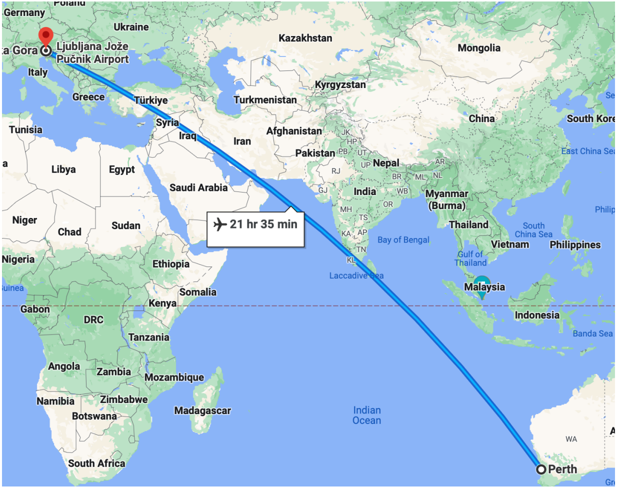
Every One a Winner: An Introduction to Orderly Algorithms

Gordon Royle

Centre for the Mathematics of Symmetry & Computation
School of Mathematics & Statistics
The University of Western Australia

June 2023

Kranjska Gora to Perth



Perth City (Kane Artie Photograph)

Ronald C Read (1924 – 2019)

In February 1988, I walked into the forbidding UWaterloo Math and Computer Building† to start a short postdoc with Ron Read.

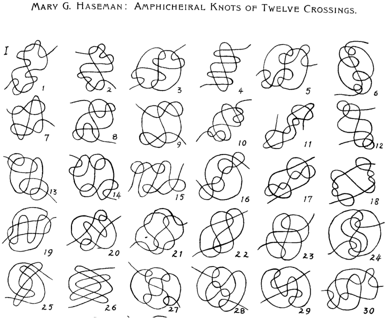
He was a true polymath, not only a brilliant mathematician and witty author, but a talented musician and composer, a lover of puzzles and an enthusiastic early adopter of computational tools in graph theory.

† Architect's motto: we put the *"brutal"* into *"brutalist architecture"*

LISTS, CATALOGUES AND CENSUSES

For hundreds—even thousands—of years, mathematicians and others have created *databases* of interesting mathematical objects.

- Numbers (prime)
- Squares (Latin, magic)
- Knots
- Matroids
- Groups (permutation, abstract)
- and of course ... *graphs*



All science is either physics or stamp-collecting

— *Ernest Rutherford*

Graph-collecting seems to have started with Isadore Kagno (1946).†

3. Graphs of Degree 6. There are eighteen admissible graphs,

$H_1 \equiv N_6$, the complete 6-point.
$H_2 \equiv [ab, ac, ad, ae, af, bc, bd, be, bf, cd, ce, cf, de, df]$
$H_3 \equiv [ab, ac, ad, ae, af, bc, bd, be, bf, cd, ce, cf, de]$
$H_4 \equiv [ac, ad, ae, af, bc, bd, be, bf, ce, cf, de, df, ef]$
$H_5 \equiv [ac, ad, ae, af, bc, bd, be, bf, ce, cf, de, df]$
$H_6 \equiv [ac, ad, ae, af, bc, bd, be, bf, cd, cf, df, ef]$
$H_7 \equiv [ab, ac, ae, af, bd, be, bf, ce, cf, de, df, ef]$
$H_8 \equiv [ab, ad, ae, af, bc, be, bf, ce, cf, de, df]$
$H_9 \equiv [ab, ac, ae, af, bd, be, bf, cd, cf, df, ef]$
$H_{10} \equiv [ad, ae, af, bd, be, bf, cd, ce, cf, df, ef]$
$H_{11} \equiv [ab, ac, ad, ae, bc, bd, bf, cd, ce, df, ef]$
$H_{12} \equiv [ad, ae, af, bc, be, bf, ce, cf, de, df, ef]$
$H_{13} \equiv [ad, ae, af, bc, be, bf, ce, cf, de, df]$
$H_{14} \equiv [ad, ae, af, bc, be, bf, cd, ce, df, ef]$
$H_{15} \equiv [ab, ae, af, bc, bf, cd, cf, de, df, ef]$
$H_{16} \equiv [ad, ae, af, bd, be, bf, cd, ce, cf, de]$
$H_{17} \equiv [ac, ad, ae, bd, be, bf, ce, cf, df]$
$H_{18} \equiv [ad, ae, af, bd, be, bf, cd, ce, cf]$

Amer. J. Math 1946

Number of vertices	Number of graphs	Author	Date	Reference
$p = 6$	156	I. Kagno	1946	[7]
$p = 7$	1,044	D. W. Crowe, F. Harary	1952	Unpublished*
		B. R. Heap	1952	Unpublished
$p = 8$	12,346	B. R. Heap	1969	[5]
$p = 9$	274,668	H. H. Baker, A. K. Dewdney, A. L. Szilard	1974	[1]
$p = 10$	12,005,168	R. D. Cameron, C. J. Colbourn, R. C. Read, N. C. Wormald	1978–1981	This paper

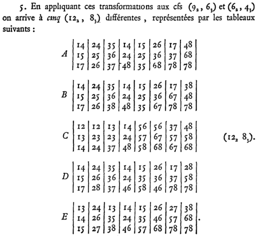
Cameron, Colbourn, Read, Wormald
J. Graph Theory 1985

†Unfortunately he missed a graph in this list.

Cubic Graphs

A *cubic graph* is one where each vertex has exactly three neighbours. Many deep problems in graph theory can be reduced to cubic graphs.

- 1966 10/12 vertices (Balaban)
- 1974 12 vertices (Petrenjuk, Petrenjuk)
- 1976 14 vertices (Bussemaker et al.)
- 1976 18 vertices (Faradžev)
- 1984 20 vertices (McKay and Royle)
- 1992 24 vertices (Brinkmann)
- 1998 24 vertices (Meringer)
- etc.



5. En appliquant ces transformations aux cfs $(9_2, 6_3)$ et $(6_2, 4_3)$ on arrive à *cinq* $(12_2, 8_3)$ différentes, représentées par les tableaux suivants :

$$A \left|\begin{array}{c|c|c|c|c|c|c|c} 14 & 24 & 35 & 14 & 15 & 26 & 17 & 48 \\ 15 & 25 & 36 & 24 & 25 & 36 & 37 & 68 \\ 17 & 26 & 37 & 48 & 35 & 68 & 78 & 78 \end{array}\right|$$

$$B \left|\begin{array}{c|c|c|c|c|c|c|c} 14 & 24 & 35 & 14 & 15 & 26 & 17 & 38 \\ 15 & 25 & 36 & 24 & 25 & 36 & 67 & 48 \\ 17 & 26 & 38 & 48 & 35 & 67 & 78 & 78 \end{array}\right|$$

$$C \left|\begin{array}{c|c|c|c|c|c|c|c} 12 & 12 & 13 & 14 & 56 & 56 & 37 & 48 \\ 13 & 23 & 23 & 24 & 57 & 67 & 57 & 58 \\ 14 & 24 & 37 & 48 & 58 & 68 & 67 & 68 \end{array}\right| \qquad (12_2\ 8_3).$$

$$D \left|\begin{array}{c|c|c|c|c|c|c|c} 14 & 24 & 35 & 14 & 15 & 26 & 17 & 28 \\ 15 & 26 & 36 & 24 & 35 & 36 & 37 & 58 \\ 17 & 28 & 37 & 46 & 58 & 46 & 78 & 78 \end{array}\right|$$

$$E \left|\begin{array}{c|c|c|c|c|c|c|c} 13 & 24 & 13 & 14 & 15 & 26 & 27 & 38 \\ 14 & 26 & 35 & 24 & 35 & 46 & 57 & 68 \\ 15 & 27 & 38 & 46 & 57 & 68 & 78 & 78 \end{array}\right|.$$

de Vries, 1891

Brinkmann, Goedgebeur, Van Cleemput, *The history of the generation of cubic graphs*, International Journal of Chemical Modeling, 2013.

Constructing databases

Many reasons to construct databases of small combinatorial objects.

- *Direct search* for examples and counterexamples
- Gaining *insight* by studying small-case behaviour
- Dealing with *low-level junk* in exact structural results

  . . . *two infinite families and five exceptional examples* . . .
- To be *integrated* into computer algebra systems

  ```
  g := SmallGroup(512,10000000);
  ```

You can also collect butterflies and make many observations. If you like butterflies, that's fine; but such work must not be confounded with research, which is concerned to discover explanatory principles (Chomsky).

The isomorphism problem

Straightforward attempts to construct databases of graphs will usually construct *many isomorphic copies* of each graph.

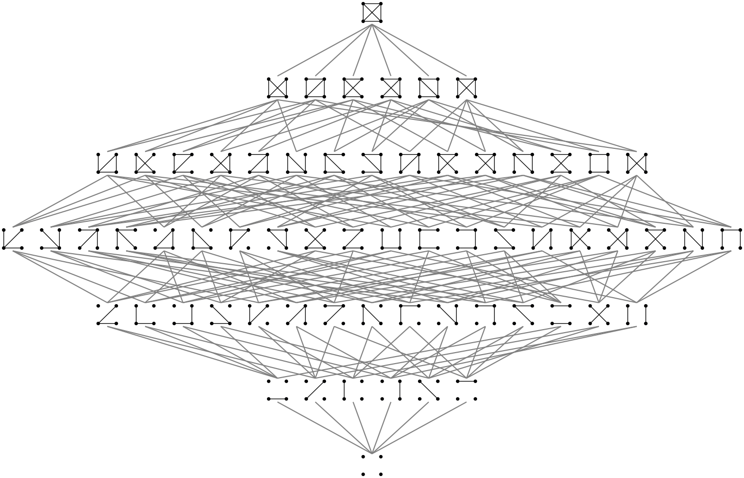
Isomorphic graphs are structurally identical so this just represents *unnecessary duplication*.

So a database should contain *exactly one graph* from each isomorphism class.

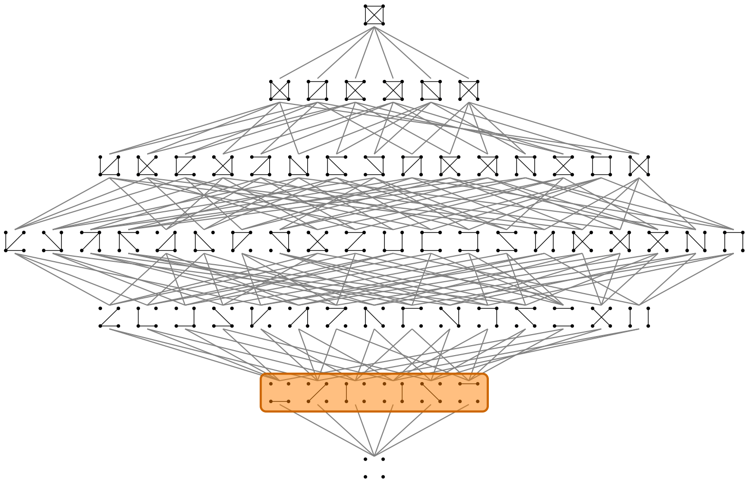
Running Example

Construct the graphs on 4 vertices.

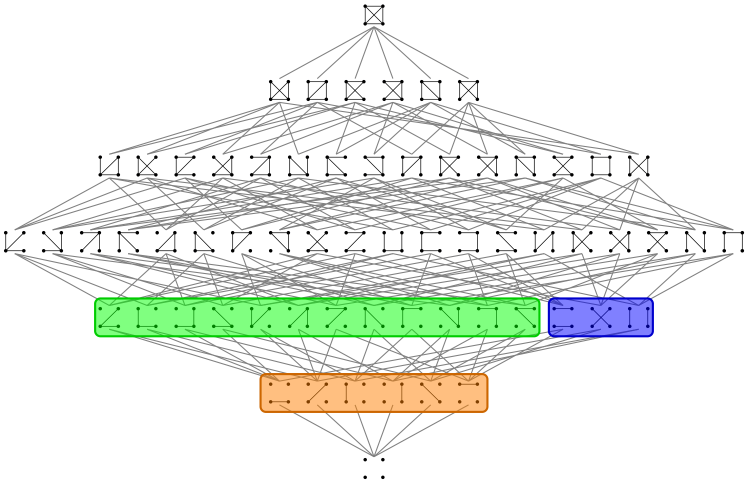
All graphs on 4 vertices (the haystack)



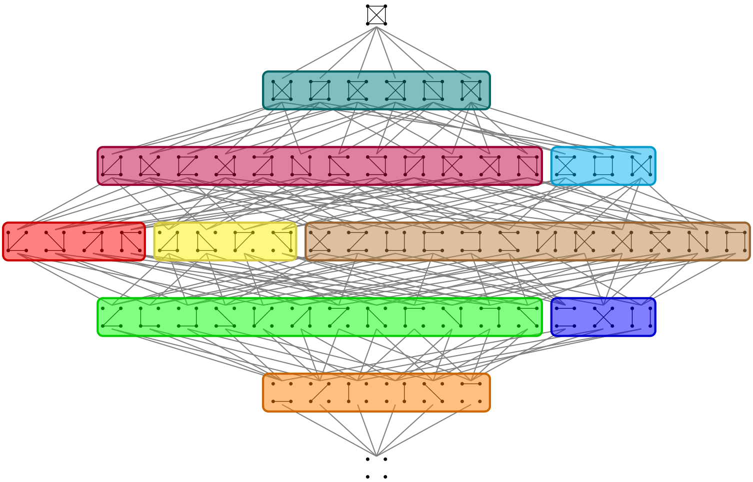
All graphs on 4 vertices (the haystack)



All graphs on 4 vertices (the haystack)



All graphs on 4 vertices (the haystack)



Isomorphism (in theory)

The decision problem

Graph Isomorphism
Instance: Graphs G and H
Question: Is G isomorphic to H?

is in the complexity class NP because if you are given the bijection it is easy to confirm that it is an isomorphism.

Graph Isomorphism is *not known* to be in P and *not known* to be NP-complete — it is a promising candidate for the elusive "intermediate" computational problem that would show that P $\neq$ NP.

Isomorphism (in practice)

In practice, most graph isomorphism programs do not directly test pairs of graphs, but instead *canonically label* individual graphs.

A *canonical labelling function* is a function

$$c : \mathcal{G} \to \mathcal{G}$$

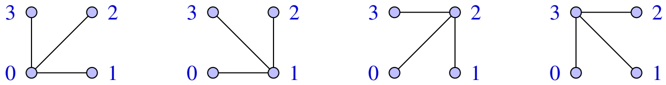
such that for all graphs G, H we have $c(G) \cong G$ and

$$G \cong H \iff c(G) = c(H).$$

A canonical labelling algorithm distinguishes *one member* of each isomorphism class as the *canonical representative* of that class.

Max-lex canonical form

Represent each graph in an isomorphism class as a binary string of length $\binom{n}{2}$ indicating which edges are present, and take the graph with the *lexicographically largest* string as the canonical representative.

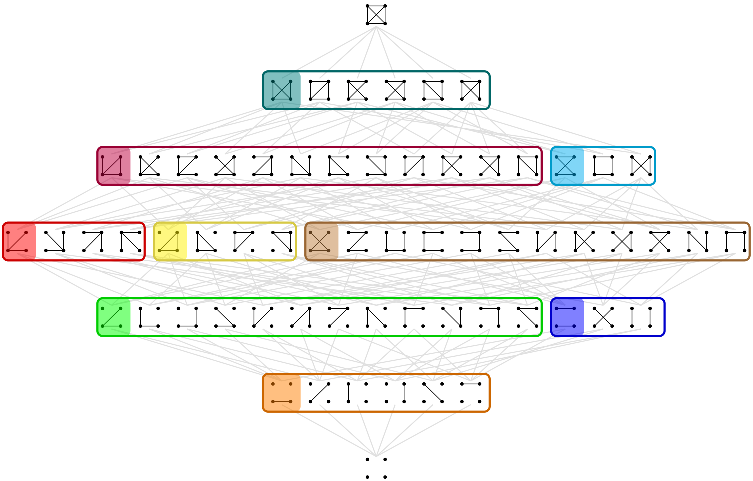


01	02	03	12	13	23
1	1	1	0	0	0
1	0	0	1	1	0
0	1	0	1	0	1
0	0	1	0	1	1

✓

This canonical labelling is *easy* to understand, but *hard* to compute.

THE NEEDLES IN THE HAYSTACK



A naive edge-addition algorithm

Initialise $\mathcal{L}_0$ be the list of all graphs on 4 vertices with no edges, and then for each $k \geqslant 0$, create $\mathcal{L}_{k+1}$ from $\mathcal{L}_k$.

For each graph $G \in \mathcal{L}_k$:

- Add a new edge to G in *all possible ways*,
- *Canonically label* each of the $(k+1)$-edge graphs that arise,
- Add each canonically-labelled graph to $\mathcal{L}_{k+1}$ if and only if it is *not already there*.

Then $\mathcal{L}_{k+1}$ contains—exactly once each—every canonically-labelled graph on $k+1$ edges.

Constructing huge lists

Two factors limit the size of database that can reasonably be generated by the naive algorithm.

- Time spent *canonically labelling* graphs
- Time/space spent *comparing* canonically-labelled graphs

Although equality checks are very quick, a list of N graphs requires at least $\binom{N}{2}$ equality tests.

If N is in the *billions* then this approach can never work.

Ron Read (and independently I. A. Faradžev) came up with the idea of an algorithm that *never performs pairwise comparisons*.

They devised a way to structure a construction algorithm so that every output is non-isomorphic to every other output—memorably encapsulated by the phrase *every one a winner*.†

The fundamental idea is to perform the search *entirely within* the subset of graphs that are canonically-labelled.

---

†Ron liked interesting paper titles such as “Is the null graph a pointless concept" discussing the pros and cons of allowing a graph to have no vertices.

Canonical labelling

There are a number of programs available that can canonically label large graphs, including `nauty`, `Traces`, `bliss`.

As a canonical labelling program can test graph isomorphism, it is *as hard* or *harder* than graph isomorphism.

Performance is very graph-dependent, but I have used `Traces` to canonically label a vertex-transitive 10-regular graph on 76 million vertices.

A naive algorithm

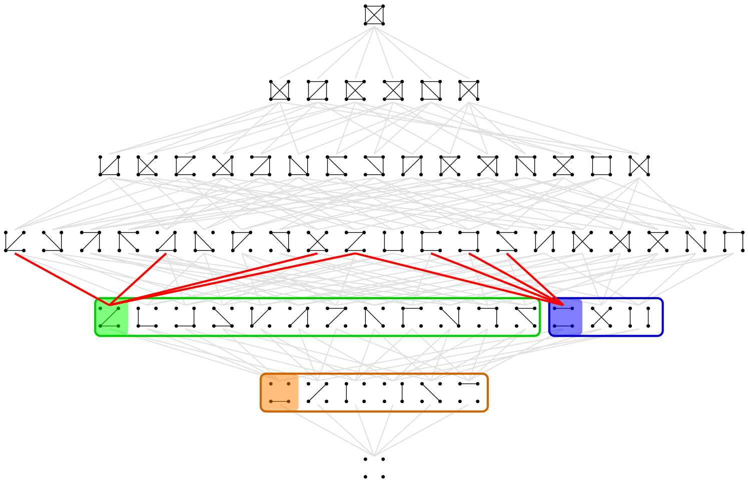
Initialise $\mathcal{L}_0$ be the list of all graphs on 4 vertices with no edges, and then for each $k \geqslant 0$, create $\mathcal{L}_{k+1}$ from $\mathcal{L}_k$.

For each graph $G \in \mathcal{L}_k$:

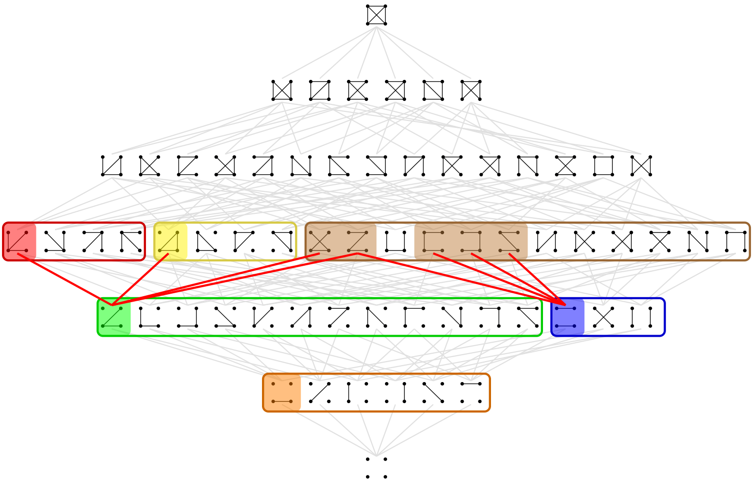
- Add a new edge to G in all possible ways,
- Canonically label each of the $(k+1)$-edge graphs that arise,
- Add each canonically-labelled graph to $\mathcal{L}_{k+1}$ if and only if it is *not already there*.

Then $\mathcal{L}_{k+1}$ contains—exactly once each—every canonically-labelled graph on $k+1$ edges.

NAIVE ALGORITHM $\mathcal{L}_2 \to \mathcal{L}_3$



NAIVE ALGORITHM $\mathcal{L}_2 \to \mathcal{L}_3$



Constructing huge lists

Two factors limit the size of database that can reasonably be generated by the naive algorithm.

- Time spent *canonically labelling* graphs
- Time/space spent *comparing* canonically-labelled graphs

Although equality checks are very quick, a list of N graphs requires at least $\binom{N}{2}$ equality tests.

If N is in the *billions* then this approach can never work.

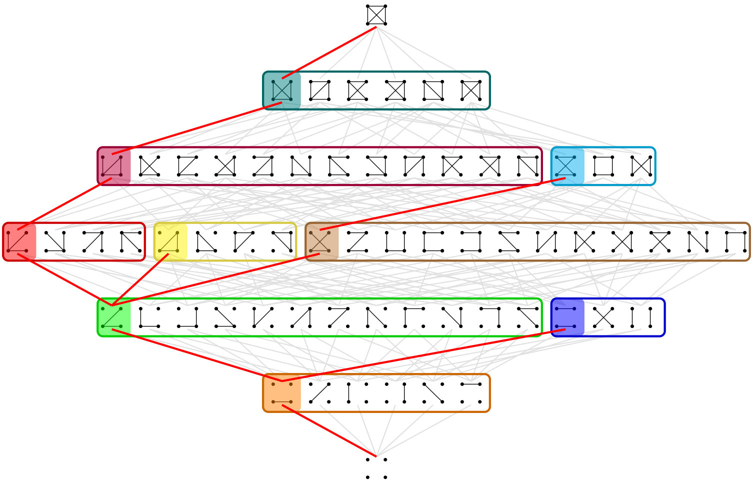
Every one a winner

Ron Read (and independently I. A. Faradžev) came up with the idea of an algorithm that *never performs pairwise comparisons*.

They devised a way to structure the construction algorithm so that every output is non-isomorphic to every other output—memorably encapsulated by the phrase *every one a winner*.

The fundamental idea is to perform the search *entirely within* the subset of graphs that are canonically-labelled.

Read-Faradžev style orderly



The orderly algorithm

For each graph $G \in \mathcal{L}_k$:

- (Augment) Add a new *last edge* e to G in all possible ways.
- (Test) Accept $G + e$ into $\mathcal{L}_{k+1}$ if it is *already canonically labelled*, and otherwise reject it.

Test for inclusion into $\mathcal{L}_{k+1}$ becomes a *single-graph test* of canonicity.

It works because the recipe *"delete the last edge"* defines a tree on the set of canonically-labelled graphs.

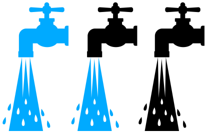
The algorithm *explores / constructs* the tree "upwards" starting from the empty graph.

ADVANTAGES

The search tree can be traversed in a depth-first (backtrack) fashion, dramatically reducing the total *time* and *space* required.

The search tree can be partitioned into *arbitrarily many* subtrees.

- Each part is totally independent of the others
- Each part can run on a *different* thread / core / chip or computer
- Graphs produced can be counted / examined and then *discarded*



Hierarchical canonical labelling

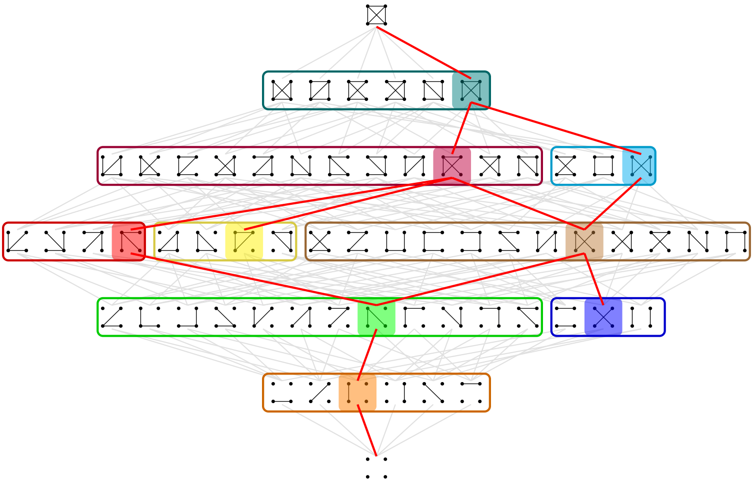
This works because the max-lex canonical labelling is *hierarchical*, so the recipe "*remove the last edge*" produces a smaller canonically-labelled graph.

$$\texttt{110100} \rightarrow \texttt{110000}$$

In particular, every canonically labelled graph can be obtained by *augmenting* a smaller canonically labelled graph.

Unfortunately the canonical labellings found by fast algorithms such as `nauty` and `Traces` *do not* have this hierarchical property.

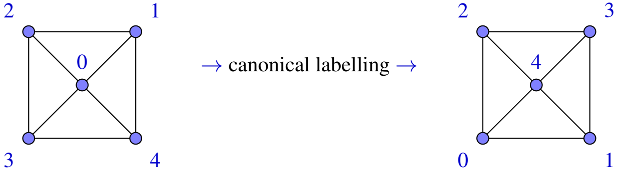
THE PROBLEM



Canonical destruction

Brendan McKay explained how to use an *arbitrary* canonical labelling function in such an algorithm.

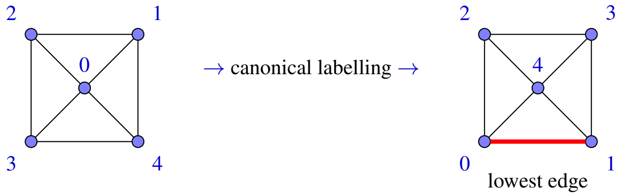
He focusses on *isomorphism classes* not *individual graphs*.



Canonical destruction

Brendan McKay explained how to use an *arbitrary* canonical labelling function in such an algorithm.

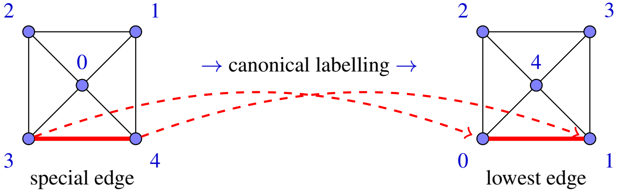
He focusses on *isomorphism classes* not *individual graphs*.



Canonical destruction

Brendan McKay explained how to use an *arbitrary* canonical labelling function in such an algorithm.

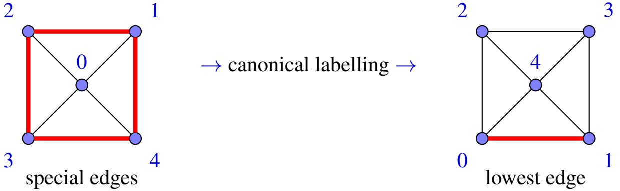
He focusses on *isomorphism classes* not *individual graphs*.



Canonical destruction

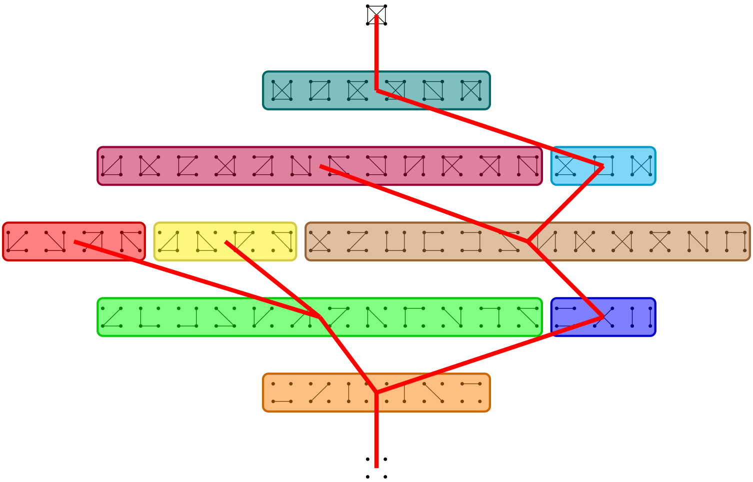
Brendan McKay explained how to use an *arbitrary* canonical labelling function in such an algorithm.

He focusses on *isomorphism classes* not *individual graphs*.



The recipe *"delete a special edge"* defines a tree on the set of isomorphism classes.

Canonical search tree



Canonical Augmentation

The algorithm repeatedly *augments* a graph G (by adding an edge e) and then *accepts* or *rejects* the augmented graph $G + e$.

Key Idea

Accept $G + e$ *if and only if* e is a special edge of $G + e$.

So a graph will only be accepted if the augmentation was consistent with the canonical search tree — if it was a *canonical augmentation*.

If G and H are non-isomorphic and $G + e$ and $H + f$ are both accepted, then $G + e$ is not isomorphic to $H + f$.

Avoiding isomorphs

When G is augmented we need to consider augmenting by *every* non-edge.

If e and f are *equivalent* non-edges under the *automorphism group* of G, then $G+e$ and $G+f$ will definitely be isomorphic.

As canonical labelling algorithms such as `nauty/Traces` compute the automorphism group of a graph, the augmentation step can easily be modified to avoid this.

Canonical construction path algorithm

Brendan McKay calls this the *canonical construction path* algorithm (and prefers to reserve the word "orderly" for Read-style orderly).

To reiterate:

- The *augmentation step* produces pairwise non-isomorphic graphs

  Graphs obtained by augmenting G are pairwise non-isomorphic.

- The *canonicity check* tests that the newly added edge is (equivalent to) the special edge

  Graphs obtained by augmenting G are not isomorphic to those obtained by augmenting H.

More generally

Applies to *far more* than just graphs—if you have any set of combinatorial objects and you can find:

- A isomorph-invariant *canonical reduction* from a larger object to a smaller one.
- A reverse method of *augmenting* a smaller object to a set of pairwise non-isomorphic larger ones.

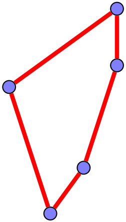
then you can run the canonical construction path algorithm.

For example, Brendan's graph generation program `geng` uses *vertex* addition/deletion.

```
00013890@DEP52010 nauty27r1 % geng -u 10
>A geng -d0D9 n=10 e=0-45
>Z 12005168 graphs generated in 3.36 sec
```

The first example

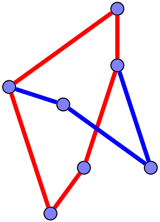
For example, my thesis[†] used a somewhat clunky ad-hoc CCP algorithm to construct cubic graphs *ear-by-ear*.



[†]heavily influenced by Brendan, of course

THE FIRST EXAMPLE

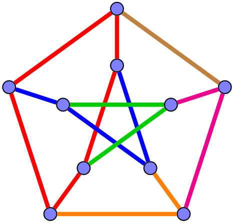
For example, my thesis[†] used a somewhat clunky ad-hoc CCP algorithm to construct cubic graphs *ear-by-ear*.



[†] heavily influenced by Brendan, of course

The first example

For example, my thesis[†] used a somewhat clunky ad-hoc CCP algorithm to construct cubic graphs *ear-by-ear*.



According to Brinkmann, Goedgebeur and Van Cleemput

> This method already contains all ingredients that make later algorithms using the canonical construction path method so efficient.

[†]heavily influenced by Brendan, of course

A COMMON SITUATION

Many combinatorial generation problems are of the form:

> *Given a graph* X *find one representative from each* $\mathrm{Aut}(X)$*-orbit of* subsets *of* $V(X)$.

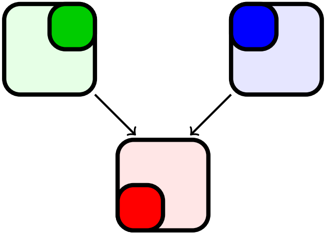
> *Given a* permutation group $G \subseteq \mathrm{Sym}(\Omega)$ *find one representative from each* G*-orbit of* subsets *of* Ω.

Use `nauty/Traces` in the first case, or the GAP functions `SmallestImageSet`, `MinimalImage` or `CanonicalImage` in the second.

Jump To End

Canonical Labelling

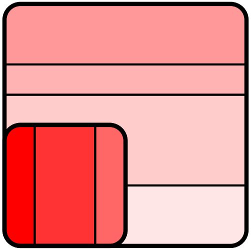
Programs such as `nauty/Traces` can take a *partitioned graph*, and *relabel it* in an isomorph-invariant fashion.



Two sets of the same size are *isomorphic* if the canonically labelled graphs are *identical*.

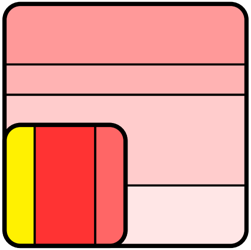
THE SPECIAL ORBIT

In addition, `nauty/Traces` gives the *orbits* of $\mathrm{Aut}(\Gamma)_X$, i.e., the automorphisms of Γ that fix X.



The special orbit

In addition, `nauty/Traces` gives the *orbits* of $\mathrm{Aut}(\Gamma)_X$, i.e., the automorphisms of Γ that fix X.



In particular, this allows us to identify a *particular orbit* of $\mathrm{Aut}(\Gamma)_X$ in a *labelling-independent* way—this is the *special orbit*.

Binary Matroids

A *simple binary matroid* is a subset $B \subseteq (\mathbb{Z}_2^n)^*$.

For example,

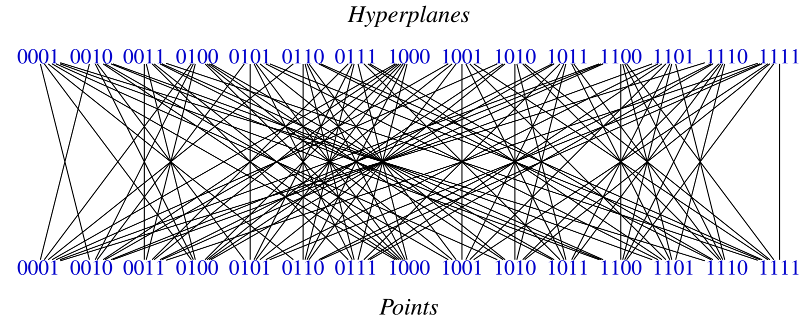
$$K_4 = \{001, 010, 011, 100, 101, 110\}$$

Two binary matroids B_1 and B_2 are isomorphic if there is an invertible matrix $A \in \mathrm{GL}(n, 2)$ such that $AB_1 = B_2$.

A catalogue of *all binary matroids* requires one representative of each $\mathrm{GL}(n, 2)$-orbit on subsets of $(\mathbb{Z}_2^n)^*$.

What graph to use?

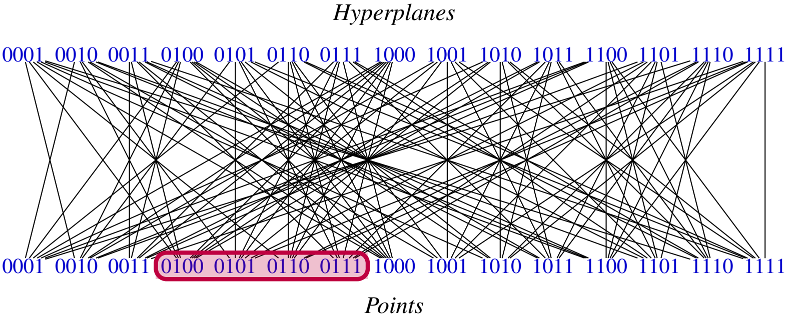
Construct the point/hyperplane incidence graph Γ of $\mathrm{PG}(n-1, 2)$



This has automorphism group $\mathrm{GL}(n, 2) : C_2$.

What graph to use?

Construct the point/hyperplane incidence graph Γ of $\mathrm{PG}(n-1, 2)$



This has automorphism group $\mathrm{GL}(n, 2) : C_2$.

Totals

Rank	Number
3	10
4	46
5	1372
6	475499108
7	10383979818409945095779948
8	10825608503765473087803384381127710579846422820261084889808

This is A000613 “*Number of equivalence classes of boolean functions*” in Sloane’s OEIS, the low A-number reflecting the fundamental nature of this sequence.

45ACC @ UWA : Dec 11 – Dec 15

https://45acc.github.io.

Water is good, air is better, but sunlight is best of all —Arnold Rikli, Bled.