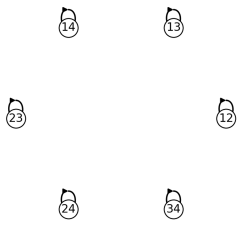
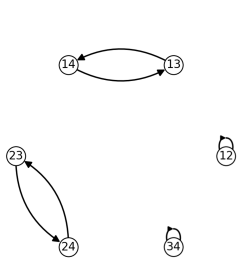
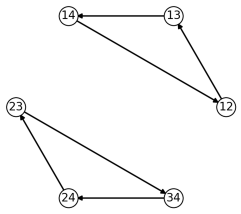
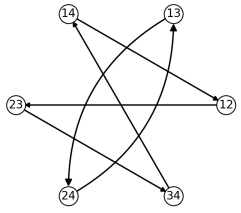
Term of $\mathbf{Z(S_4)}$	**Permutation in $\mathbf{S_4}$**	**Diagram**
s_1^4	(1)(2)(3)(4)	
$s_1^2 s_2$	(1)(2)(34)	
$s_1 s_3$	(1)(234)	
s_2^2	(12)(34)	
s_4	(1234)	

Table 2: Conjugacy classes in $Z\big(S_4^{(2)}\big)$

Term of $\mathbf{Z\big(S_4^{(2)}\big)}$	**Permutation in $\mathbf{S_4^{(2)}}$**	**Diagram**
s_1^6	(12)(13)(14)(23)(24)(34)	
$s_1^2 s_2^2$	(12)(34)(13 14)(23 24)	
s_3^2	(12 13 14)(23 34 24)	
$s_2 s_4$	(13 24)(12 23 34 14)	

2 Isomorphism Testing via McKay's Canonical Labelling

We now turn to the problem of determining isomorphisms amongst graphs - that is, given two input graphs, does there exist a bijection between their vertex sets that preserves adjacency?

McKay's Algorithm generates a canonical label for each graph, typically by choosing the lexicographically maximal reordering of vertices, such that graphs share the same label if and only if they are in the same isomorphism class. Hence, checking for the existence of an isomorphism between graphs can be reduced to checking the equivalence of their respective canonical labels.

Since the graph isomorphism problem is known to be in NP (though not known to be NP-complete or in P), the labelling process itself is in $O(\exp(n))$. However, careful algorithmic design ensures that it typically runs very quickly on real-world graphs.

This section was informed in part by Hartke and Radcliffe's *McKay's Canonical Graph Labelling Algorithm* [7], which provided useful insights into the labelling process and its practical performance.

2.1 Ordered Partitions and Refinement

Definition 2.1 (Ordered Partition)**.** *An ordered partition of* $\{1, \ldots, n\}$ *is a sequence;*

$$\pi = (V_1, \ldots, V_r)$$

consisting of non-empty, disjoint subsets with $\bigcup_i V_i = \{1, \ldots, n\}$*.*

Each V_i is called a *part*. We call a part of size one a *singleton*. There are two special cases for partitions of $\{1, \ldots, n\}$:

1. The *unit partition* μ has one part.
2. The *discrete partition* has n singletons.

We can define a partial ordering on ordered partitions, namely *refinement*:

Definition 2.2 (Refinement)**.** *Let π and π' be two ordered partitions. We say that π is finer than π' if:*

- *Every part $V_i \in \pi$ is contained in some part $W_k \in \pi'$.*
- *The ordering of parts in π is maintained, i.e. if $V_i, V_j \in \pi,\ i < j$, then for $W_k, W_l \in \pi'$ where $V_i \subseteq W_k, V_j \subseteq W_l$, we have that $k \leq l$.*

For McKay's Algorithm, we say that $\{1, \ldots, n\}$ corresponds to the list vertices of a graph $G = (V, E)$ where $|V| = n$.

Furthermore, a partition π can be seen as a classification of the vertices (namely degree information). Vertices in different parts of π have been distinguished from one another, while vertices in the same part have not.

If we have two vertices $v, v' \in V_i$ which can be distinguished (here we check if they have different degrees into a part of π), we can further refine our classification of vertices.

At some point we will reach a stage where all distinguishable vertices have been refined to a separate part. We call this an *equitable partition*:

Definition 2.3 (Equitable Partition)**.** *An ordered partition $\pi = (V_1, \ldots, V_r)$ is equitable for a graph $G = (V, E)$ if $\forall\ 1 \leq i, j \leq r$ and $v, v' \in V_i$,*

$$|\Gamma(v) \cap V_j| = |\Gamma(v') \cap V_j|,$$

where $\Gamma(v)$ denotes the neighbourhood of v in G.

We want to be able to take an inequitable ordered partition, break up the parts containing vertices with differing degrees to parts of π, and refine iteratively until we reach a coarsest equitable partition.

The process in which we "break up" these parts is called *shattering*:

Definition 2.4 (Shattering)**.** *Given a partition $\pi = (V_1, \ldots, V_r)$, a part V_j* shatters *part V_i if there exists $v, v' \in V_i$ with*

$$|\Gamma(u) \cap V_j| \neq |\Gamma(u') \cap V_j|$$

The shattering of V_i by V_j splits V_i into subparts $X_1, \ldots, X_t$, grouping vertices by equal numbers of neighbours in V_j, ordered by increasing degree.

With these procedures, we can define the equitable refinement algorithm which lies at the heart of McKay's Algorithm.

2.2 Equitable Refinement Algorithm

Input: Graph G, ordered partition π
Output: Coarsest equitable refinement $R(\pi)$
`// Initialize`
$\tau \leftarrow \pi$;
while *there exist parts* i, j *with* V_j *shattering* V_i *in* τ **do**
  pick lexicographically minimal (i, j);
  compute shattering $\{X_1, \ldots, X_t\}$ of V_i by V_j;
  replace V_i in τ with $X_1, \ldots, X_t$;
end
return τ

Lemma 2.5 (Termination and Correctness)**.** *For any ordered partition* π*, the algorithm terminates in at most* $n - 1$ *shatterings and returns a coarsest equitable refinement* $R(\pi)$*.*

Proof. Each shattering strictly increases the number of parts; at most n parts exist, so termination holds. In the outputted partition, no part shatters another, so τ is equitable. Any equitable refinement must coarsen each shattering, so τ is finer than before at each stage, thus $R(\pi)$ is finer than π.

Let α be any coarsest equitable partition of π. By definition, any equitable refinement of π must be finer than or equal to α. The partition $R(\pi)$ is an equitable refinement of π; hence, $R(\pi)$ must be finer than or equal to α.

However, since shattering is only required to satisfy the equitability of τ, we have that $R(\pi)$ is in its coarsest equitable form, and therefore should be coarser than or equal to α. Thus $R(\pi)$ and α are the same up to ordering of parts. □

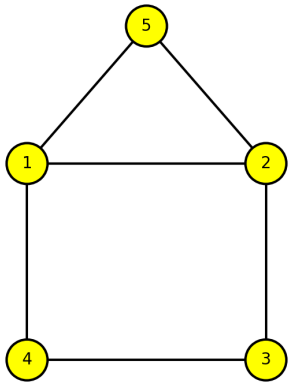
Example: Equitable Refinement on "House" Graph

Consider the graph G shown below on vertices $\{1, 2, 3, 4, 5\}$ with edges

$$E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{1, 5\}, \{2, 5\}\},$$

which forms a 4-cycle with roof vertex 5.

Starting from the unit partition $\mu = (\{1, 2, 3, 4, 5\})$, we compute shattering steps:

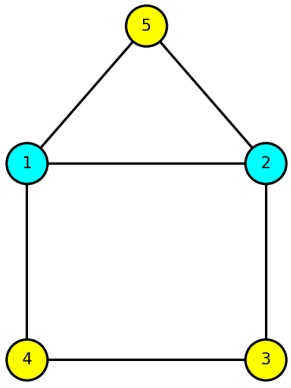


Step 1: Compute vertex degrees in the sole part:

$$\deg(1) = 3, \quad \deg(2) = 3, \quad \deg(3) = 2, \quad \deg(4) = 2, \quad \deg(5) = 2.$$

Shattering by these degrees gives:

$$\pi = (\{3, 4, 5\}, \{1, 2\})$$

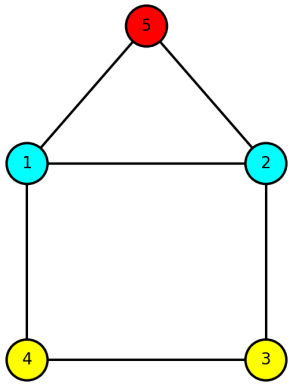


Step 2: Now check if part $\{3, 4, 5\}$ is shattered by $\{1, 2\}$:

$$|\Gamma(3) \cap \{1, 2\}| = 1, \quad |\Gamma(4) \cap \{1, 2\}| = 1, \quad |\Gamma(5) \cap \{1, 2\}| = 2.$$

Splitting sorts by these counts (increasing):

$$\pi = (\{3, 4\}, \{5\}, \{1, 2\})$$



No further shattering is possible: vertices in each part have identical neighbour counts into every other part.

Hence the coarsest equitable refinement is

$$R(\mu) = (\{3,4\}, \{5\}, \{1,2\}).$$

2.3 Search Tree and Individualisation

McKay's Algorithm works by starting from initial partition $R(\mu)$ and building a search tree $T(G)$ whose nodes are ordered partitions. If a node π is discrete, it is a leaf. Otherwise, we must introduce a distinction between vertices of parts, importantly this distinction is artificial as the partition is already in equitable form.

To do this, we introduce the notion of *splitting*:

Definition 2.6 (Splitting)**.** *Let k be the smallest index with $|V_k| > 1$, and let $v \in V_k$. We define the splitting of V_k by v as:*

$$\pi \perp v = (V_1, \ldots, V_{k-1}, \{v\}, V_k \setminus \{v\}, V_{k+1}, \ldots, V_r)$$

With this notation, we can define the search tree as follows:

Definition 2.7 (Search Tree)**.** *The search tree $T(G)$ is defined by its nodes:*

$$(\pi; u) : \pi \text{ is an ordered partition of } \{1, \ldots, n\},$$
$$u \text{ is a sequence of vertices } (u_1, u_2, \ldots, u_k).$$

Given a node $(\pi; u)$, its children are nodes of the form $(\pi; (u_1, u_2, \ldots, u_k, v))$, where:

$$\pi = R(\pi \perp v), \quad \forall\ v \in V_k.$$

If we let u be the empty sequence, we obtain the root node with $\pi = R(\mu)$, which is the initial equitable refinement of the vertex set.

Leaf nodes correspond to discrete ordered partitions of $\{1, \ldots, n\}$ by definition of the equitable refinement algorithm. As such, we can define a permutation $\sigma_\pi \in S_n$ for each leaf node π, where:

$$\sigma_\pi(i) = j \quad \text{if vertex } i \text{ appears in the } j\text{-th cell of } \pi, \quad \text{for } i, j \in \{1, \ldots, n\}.$$

Comparing the permutations σ_π corresponding to the leaf nodes of the search tree is the core idea in McKay's canonical labelling algorithm. This is shown below.

Definition 2.8 (Canonical Label)**.** *Given a graph $G = (V, E)$, McKay's canonical labelling $C_M(G)$ is defined as:*

$$C_M(G) = \max_{\preceq} \{\sigma_\pi(G) : \pi \text{ is a leaf of } T(G)\},$$

where $\preceq$ is a lexicographic order on graph edge sets.

To show correctness of this labelling function, we must first explore how permutations on a graph affect the search tree.

We should expect the canonical label of the graph and its permutation to be the same since the graphs are isomorphic (we can define a bijection between vertices as the permutation given).

Lemma 2.9 (Tree Equivalence)**.** *Given a graph G and some $\psi \in S_n$, if $H = \psi \circ G$, then $T(H) = \psi \circ T(G)$.*

Proof. We prove by induction on tree depth.

At the root, we have $R(\psi \circ \mu) = \psi \circ R(\mu)$ since the refinement algorithm respects action by ψ.

When splitting, if the first non-trivial part of π is V_i, then the first non-trivial part of $\psi \circ \pi$ is $\psi \circ V_i$. As such, each child of $\psi \circ \pi$ is precisely the image of children of π under ψ.

By induction, this correspondence holds at every level of the tree. Hence, $T(H) = \psi \circ T(G)$. □

Using this, we can easily prove that $C_M(G)$ is indeed a canonical labelling of G:

Theorem 2.10 (Correctness of C_M)**.** *$C_M(G)$ is a canonical labelling of G.*

Proof. If $G \cong H$, let $H = \psi \circ G$. By Lemma 2.3, we have that the leaves of $T(H)$ are equivalent to that of $T(G)$ under permutation by ψ. From the action of ψ on the partition π, we have:

$$\sigma_{\psi \circ \pi} = \psi^{-1} \circ \sigma_\pi$$

As such, the canonical label of H for a given leaf node $\psi \circ \pi$ is:

$$\sigma_{\phi \circ \pi}(H) = \psi^{-1} \circ \sigma_\pi(H) = \psi^{-1} \circ \sigma_\pi \circ (\psi(G)) = \sigma_\pi(G)$$

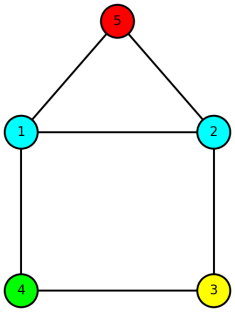
Thus the labels are the same, as are the maximum labels, and so $C_M(G) = C_M(H)$. □

Example: Search Tree Construction

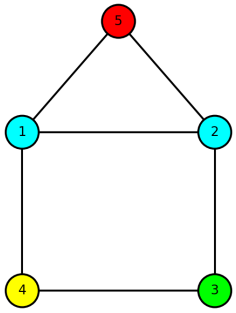
Continuing from the previous example and starting from the equitable partition $\pi_0 = (\{3,4\}, \{5\}, \{1,2\})$, we build the search tree:

1. The first non-singleton part is $\{3,4\}$. Splitting by each gives two children:

$$\pi_1 = (\{3\}, \{4\}, \{5\}, \{1,2\}), \quad \pi_2 = (\{4\}, \{3\}, \{5\}, \{1,2\}).$$



Splitting by $\{3\}$ - π_1



Splitting by $\{4\}$ - π_2

2. Refine each child using equitable refinement algorithm:

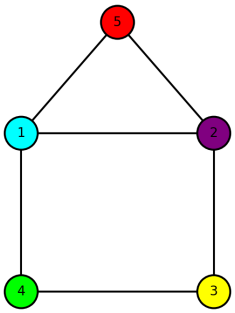
   In both π_1, π_2, the only non-singleton part remains $\{1, 2\}$. By shattering again we have:

   $$(\{3\}, \{4\}, \{5\}, \{1\}, \{2\}) \quad \text{and} \quad (\{4\}, \{3\}, \{5\}, \{2\}, \{1\}),$$

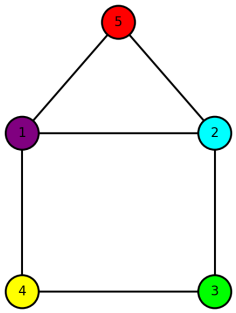
   respectively, hence π_1 and π_2 are leaf nodes.

Thus $T(G)$ has root π_0 with two leaves (children) corresponding to the vertex orderings:

$$[3, 4, 5, 1, 2], \ [4, 3, 5, 2, 1]$$

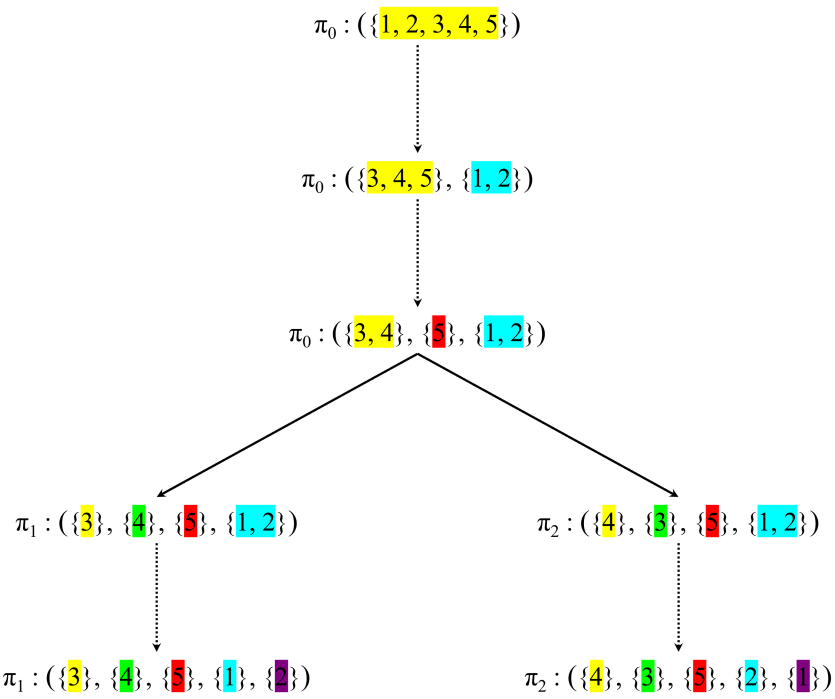


Leaf node π_1



Leaf node π_2

Each leaf induces a permutation $\sigma_\pi \in S_5$ mapping original labels to these orders.



Complete search tree for G, with solid lines indicating splitting steps and dashed lines representing equitable refinement steps.

2.4 Pruning via Automorphisms

Although the search tree correctly identifies the canonical label of a graph G, we can increase the efficiency of the algorithm by pruning via the automorphism group of G.

Since there will be at least $|\mathrm{Aut}(G)|$ leaf nodes corresponding to equivalent labellings, detecting these symmetries early allows us to avoid redundant exploration when using depth-first search.

First, we show that an automorphism exists between two leaf nodes if the permutation induced on the graph is the same between them:

Lemma 2.11 (Tree Automorphism)**.** *Suppose that there exists two leaf nodes, corresponding to individual permutations π and τ, such that:*

$$\sigma_\pi(G) = \sigma_\tau(G).$$

Then the composition

$$\sigma' = \sigma_\tau^{-1} \circ \sigma_\pi$$

is a non-trivial automorphism of G, i.e. $\sigma' \in \mathrm{Aut}(G)$.

Proof. We compute:

$$\sigma'\big(G\big) = \big(\sigma_\tau^{-1} \circ \sigma_\pi\big)(G) = \sigma_\tau^{-1}\big(\sigma_\pi(G)\big) = \sigma_\tau^{-1}\big(\sigma_\tau(G)\big) = G.$$

Hence $\sigma' \in \mathrm{Aut}(G)$. □

Using this result, we prune as follows:

Let π and τ be leaf nodes in the depth-first search tree with

$$\sigma_\pi(G) = \sigma_\tau(G).$$

Let P be their deepest common ancestor, and denote by P_π and P_τ the respective children of P on the paths to π and τ. The automorphism $\sigma' = \sigma_\tau^{-1} \circ \sigma_\pi$ sends P_π to P_τ while fixing P, so the entire subtree under P_τ is isomorphic to that under P_π. After fully exploring P_π, we may prune P_τ and all its descendants.

Alternatively, we can maintain a set of discovered automorphisms to prune earlier:

Let D be any node visited during the search, and let Λ be the group generated by all automorphisms identified so far. Let

$$\Phi = \{\, \lambda \in \Lambda : \lambda(D) = D\}$$

be the subgroup that stabilises D.

Suppose two children D_α and D_β of D satisfy

$$\exists\ \lambda \in \Phi,\ \text{s.t. } \lambda(D_\alpha) = D_\beta$$

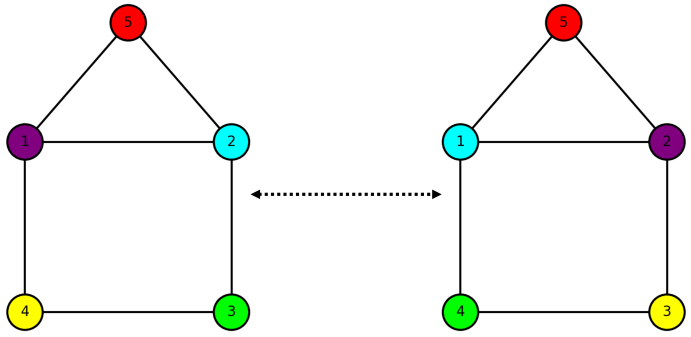
i.e. an element of Φ maps D_α to D_β.

Then after exploring the subtree rooted in D_α, we can prune the subtree rooted in D_β since all children of D_β will be isomorphic to those we have already discovered.

Example: Automorphism Detection and Pruning

As can be seen below, the house graph G admits the reflection symmetry swapping $1 \leftrightarrow 2$ and $3 \leftrightarrow 4$, namely

$$\sigma' = (1\ 2)(3\ 4), \quad \sigma'(E) = E.$$



In the search tree above, each corresponds to an orbit under this automorphism:

- Leaves with $\{1, 2\}$ ordered as $(1, 2)$ from splitting by $\{3, 4\}$.
- Leaves with $\{1, 2\}$ ordered as $(2, 1)$ from splitting by $\{3, 4\}$.

Once we explore, the leaf node $\pi_1 = (\{3\}, \{4\}, \{5\}, \{1\}, \{2\})$, we detect that σ' maps its children to those under π_2.

We therefore prune the entire π_2 branch, avoiding exploration of the second leaf node and obtaining the canonical label from the first leaf node visited.

In larger graphs, such pruning can dramatically reduce the search tree computation, thus allowing for fast and efficient isomorphism detection.

3 Generation of Simple Unlabelled Graphs

3.1 Recursive Generation via Automorphism Elimination

We now give a formal recursive method to generate all (simple and unlabelled) graphs on n vertices.

The key idea is to extend each graph on $n-1$ vertices by introducing a new vertex, picking a representative for each edge combination up to automorphism, and finally checking for isomorphism between these generated graphs.

Definition 3.1 (Automorphisms on Subsets)**.** $\mathrm{Aut}(G)$ *acts on the power set* $\mathcal{P}(V)$*, such for a given* $S \in \mathcal{P}(V)$*,* $\gamma \in \mathrm{Aut}(G)$*, we have:*

$$\gamma \cdot S = \{\gamma(v) \mid v \in S\}$$

This partitions $\mathcal{P}(V)$ into disjoint *orbits*.

To avoid redundancy when extending G, we select one representative from each orbit, since any two subsets in the same orbit produce isomorphic extensions:

Definition 3.2 (Orbit Representatives)**.** *Let* G *be a graph with vertex set* V*, and let* $\mathrm{Aut}(G)$ *act on the power set* $\mathcal{P}(V)$*. Denote by*

$$\mathcal{O}_G = \mathcal{P}(V) \,/\, \mathrm{Aut}(G)$$

the collection of all $\mathrm{Aut}(G)$*–orbits in* $\mathcal{P}(V)$*. We choose a single representative from each orbit, and write*

$$\mathcal{S}_G = \{\, S_1, S_2, \ldots, S_m\},$$

where each S_i *is arbitrarily chosen element of the* i*-th orbit* $o_i \in \mathcal{O}_G$*.*

Example: Orbit Representatives in the "Cherry" Graph on 3 Vertices

Let G be the "cherry" graph on vertex set $V = \{0, 1, 2\}$ with edge set

$$E = \big\{\{0, 1\},\, \{1, 2\}\big\}.$$

Its automorphism group $\mathrm{Aut}(G)$ is generated by the swap $\gamma = (0\ 2)$, which fixes vertex 1. We list the orbits of $\mathrm{Aut}(G)$ acting on $\mathcal{P}(V)$:

$$\begin{aligned}
&\text{Orbit 1:} && \emptyset \\
&\text{Orbit 2:} && \{1\} \\
&\text{Orbit 3:} && \{0\},\ \{2\} \\
&\text{Orbit 4:} && \{0,1\},\ \{1,2\} \\
&\text{Orbit 5:} && \{0,2\} \\
&\text{Orbit 6:} && \{0,1,2\}
\end{aligned}$$

We choose one representative from each orbit, for example:

$$\mathcal{S}_G = \big\{\, \emptyset,\ \{1\},\ \{0\},\ \{0,1\},\ \{0,2\},\ \{0,1,2\} \big\}.$$

Each $S_i \in \mathcal{S}_G$ forms a distinct extension G_i on 4 vertices up to isomorphism.

Once we have a list of all orbit representatives, we can extend G to include all unique n vertex extensions of G up to isomorphism.

Definition 3.3 (Graph Extension)**.** *Given G on vertex set $V = [n-1]$, and a subset $S \subseteq V$. We define the extension $G_{\{S\}} = (V_{\{S\}}, E_{\{S\}})$ on n vertices by:*

$$V_{\{S\}} = V \cup \{n\}, \quad E_{\{S\}} = E \cup \{\{u, n\} : u \in S\}.$$

Repeating this process on all $G \in \mathcal{G}_{n-1}$ and each orbit representative S_i, generates a complete collection of graphs G_n on n vertices:

Definition 3.4 (Collection)**.** *Let $\mathcal{E}_n$ denote the collection of all graph extensions generated from orbit representatives:*

$$\mathcal{E}_n = \left\{ G_{\{S\}} : G \in \mathcal{G}_{n-1},\ S \in \mathcal{S}_G \right\},$$

However, even after pruning using the automorphism group for each $G \in \mathcal{G}_{n-1}$, graphs G_i and H_j from differing $G, H \in \mathcal{G}_{n-1}$ may still be isomorphic.

Example: Isomorphism Between Two Different Extensions

Let G again be the cherry graph on $\{0,1,2\}$, and H be the complete graph K_3 on the same vertex set. Consider two extensions:

- Extend G by $S_4 = \{0,1\}$. Define new vertex 3. Then

$$G' \;=\; G \cup \big\{\{0,3\},\{1,3\}\big\} \quad \text{has edges} \quad \{\{0,1\},\{1,2\},\{0,3\},\{1,3\}\}.$$

- Extend $H = K_3$ by $T = \{0\}$. Call the new vertex 3. Then

$$H' \;=\; K_3 \cup \big\{\{0,3\}\big\} \quad \text{has edges} \quad \{\{0,1\},\{0,2\},\{1,2\},\{0,3\}\}.$$

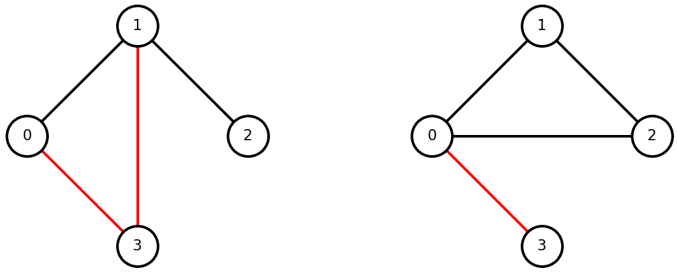
Define a bijection $\phi : V(G') \to V(H')$ by

$$\phi(0) = 1, \quad \phi(1) = 0, \quad \phi(2) = 3, \quad \phi(3) = 2.$$

By checking;

$$\{u,v\} \in E(G') \iff \{\phi(u),\phi(v)\} \in E(H'),$$

we conclude $G' \cong H'$. Thus, although these came from different base graphs and subsets, they represent the same isomorphism class.



Isomorphic G' (left) and H' (right).

To remove isomorphic duplicates, we apply a canonical labelling algorithm (e.g. via McKay's) to $\mathcal{E}_n$, which can be written as:

$$\mathcal{G}_n = \mathcal{E}_n \,/\, \cong$$

This is the set of all (simple, unlabelled) graphs on n vertices.

3.2 Graph Generation Algorithm

We can describe the procedure to generate $\mathcal{G}_n$ as follows:

Input: List $\mathcal{G}_{n-1}$ of all graphs on $n-1$ vertices with their automorphism groups
Output: List $\mathcal{G}_n$ of all graphs on n vertices

$\mathcal{E}_n \leftarrow \emptyset$;
foreach $G \in \mathcal{G}_{n-1}$ **do**
  compute $\mathrm{Aut}(G)$;
  foreach *representative* $S \in \mathcal{P}(V) \,/\, \mathrm{Aut}(G)$ **do**
    declare $G_{\{S\}} \leftarrow G \cup \{n\}$ with $N(n) = S$;
    add $G_{\{S\}}$ to $\mathcal{E}_n$;
  end
end
$\mathcal{G}_n \leftarrow \mathcal{E}_n \,/\, \cong$ using McKay's canonical labelling algorithm;
return $\mathcal{G}_n$

Theorem 3.5 (Correctness)**.** *The algorithm described above constructs exactly one representative from each isomorphism class of n-vertex graphs:*

$$\mathcal{G}_n = \{G_i : G \in \mathcal{G}_{n-1},\, S_i \in \mathcal{P}(V) \,/\, \mathrm{Aut}(G)\} \Big/ \cong .$$

Proof. Let H be any simple graph on n vertices. By removing any vertex v, we obtain a subgraph $G = H \setminus \{v\} \in \mathcal{G}_{n-1}$. The neighbourhood of v corresponds to some subset $S \subseteq V(G)$.

Since the algorithm considers one representative from each orbit of $\mathcal{P}(V)$ under $\mathrm{Aut}(G)$, the corresponding extension will be generated.

Finally, applying canonical labelling ensures that any duplicate graphs arising from different $G \in \mathcal{G}_{n-1}$ are identified, and only one representative per isomorphism class remains. □

3.3 Running Time and Complexity Analysis

3.3.1 Worst-Case Complexity

Let $g_{n-1} = |\mathcal{G}_{n-1}|$ and for each $G \in \mathcal{G}_{n-1}$, let $|\mathcal{O}_G| = |\mathcal{P}(V)/\mathrm{Aut}(G)|$ be the number of orbit representatives.

In the extreme case (when G is asymmetric), $\mathrm{Aut}(G)$ is trivial and hence:

$$|\mathcal{O}_G| = 2^{n-1}$$

Therefore,

$$|\mathcal{E}_n| \;=\; \sum_{G \in \mathcal{G}_{n-1}} |\mathcal{O}_G| \;\le\; g_{n-1} \cdot 2^{n-1}.$$

On the other hand, McKay's algorithm may require exploring all $n!$ vertex permutations in the worst case.

Since there are at most $|\mathcal{E}_n| \le g_{n-1} \cdot 2^{n-1}$ possible inputs, the total number of steps in the worst case is bounded by the number of inputs times the cost per input. That is, the worst-case runtime is at most:

$$(g_{n-1} \cdot 2^{n-1}) \cdot n!$$

which simplifies (up to constant factors) to

$$O(g_{n-1} \cdot n!)$$

3.3.2 Average-Case Complexity

Empirically (and proven by Erdős, Rényi), almost every graph on $n-1$ vertices is asymmetric [8]:

$$\Pr\big(\mathrm{Aut}(G) = \{\mathrm{id}\}\big) \;\longrightarrow\; 1 \qquad \text{as } n \to \infty.$$

As such, the average number of orbit representatives per graph satisfies:

$$\mathbb{E}\big[|\mathcal{O}_G|\big] \;=\; (1 - o(1))\, 2^{\,n-1} \;+\; o(1) \times (\text{smaller terms}) \;=\; (1 - o(1))\, 2^{\,n-1}.$$

Summing over all $g_{n-1} = |\mathcal{G}_{n-1}|$ graphs gives:

$$\mathbb{E}\big[|\mathcal{E}_n|\big] \;=\; \sum_{G \in \mathcal{G}_{n-1}} \mathbb{E}\big[|\mathcal{O}_G|\big] \;=\; g_{n-1}\,(1 - o(1))\, 2^{\,n-1} \;=\; (1 - o(1))\, g_{n-1}\, 2^{\,n-1}.$$

We note that when McKay's canonical labelling algorithm is optimised (via automorphism pruning, etc.), the average-case runtime can be seen as polynomial, typically $O(n^c)$ for some small constant c [4].

Hence the total average-case runtime is:

$$((1 - o(1))\, g_{n-1}\, 2^{n-1} \cdot \mathrm{poly}(n),$$

which, up to constant factors, is

$$O\big(g_{n-1}\, 2^{n-1}\, \mathrm{poly}(n)\big).$$

We can see that the average-case cost is dominated by the 2^{n-1} orbit expansion term (times a small polynomial in n) in the case for asymmetric graphs.

However, when n is small, a much greater proportion of the $n - 1$ vertex graphs contain non-trivial automorphisms, thus decreasing the number of orbit representatives and lowering the average runtime.

Appendix: Python Code

PET.py

```
from sympy.combinatorics.named_groups import SymmetricGroup
from sympy.combinatorics.permutations import Permutation
from itertools import combinations
from math import factorial, comb

# Function to count the number of non-isomorphic graphs on n vertices
def count_graphs(n):
    if n == 0:
        return 1

    # Prepare all possible vertex-pairs and their indices
    vertex_pairs = list(combinations(range(n), 2))
    index_of_pairs = {pair: idx for idx, pair in enumerate(vertex_pairs)}

    # The symmetric group acting on n vertices
    group = SymmetricGroup(n)
    total = 0

    # Sum over the conjugacy classes (Burnside's Lemma)
    for conjugacy_class in group.conjugacy_classes():
        f = next(iter(conjugacy_class))

        # Build the induced permutation on edges
        induced_index = []
        for (u, v) in vertex_pairs:
            new_u, new_v = f(u), f(v)
            image = tuple(sorted((new_u, new_v)))
            induced_index.append(index_of_pairs[image])

        induced_group = Permutation(induced_index)

        # Count how many edges are fixed
        cycle_lengths = [len(c) for c in induced_group.cyclic_form]
        moved_edges = sum(cycle_lengths)
        total_edges = comb(n, 2)
        fixed_edges = total_edges - moved_edges

        # Number of edge-orbits under this group element
        edge_orbits = len(cycle_lengths) + fixed_edges

        # Each orbit can be present/absent → 2^orbits
```

```
        term = 2 ** edge_orbits

        total += len(conjugacy_class) * term

    # Divide by |G| = n! to count distinct graphs
    return total // factorial(n)
```

McKay.py

```python
import networkx as nx

# Check if the partition is discrete (each cell has only one vertex)
def is_discrete(partition):
    return all(len(cell) == 1 for cell in partition)


# Equitable refinement of the partition
def refine(G, partition):
    P = [sorted(cell) for cell in partition]
    changed = True

    # Keep refining until partition stabilises (equitable)
    while changed:
        changed = False
        new_P = []

        for cell in P:
            if len(cell) <= 1:
                new_P.append(cell)
                continue

            buckets = {}

            # Assign vertices to buckets based on number of neighbours in
            other cells
            for v in cell:
                sig = tuple(len(set(G.neighbors(v)).intersection(other))
                for other in P if other is not cell)
                buckets.setdefault(sig, []).append(v)

            # Split the cell if vertices have different number of
            neighbours in other cells
            if len(buckets) > 1:
                changed = True
                for group in sorted(buckets):
                    new_P.append(sorted(buckets[group]))
            else:
                new_P.append(cell)

        P = new_P

    return P
```

```
# Select a non-trivial cell from the partition
def select_nontrivial_cell(partition):
    for cell in partition:
        if len(cell) > 1:
            return cell
    return None

# Individualise a partition by a vertex v
def individualize(partition, v):
    new = []

    for cell in partition:
        if v in cell:
            rest = [u for u in cell if u != v]
            new.append([v])
            if rest:
                new.append(sorted(rest))
        else:
            new.append(cell)

    return new

# Convert a partition to an ordering
def partition_to_ordering(partition):
    return [v for cell in partition for v in cell]

# Apply a labelling to a graph, stored as an adjacency matrix
def adj_matrix(G, ordering):
    idx = {v: i for i, v in enumerate(ordering)}
    n = len(ordering)
    mat = [[0] * n for _ in range(n)]

    for i, v in enumerate(ordering):
        for u in set(G.neighbors(v)):
            mat[i][idx[u]] = 1

    return mat

# Main search
def search(G, part, aut_gens, best_order=None, best_mat=None):
    # Refine the current partition
    P = refine(G, part)

    # If fully discrete, we have a complete ordering
    if is_discrete(P):
```

```
            return partition_to_ordering(P)

        # Choose a cell to individualise further
        cell = select_nontrivial_cell(P)

        for v in sorted(cell):
            # Create a refined partition by individualising v
            Pi = individualize(P, v)

            # Recursively search with the new partition
            cand = search(G, Pi, aut_gens, best_order, best_mat)
            mat = adj_matrix(G, cand)

            if best_order is None:
                best_order, best_mat = cand, mat
            else:
                if mat == best_mat and cand != best_order:
                    # New automorphism found
                    gen = {best_order[i]: cand[i] for i in range(len(cand))}
                    if gen not in aut_gens:
                        aut_gens.append(gen)
                elif mat > best_mat:
                    # Lexicographically better labeling
                    best_order, best_mat = cand, mat

        return best_order

    # Canonical labelling + automorphism group
    def canonical_label_and_aut_group(G):
        if G.nodes() == None or len(G.nodes()) == 0:
            return [], []

        init = list(G.nodes())
        gens = [{v: v for v in init}]
        lab = search(G, [init], gens)

        return lab, gens

    # Isomorphism check
    def isomorphic(G1, G2):
        lab1, _ = canonical_label_and_aut_group(G1)
        lab2, _ = canonical_label_and_aut_group(G2)
        A1, A2 = adj_matrix(G1, lab1), adj_matrix(G2, lab2)

        return A1 == A2
```

Enumeration.py

```
import networkx as nx
from itertools import combinations
from McKay import canonical_label_and_aut_group, adj_matrix

# Function to enumerate all non-isomorphic graphs on n vertices
def enumerate_graph(n):
    # Base case: n = 0 or 1 has only the empty graph
    if n <= 1:
        G = nx.empty_graph(n)
        return [G]

    # Build up from graphs on n-1 vertices
    old_graphs = enumerate_graph(n - 1)
    all_graphs = []

    for graph in old_graphs:
        V = list(graph.nodes())
        _, aut = canonical_label_and_aut_group(graph)

        seen = []
        # Try all ways of connecting the new vertex to subsets of V
        for r in range(len(V) + 1):
            for subset in combinations(V, r):
                # Skip subsets equivalent under automorphisms
                if any({gen[v] for v in subset} in seen for gen in aut):
                    continue

                seen.append(set(subset))

                # Add new vertex and edges
                graph_to_add = graph.copy()
                new_v = n - 1
                graph_to_add.add_node(new_v)
                for u in subset:
                    graph_to_add.add_edge(u, new_v)

                all_graphs.append(graph_to_add)

    # Filter out isomorphic duplicates via canonical labelling
    matrices = []
    all_graphs_iso = []

    for g in all_graphs:
        label, _ = canonical_label_and_aut_group(g)
```

```
        matrix = adj_matrix(g, label)

        if matrix not in matrices:
            matrices.append(matrix)
            all_graphs_iso.append(g)

    return all_graphs_iso
```

References

[1] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[2] Kenneth P Bogart. An obvious proof of burnside's lemma. *The American Mathematical Monthly*, 98(10):927–928, 1991.

[3] G. Pólya. Kombinatorische anzahlbestimmungen für gruppen, graphen und chemische verbindungen. *Acta Mathematica*, 68:145–254, 1937.

[4] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[5] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[6] F. Harary. *Graphical Enumeration*. Academic Press, 1969.

[7] Stephen G. Hartke and A. J. Radcliffe. Mckay's canonical graph labeling algorithm. In *Communicating Mathematics: A Conference in Honor of Joseph A. Gallian's 65th Birthday, July 16-19, 2007, University of Minnesota, Duluth, Minnesota*, Contemporary mathematics - American Mathematical Society. American Mathematical Society, 2009.

[8] P. Erdős and A. Rényi. Asymmetric graphs. *Acta Math. Acad. Sci. Hungarica*, 14(3–4):295–315, 1963.