Practical Graph Isomorphism, II

Brendan McKay

Australian National University

Adolfo Piperno

University of Rome

Isomorphism and Automorphism

We have some finite objects consisting of some finite sets and some relations on them.

An isomorphism between two objects is a bijection between their sets so that the relations are preserved. Hopefully, isomorphism is an equivalence relation on the set of all objects.

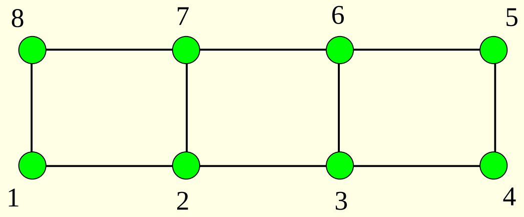
Isomorphism and Automorphism

We have some finite objects consisting of some finite sets and some relations on them.

An isomorphism between two objects is a bijection between their sets so that the relations are preserved. Hopefully, isomorphism is an equivalence relation on the set of all objects.

Isomorphisms from an object to itself are automorphisms. If we didn't do anything silly, the set of automorphisms forms a group under composition.

Of course we call it the automorphism group.



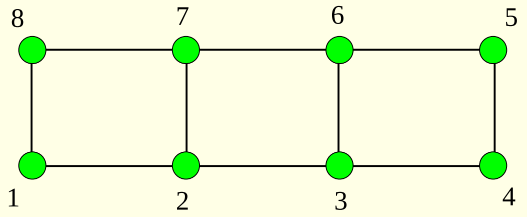
Isomorphism and Automorphism

We have some finite objects consisting of some finite sets and some relations on them.

An isomorphism between two objects is a bijection between their sets so that the relations are preserved. Hopefully, isomorphism is an equivalence relation on the set of all objects.

Isomorphisms from an object to itself are automorphisms. If we didn't do anything silly, the set of automorphisms forms a group under composition.

Of course we call it the automorphism group.



(1)

(1 4)(2 3)(5 8)(6 7)

(1 8)(2 7)(3 6)(4 5)

(1 5)(2 6)(3 7)(4 8)

Ubiquity of graph isomorphism

Very many isomorphism problems can be efficiently expressed as graph isomorphism problems. For convenience, we use coloured graphs, in which the vertices (and possibly the edges) are coloured. Isomorphisms must preserve vertex and edge colour.

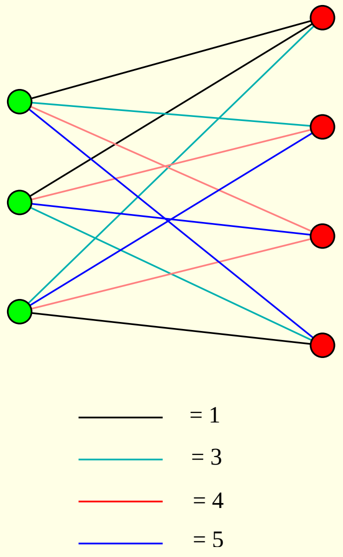
$$\begin{pmatrix} 1 & 3 & 5 & 4 \\ 1 & 5 & 4 & 3 \\ 3 & 4 & 5 & 1 \end{pmatrix}$$

Say two matrices are isomorphic
if one can be obtained from the
other by permuting rows and columns.

Ubiquity of graph isomorphism

Very many isomorphism problems can be efficiently expressed as graph isomorphism problems. For convenience, we use coloured graphs, in which the vertices (and possibly the edges) are coloured. Isomorphisms must preserve vertex and edge colour.

$$\begin{pmatrix} 1 & 3 & 5 & 4 \\ 1 & 5 & 4 & 3 \\ 3 & 4 & 5 & 1 \end{pmatrix}$$



Say two matrices are isomorphic if one can be obtained from the other by permuting rows and columns.

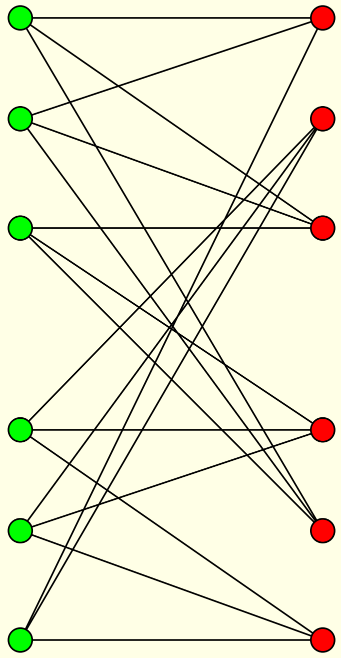
Hadamard equivalence

$$\begin{pmatrix} 1 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & -1 & 1 \end{pmatrix}$$

Say isomorphism of two ± 1 matrices
allows permutation and negation
of rows and columns.

Hadamard equivalence

$$\begin{pmatrix} 1 & -1 & 1 \\ 1 & -1 & 1 \\ -1 & -1 & 1 \end{pmatrix}$$



Say isomorphism of two ± 1 matrices allows permutation and negation of rows and columns.

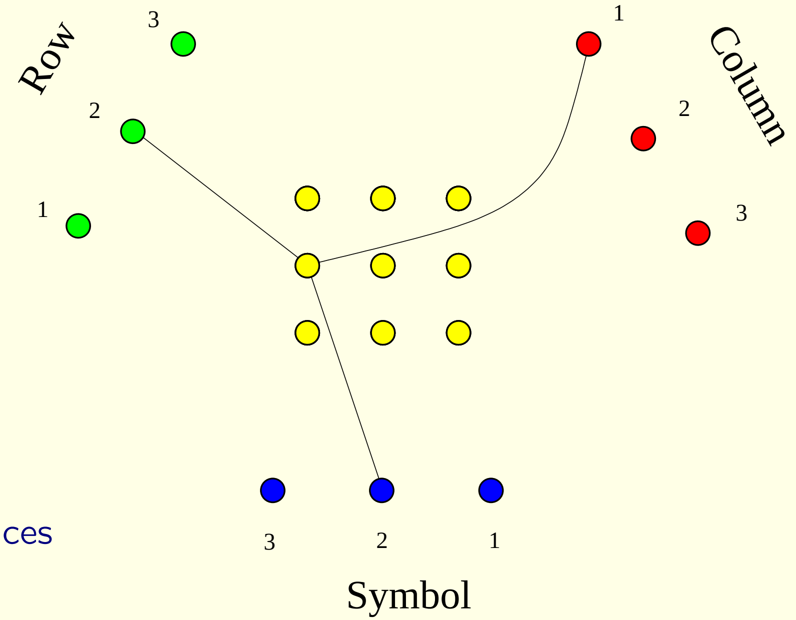
Isotopy

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Say isomorphism of two matrices allows permutation of rows, columns and symbols.

Isotopy

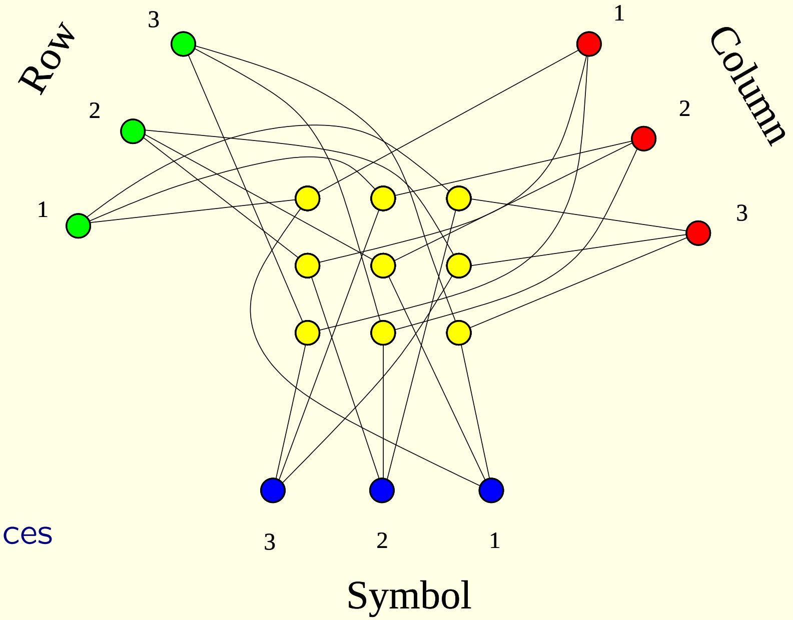
$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$



Say isomorphism of two matrices allows permutation of rows, columns and symbols.

Isotopy

$$\begin{pmatrix} 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$



Say isomorphism of two matrices allows permutation of rows, columns and symbols.

Group isomorphism

$$\begin{bmatrix} e & g_1 & g_2 & g_3 \\ g_1 & e & g_3 & g_2 \\ g_2 & g_3 & e & g_1 \\ g_3 & g_2 & g_1 & e \end{bmatrix}$$

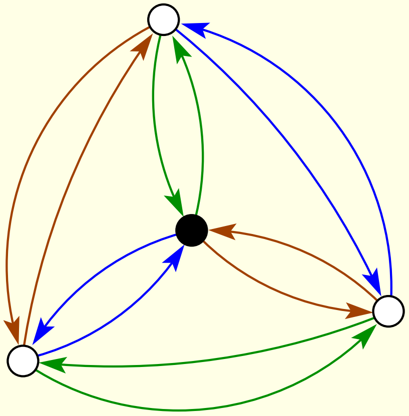
The group elements are edge colours and vertices.

The black vertex is the identity element.

Preserve vertex colours but allow permutation of edge colours.

Group isomorphism

$$\begin{bmatrix} e & g_1 & g_2 & g_3 \\ g_1 & e & g_3 & g_2 \\ g_2 & g_3 & e & g_1 \\ g_3 & g_2 & g_1 & e \end{bmatrix}$$



The group elements are edge colours and vertices.

The black vertex is the identity element.

Preserve vertex colours but allow permutation of edge colours.

Permutation equivalence of linear codes

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Permutation equivalence of linear codes

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad ?$$

Suppose isomorphism of two 0-1 matrices means that their columns can be permutated so that the rows generate the same vector space over $GF(2)$.

Nobody knows how to make an equivalent graph problem of size which is polynomial in the size of the generator matrix.

Theoretical complexity

The graph isomorphism problem is:

> **GI**: Given two graphs, determine whether they are isomorphic.

- The fastest known algorithm for GI is $2^{O(\sqrt{(n \log n)})}$ time; no polynomial time algorithm is known (despite many claims).
- GI is not known to be NP-complete.
- GI is not known to be in co-NP.
- There are polynomial time algorithms for many special classes of graphs (bounded degree, bounded genus, classes with excluded minors or topological subgraphs). Few of these are practical, planar graphs being a notable exception.

It is possible that group isomorphism is easier than GI and that permutation equivalence of linear codes is harder than GI. But nobody can prove it.

Canonical labelling

Given a class of objects, and a definition of "isomorphism", we can choose one member of each isomorphism class and call it the canonical member of the class.

The process of finding the canonical member of the isomorphism class containing a given object is called "canonical labelling".

Canonical labelling

Given a class of objects, and a definition of “isomorphism”, we can choose one member of each isomorphism class and call it the canonical member of the class.

The process of finding the canonical member of the isomorphism class containing a given object is called “canonical labelling”.

Two labelled objects which are isomorphic become identical when they are canonically labelled. Since identity of objects is usually far easier to check than isomorphism, canonical labelling is the preferred method of sorting a collection of objects into isomorphism classes.

Canonical labelling

Given a class of objects, and a definition of “isomorphism”, we can choose one member of each isomorphism class and call it the canonical member of the class.

The process of finding the canonical member of the isomorphism class containing a given object is called “canonical labelling”.

Two labelled objects which are isomorphic become identical when they are canonically labelled. Since identity of objects is usually far easier to check than isomorphism, canonical labelling is the preferred method of sorting a collection of objects into isomorphism classes.

A possible definition of the canonical member of an isomorphism class would be the member which maximises some linear representation (such as a list of edges).

In practice, most programs compute a canonical member which is easy for computers to find rather than easy for humans to define.

Software for graph isomorphism

This task was already described as a disease in 1970, and programs still appear regularly. The most successful programs still supported are:

- `nauty` (McKay, 1976–) canonical label and automorphism group
- `VF2` (Cordella, Foggia, Sansone and Vento, 1999–) comparison of two graphs
- `saucy` (Darga, Sakallah and Markov, 2004–) automorphism group
- `bliss` (Juntilla and Kaski, 2007–) canonical label and automorphism group
- `Traces` (Piperno, 2008–) canonical label and automorphism group
- `conauto` (López-Presa, Anta and Chiroque, 2009–) automorphism group and comparison of two graphs

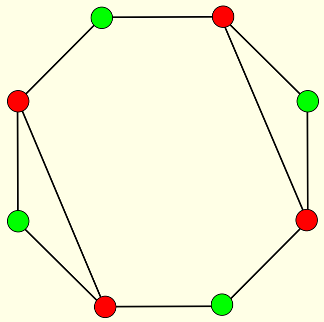
Many similar principles appear in these programs. `saucy` and `bliss` began life as reimplementations of `nauty` but have since diverged.

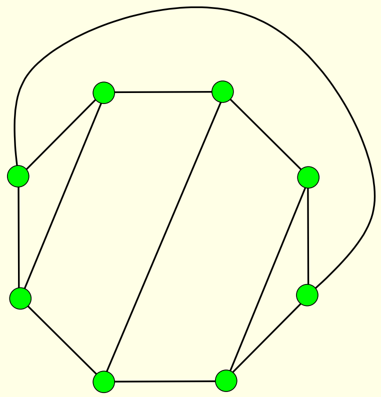
Partition refinement

A key concept used by `nauty` is partition refinement. (“partition” = “colouring”)

An equitable partition is one where every two vertices of the same colour are adjacent to the same number of vertices of each colour.

Two examples of equitable partitions:



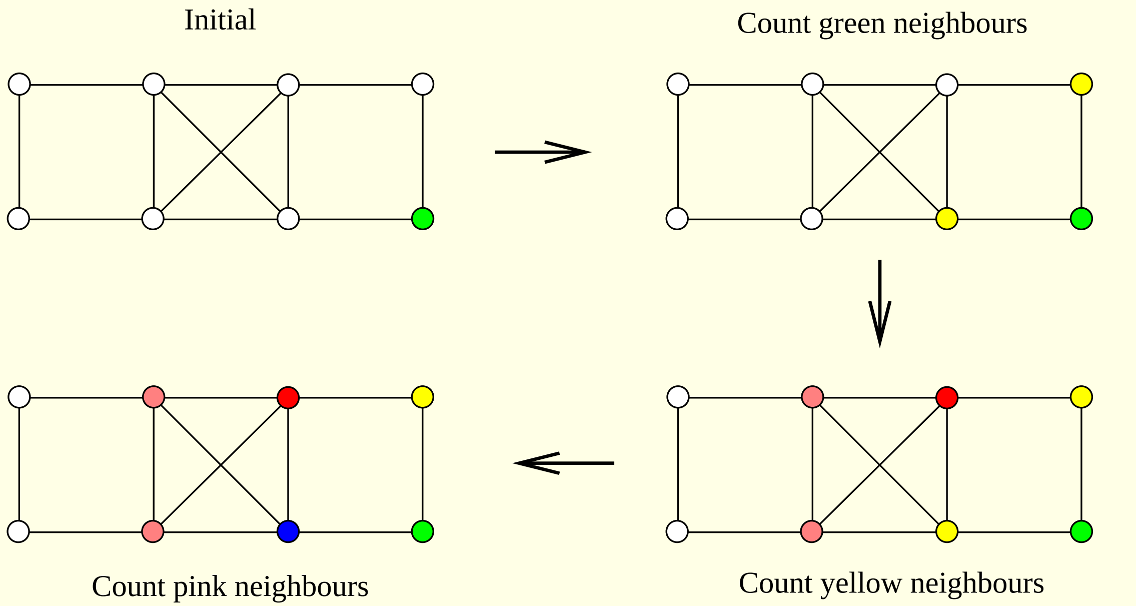


Refinement of a partition means to subdivide its cells.

Given a partition (colouring) π, there is a unique equitable partition that is a refinement of π and has the least number of colours.

Refinement of a partition means to subdivide its cells.

Given a partition (colouring) π, there is a unique equitable partition that is a refinement of π and has the least number of colours.



Search tree

All competitive isomorphism programs generate a tree based on partition refinement. The nodes of the tree correspond to partitions (colourings).

The root of the tree corresponds to the initial colouring, refined.

If a node corresponds to a discrete partition (each vertex with a different colour), it has no children and is a leaf.

Otherwise we choose a colour used more than once (the target cell), individualize one of those vertices by giving it a new unique colour, and refine to get a child.

Search tree

All competitive isomorphism programs generate a tree based on partition refinement. The nodes of the tree correspond to partitions (colourings).

The root of the tree corresponds to the initial colouring, refined.

If a node corresponds to a discrete partition (each vertex with a different colour), it has no children and is a leaf.

Otherwise we choose a colour used more than once (the target cell), individualize one of those vertices by giving it a new unique colour, and refine to get a child.

Each leaf lists the vertices in some order (since colours have a predefined order), so it corresponds to a labelling of the graph.

Search tree

All competitive isomorphism programs generate a tree based on partition refinement. The nodes of the tree correspond to partitions (colourings).

The root of the tree corresponds to the initial colouring, refined.

If a node corresponds to a discrete partition (each vertex with a different colour), it has no children and is a leaf.

Otherwise we choose a colour used more than once (the target cell), individualize one of those vertices by giving it a new unique colour, and refine to get a child.

Each leaf lists the vertices in some order (since colours have a predefined order), so it corresponds to a labelling of the graph.

If we have two leaves giving the same labelled graph, we have found an automorphism.

If we define an order on labelled graphs, such as lexicographic order, then the greatest labelled graph corresponding to a leaf is a canonical graph.

Search tree

All competitive isomorphism programs generate a tree based on partition refinement. The nodes of the tree correspond to partitions (colourings).

The root of the tree corresponds to the initial colouring, refined.

If a node corresponds to a discrete partition (each vertex with a different colour), it has no children and is a leaf.

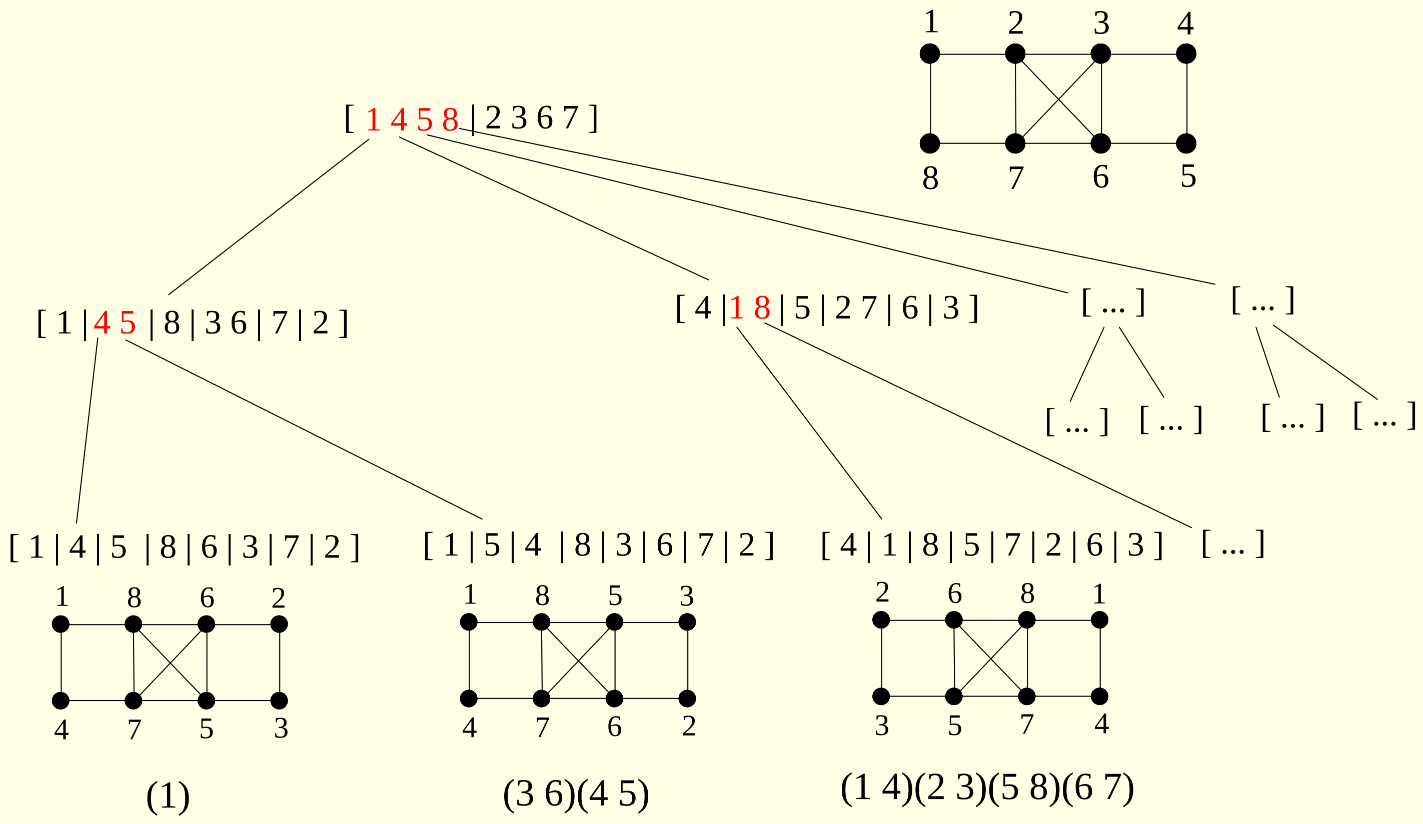
Otherwise we choose a colour used more than once (the target cell), individualize one of those vertices by giving it a new unique colour, and refine to get a child.

Each leaf lists the vertices in some order (since colours have a predefined order), so it corresponds to a labelling of the graph.

If we have two leaves giving the same labelled graph, we have found an automorphism.

If we define an order on labelled graphs, such as lexicographic order, then the greatest labelled graph corresponding to a leaf is a canonical graph.

However the tree can be much too large to generate completely.


[1 4 5 8 | 2 3 6 7]
[1 | 4 5 | 8 | 3 6 | 7 | 2]
[4 | 1 8 | 5 | 2 7 | 6 | 3]
[...]
[...]
[...]
[...]
[...]
[...]
[1 | 4 | 5 | 8 | 6 | 3 | 7 | 2]
[1 | 5 | 4 | 8 | 3 | 6 | 7 | 2]
[4 | 1 | 8 | 5 | 7 | 2 | 6 | 3]
[...]
(1)
(3 6)(4 5)
(1 4)(2 3)(5 8)(6 7)

Node invariants

A node invariant is a value $\phi(\nu)$ attached to each node in the tree that depends only on the combinatorial properties of ν and its ancestors in the search tree (not on the labelling of the graph).

Moreover, the order of node invariants must be preserved by descendants:
Let ν_1, ν_2 be nodes at the same level in the search tree. If ν_1' is a descendant of ν_1, and ν_2' is a descendant of ν_2, then

$$\phi(\nu_1) < \phi(\nu_2) \implies \phi(\nu_1') < \phi(\nu_2').$$

Node invariants

A node invariant is a value $\phi(\nu)$ attached to each node in the tree that depends only on the combinatorial properties of ν and its ancestors in the search tree (not on the labelling of the graph).

Moreover, the order of node invariants must be preserved by descendants:
Let ν_1, ν_2 be nodes at the same level in the search tree. If ν_1' is a descendant of ν_1, and ν_2' is a descendant of ν_2, then

$$\phi(\nu_1) < \phi(\nu_2) \implies \phi(\nu_1') < \phi(\nu_2').$$

For example,

$$\phi(\nu) = \bigl(c(\nu'), c(\nu''), \ldots, c(\nu)\bigr)$$

is a node invariant with lexicographic ordering, where $\nu', \nu'', \ldots, \nu$ is the path from the root of the tree to ν, and $c(\,)$ is the number of colours.

Node invariants

A node invariant is a value $\phi(\nu)$ attached to each node in the tree that depends only on the combinatorial properties of ν and its ancestors in the search tree (not on the labelling of the graph).

Moreover, the order of node invariants must be preserved by descendants:
Let ν_1, ν_2 be nodes at the same level in the search tree. If ν_1' is a descendant of ν_1, and ν_2' is a descendant of ν_2, then

$$\phi(\nu_1) < \phi(\nu_2) \implies \phi(\nu_1') < \phi(\nu_2').$$

For example,

$$\phi(\nu) = \big(c(\nu'), c(\nu''), \ldots, c(\nu)\big)$$

is a node invariant with lexicographic ordering, where $\nu', \nu'', \ldots, \nu$ is the path from the root of the tree to ν, and $c(\,)$ is the number of colours.

Let ϕ^* be the greatest node invariant of a leaf. Then the lexicographically greatest labelled graph corresponding to a leaf ν with $\phi(\nu) = \phi^*$ is a canonical graph.

Pruning operations

Node invariants, together with automorphisms, allow us to remove parts of the search tree without generating them.

1. If ν_1, ν_2 are nodes at the same level in the search tree and $\phi(\nu_1) < \phi(\nu_2)$, then no canonical labelling is descended from ν_1.
2. If ν_1, ν_2 are nodes at the same level in the search tree and $\phi(\nu_1) \neq \phi(\nu_2)$, then no labelled graph descended from ν_1 is the same as one descended from ν_2.
3. If ν_1, ν_2 are nodes with $\nu_2 = \nu_1^g$ for an automorphism g, then g maps the subtree descended from ν_1 onto the subtree descended from ν_2.

Pruning operations

Node invariants, together with automorphisms, allow us to remove parts of the search tree without generating them.

1. If ν_1, ν_2 are nodes at the same level in the search tree and $\phi(\nu_1) < \phi(\nu_2)$, then no canonical labelling is descended from ν_1.
2. If ν_1, ν_2 are nodes at the same level in the search tree and $\phi(\nu_1) \neq \phi(\nu_2)$, then no labelled graph descended from ν_1 is the same as one descended from ν_2.
3. If ν_1, ν_2 are nodes with $\nu_2 = \nu_1^g$ for an automorphism g, then g maps the subtree descended from ν_1 onto the subtree descended from ν_2.

Each of these observations enables us to prune parts of the search tree.
The hope is that the remaining parts will be small enough to compute.

Variations between programs

The competing programs vary from each other in ways that include:

1. Data structures
2. Strength of the refinement procedure
3. Order of traversal of the search tree
4. Means of discovering automorphisms
5. Processing of automorphisms

Stronger refinement — `nauty` “invariants”

Sometimes refinement to equitable partition is insufficient to separate vertices with clearly different combinatorial properties. Those properties can be used to make the refinement stronger.

Stronger refinement — `nauty` “invariants”

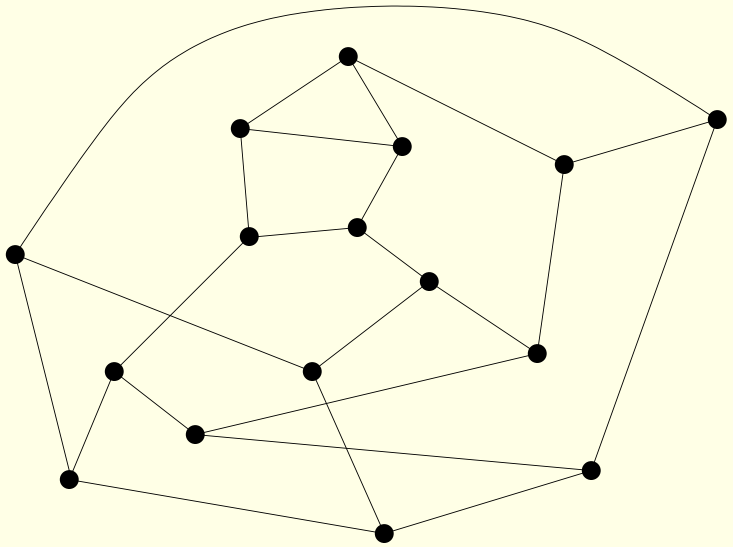
Sometimes refinement to equitable partition is insufficient to separate vertices with clearly different combinatorial properties. Those properties can be used to make the refinement stronger.

For example, we can count the number of 3-cycles and 4-cycles through each vertex:

Stronger refinement — `nauty` "invariants"

Sometimes refinement to equitable partition is insufficient to separate vertices with clearly different combinatorial properties. Those properties can be used to make the refinement stronger.

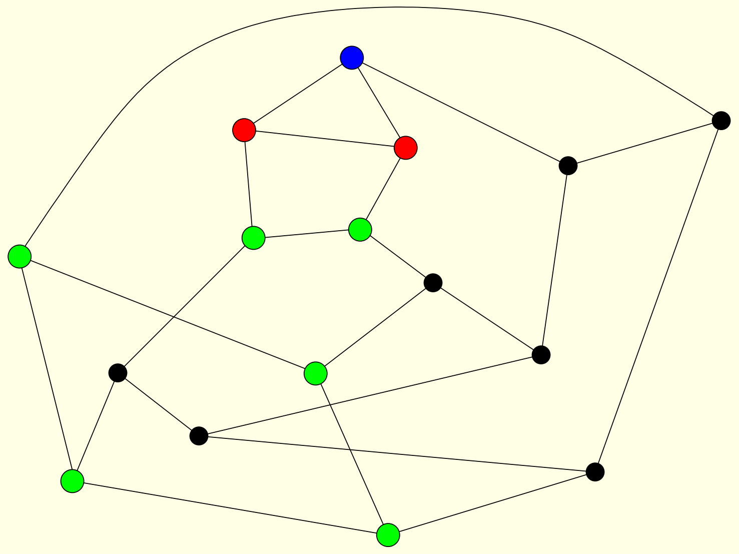
For example, we can count the number of 3-cycles and 4-cycles through each vertex:



Stronger refinement — `nauty` “invariants”

Sometimes refinement to equitable partition is insufficient to separate vertices with clearly different combinatorial properties. Those properties can be used to make the refinement stronger.

For example, we can count the number of 3-cycles and 4-cycles through each vertex:



Example of stronger refinement performance

Consider random quartic graphs with 1000 vertices.

• `nauty` (dense data structures)	5.6 seconds
• `nauty` (sparse data structures)	0.5 seconds
• `nauty` (count vertices at distance 2)	0.0014 seconds

Example of stronger refinement performance

Consider random quartic graphs with 1000 vertices.

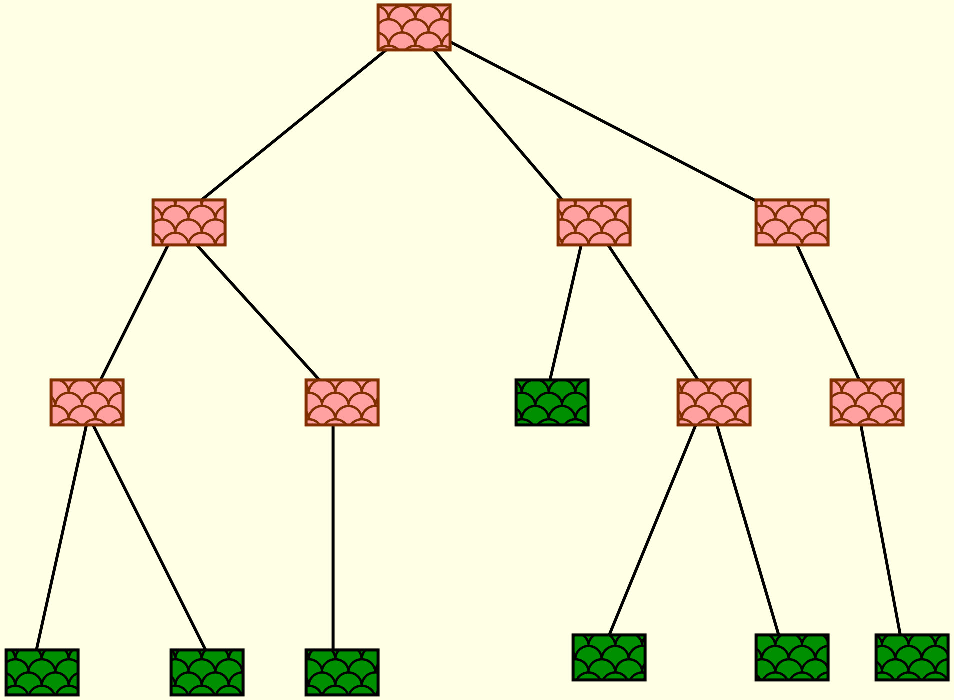
• `nauty` (dense data structures)	5.6 seconds
• `nauty` (sparse data structures)	0.5 seconds
• `nauty` (count vertices at distance 2)	0.0014 seconds

The big problem with these methods is that the best refinement technique varies from one graph class to another.

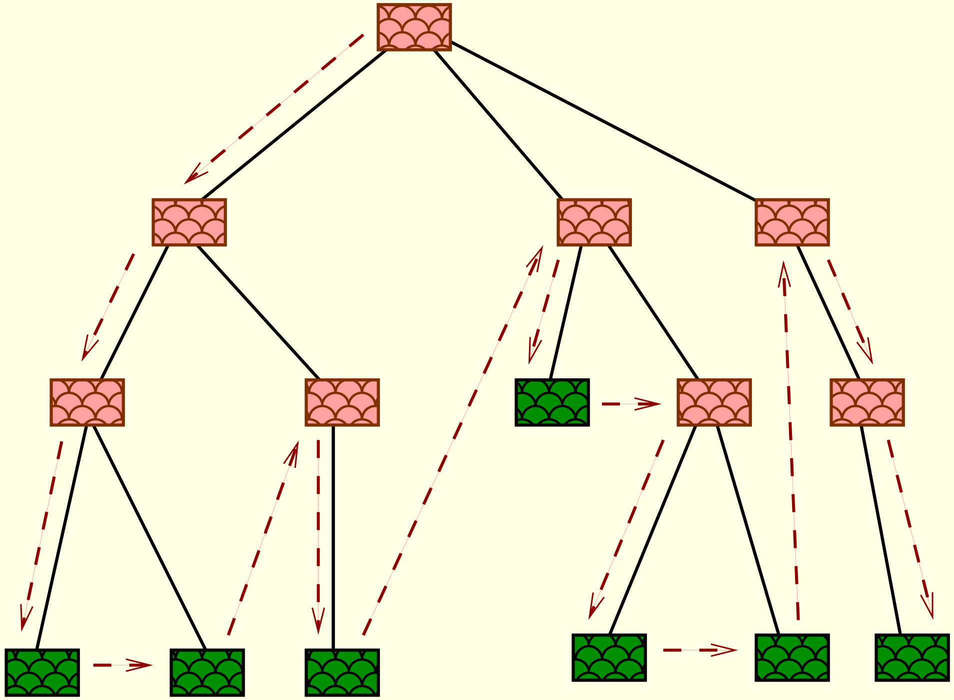
The user has to know which one to choose, which few do.

Tree traversal order — a Traces innovation

Tree traversal order — a `Traces` innovation

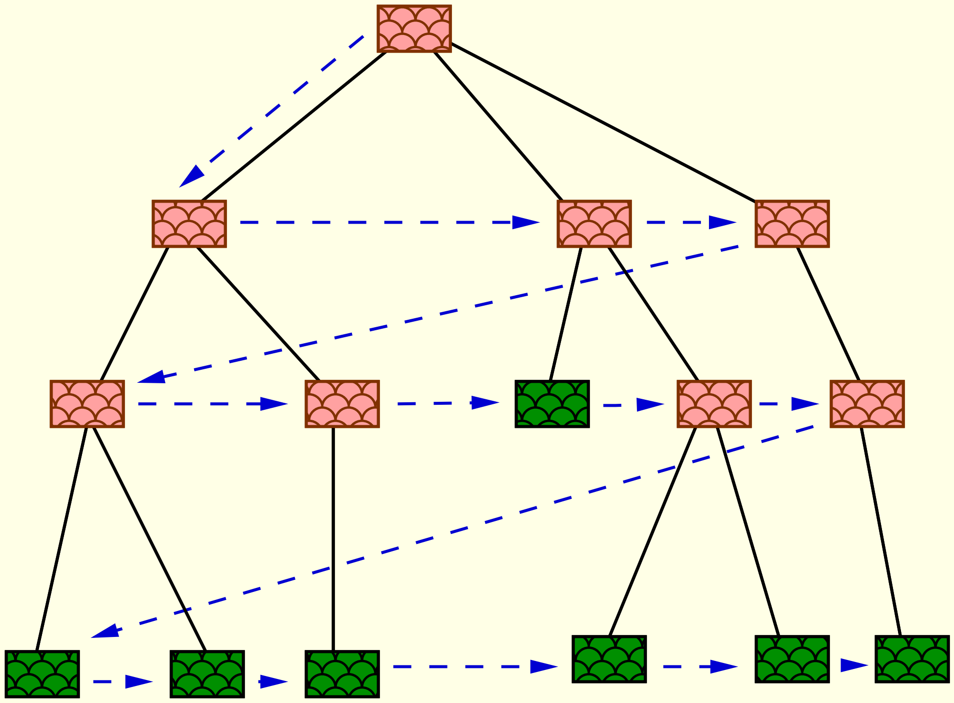


Tree traversal order — a Traces innovation



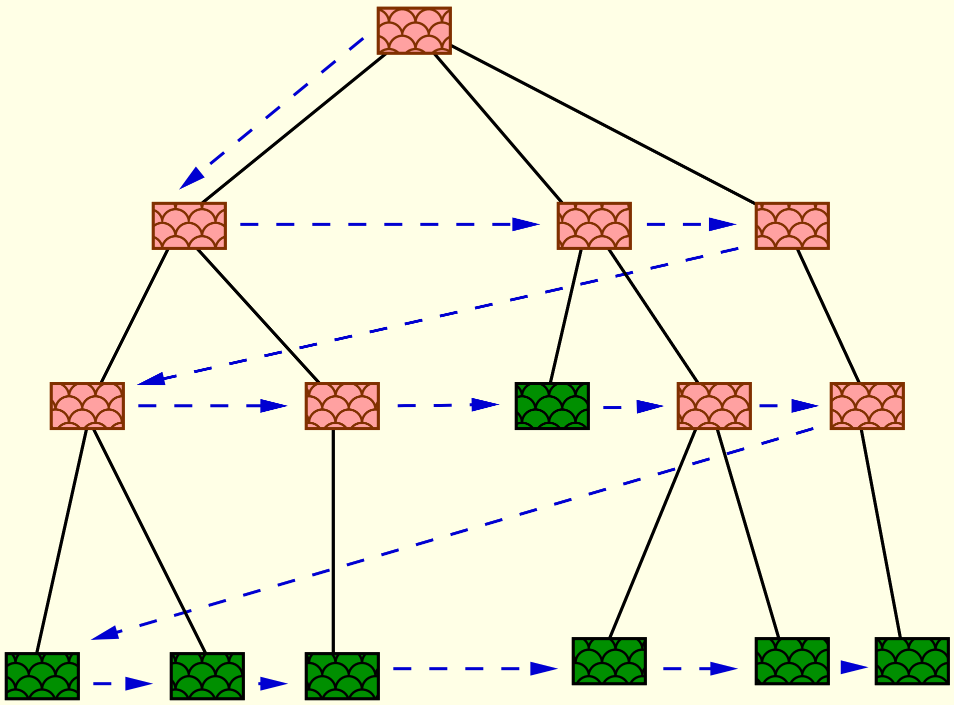
Classical order: depth-first search

Tree traversal order — a Traces innovation



Traces order: breadth-first search

Tree traversal order — a `Traces` innovation

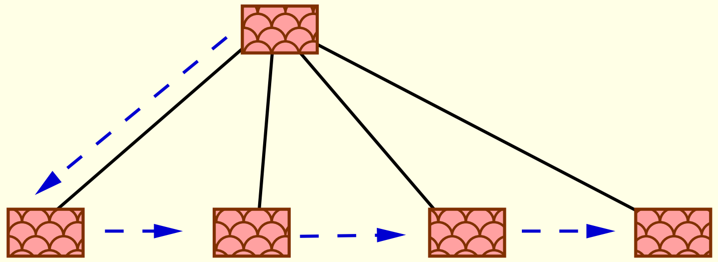


Traces order: breadth-first search

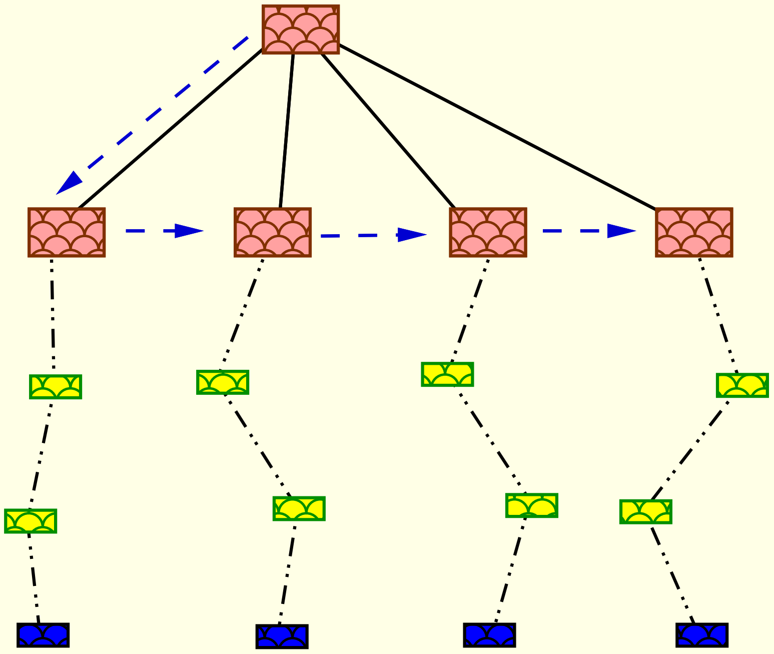
The problem with BFS is that automorphisms are discovered at leaves.

Tree traversal order — a `Traces` innovation

Tree traversal order — a Traces innovation



Tree traversal order — a Traces innovation



Traces: experimental paths

Experimental paths allow automorphism detection during breadth-first search.

Refinement traces – a Traces innovation

In all programs, for almost all graphs, most of the work involves partition refinement. However, the result of partition refinement is often thrown away because the node invariant of the resulting node is unsuitable.

Refinement traces – a `Traces` innovation

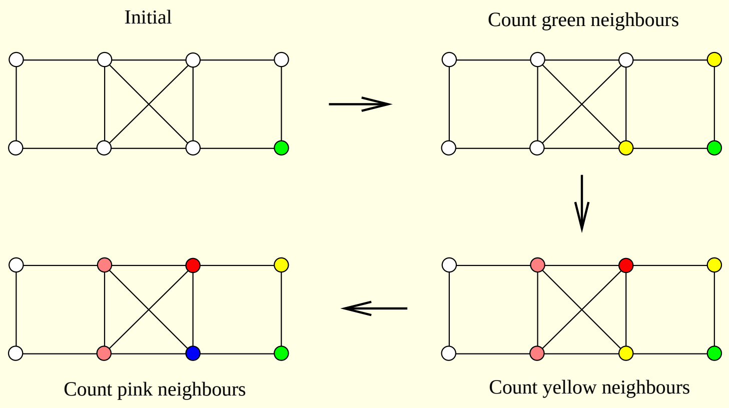
In all programs, for almost all graphs, most of the work involves partition refinement. However, the result of partition refinement is often thrown away because the node invariant of the resulting node is unsuitable.

To alleviate this problem, `Traces` defines the node invariant to be a vector whose components are computed incrementally during the refinement. Then the refinement can often be aborted early. (This extends a technique introduced by `bliss`.)

Refinement traces – a Traces innovation

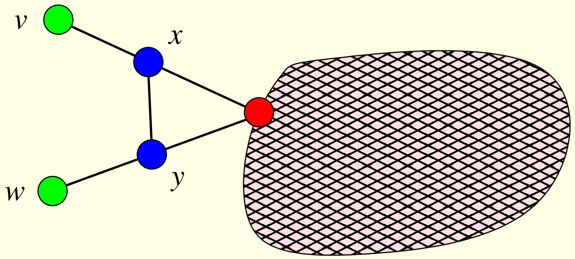
In all programs, for almost all graphs, most of the work involves partition refinement. However, the result of partition refinement is often thrown away because the node invariant of the resulting node is unsuitable.

To alleviate this problem, Traces defines the node invariant to be a vector whose components are computed incrementally during the refinement. Then the refinement can often be aborted early. (This extends a technique introduced by bliss.)



A possible trace (not Traces' trace) is the number of colours at each step: (2,3,5,6).

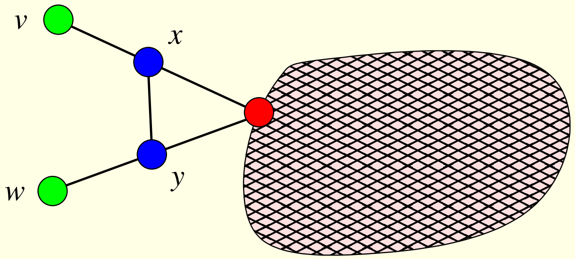
Sparse automorphism detection — a saucy innovation



equitable partition

(no other blue or green)

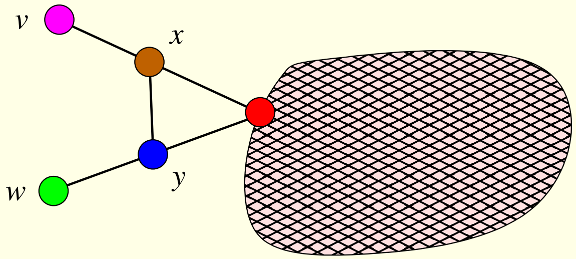
Sparse automorphism detection — a saucy innovation



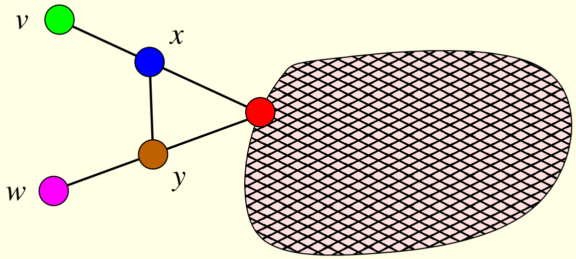
equitable partition

(no other blue or green)

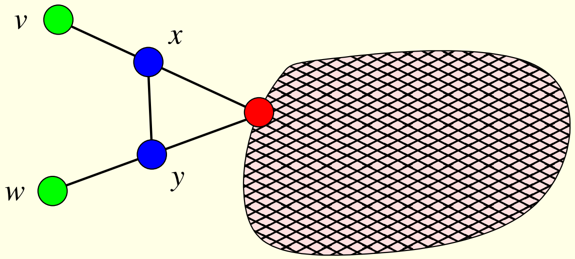
individualize *v*



individualize *w*



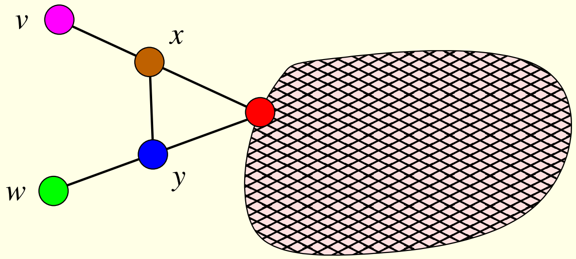
Sparse automorphism detection — a `saucy` innovation



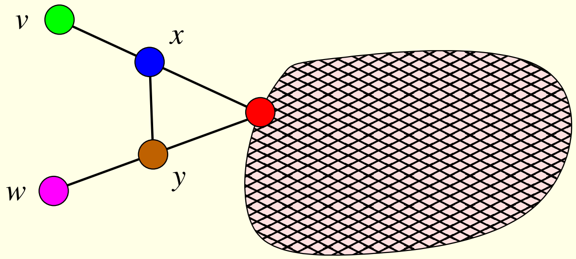
equitable partition

(no other blue or green)

individualize v



individualize w



This discovers the automorphism $(v\ w)(x\ y)$ without comparing leaves.

Automorphism group handling

Programs using depth-first search naturally produce automorphisms in the form of a base and strong generating set.

Other programs, like `Traces`, produce less predictable generators.

In order to perform automorphism-based pruning of the search tree, we need to efficiently (usually) determine if two sequences of vertices are equivalent under the group generated by the automorphisms found so far.

Automorphism group handling

Programs using depth-first search naturally produce automorphisms in the form of a base and strong generating set.

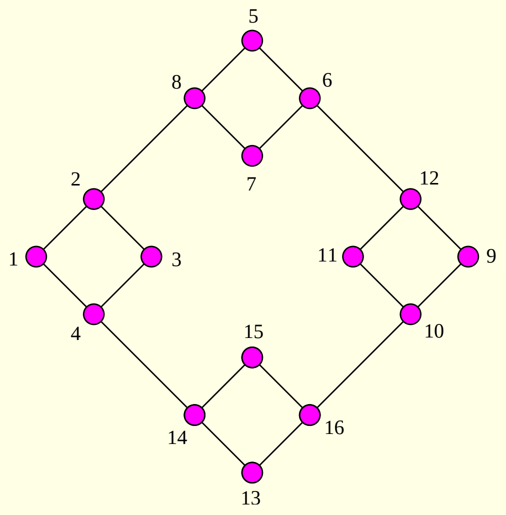
Other programs, like `Traces`, produce less predictable generators.

In order to perform automorphism-based pruning of the search tree, we need to efficiently (usually) determine if two sequences of vertices are equivalent under the group generated by the automorphisms found so far.

`Traces` uses a technique called the random Schreier method to perform this computation probabiliistically.

`nauty` also uses this in circumstances where the basic group handling is insufficient.

Base and strong generating set



Fixing 1,5,9,13 : everything fixed

Fixing 1,5,9. Orbit of 13 is $\{13, 15\}$.

(13 15)

Fixing 1,5. Orbit of 9 is $\{9, 11\}$.

(9 11)

Fixing 1. Orbit of 5 is $\{5, 7, 13, 15\}$.

(5 7)

(2 4)(5 13)(6 16)(7 15)(8 14)(10 12)

Fixing nothing. Orbit of 1 is $\{1, 3, 5, 7, 9, 11, 13, 15\}$.

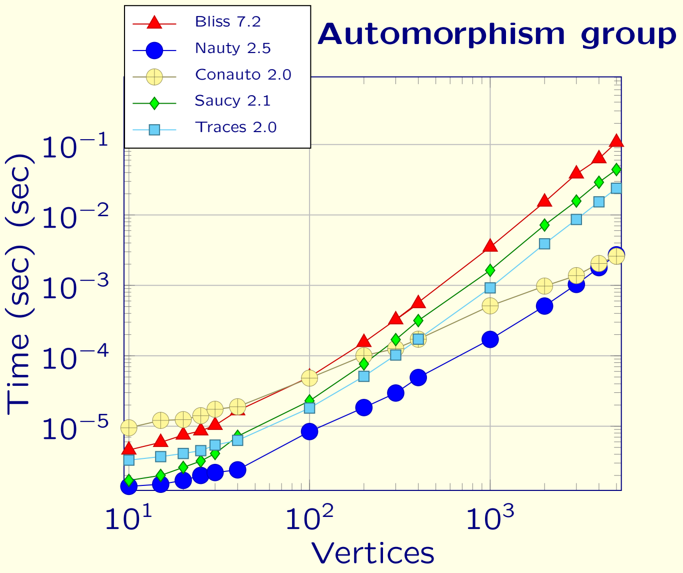
(1 3)

(1 5)(2 8)(3 7)(4 6)(9 13)(10 16)(11 15)(12 14)

Group size $= 2 \times 2 \times 4 \times 8 = 128$.

This is a base and strong generating set.

There are at most $n - 1$ generators, even though the group size can be up to $n!$.



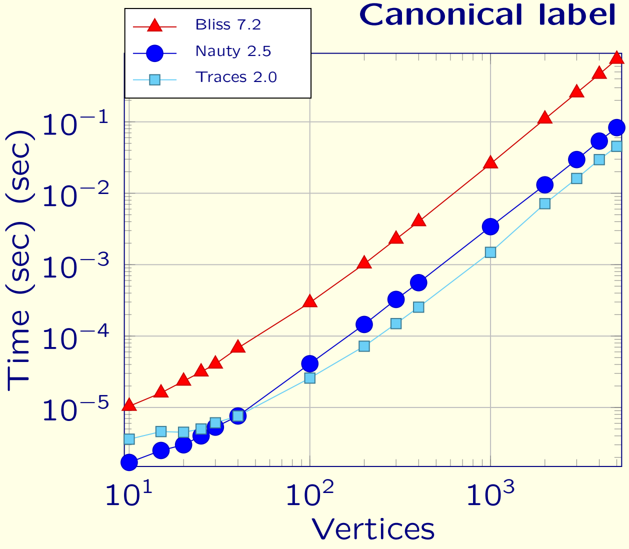
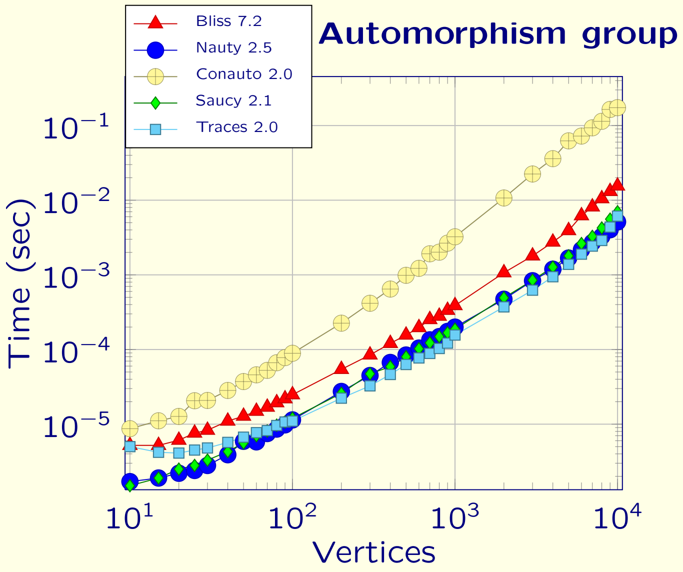


Figure 1: Random graphs with $p = \frac{1}{2}$



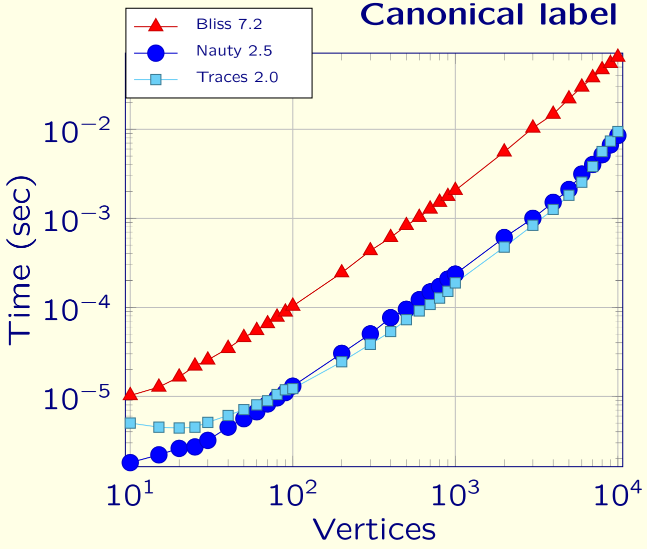
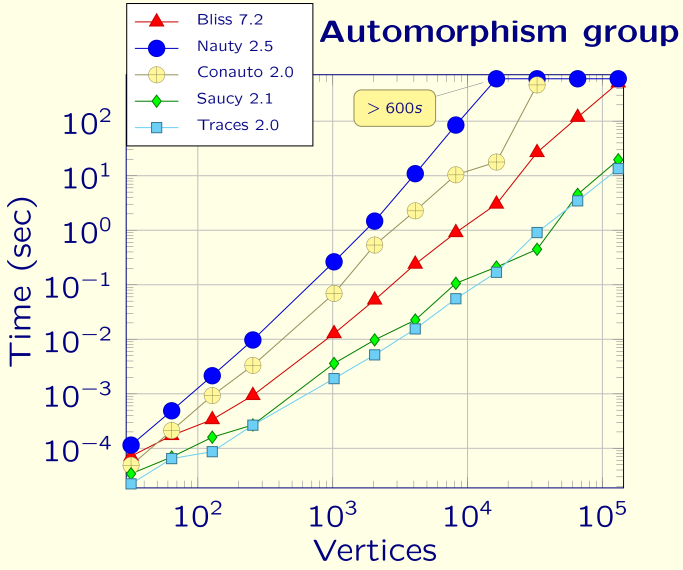


Figure 2: Random graphs with average degree $\sqrt{n}$



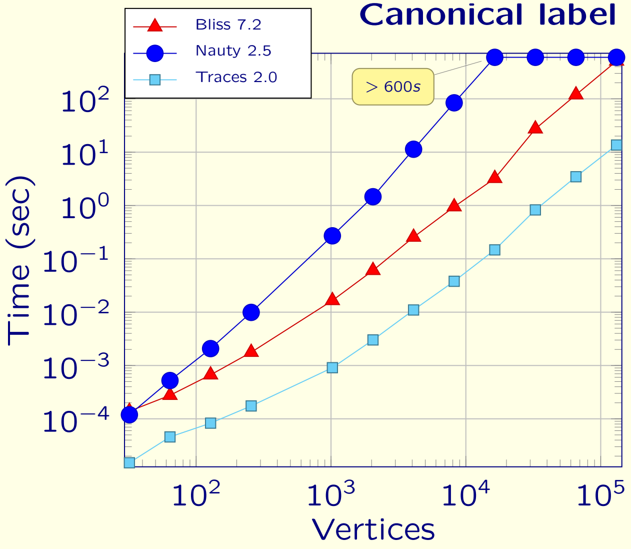
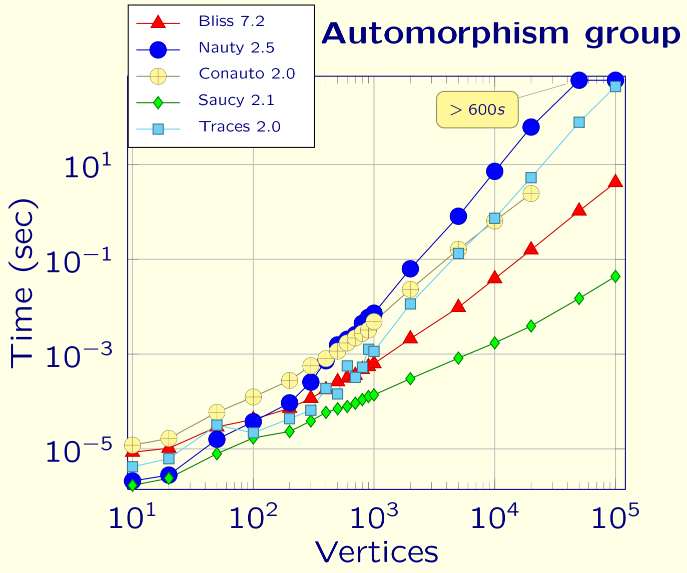


Figure 3: Random cubic graphs



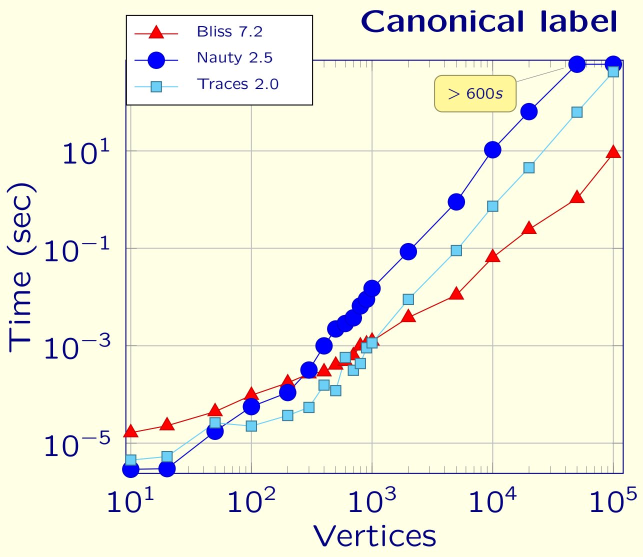
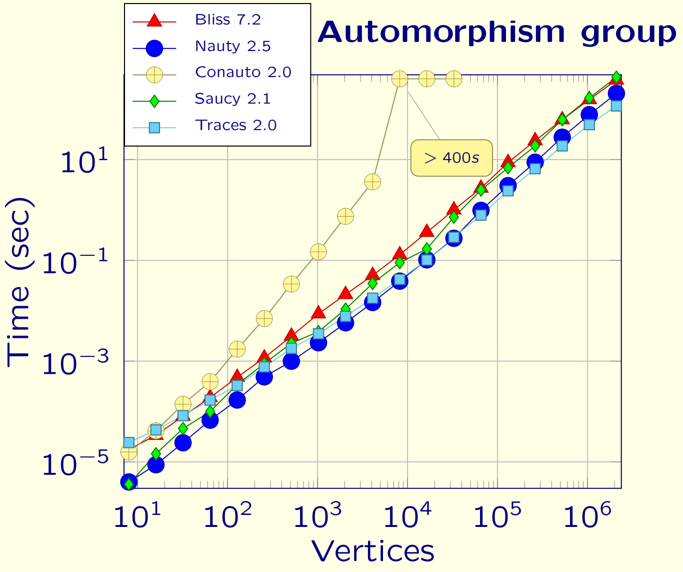


Figure 4: Random trees



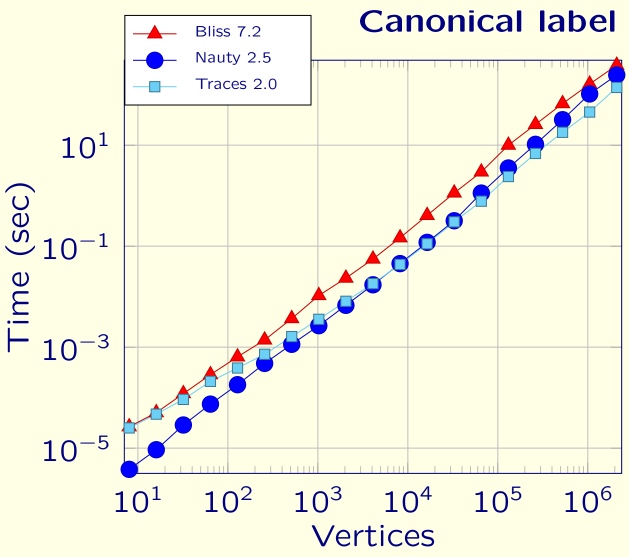
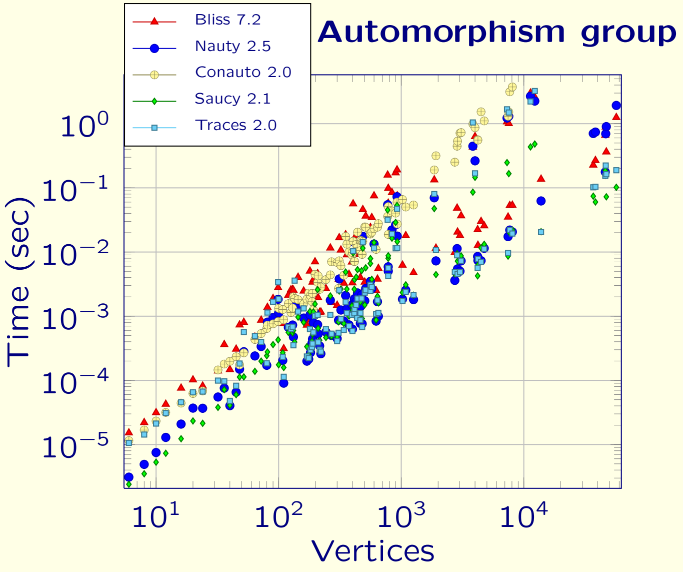


Figure 5: Hypercubes



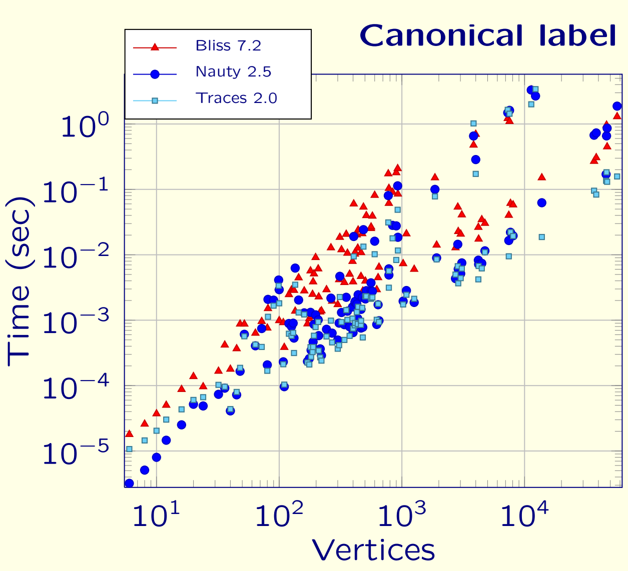
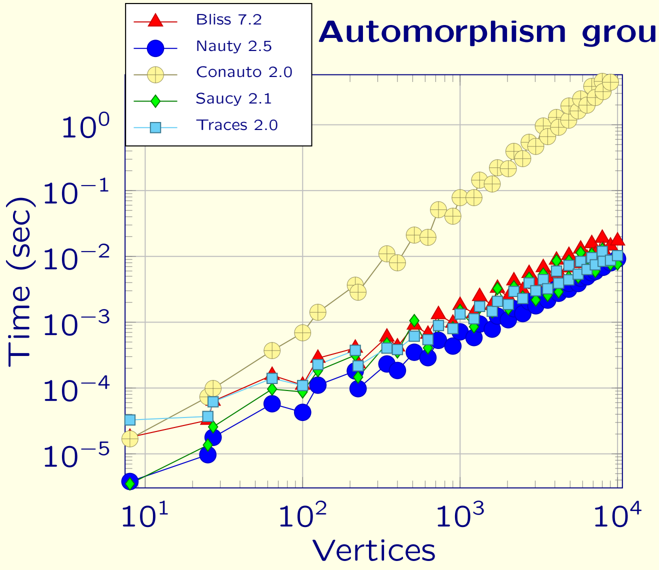


Figure 6: Miscellaneous vertex-transitive graphs



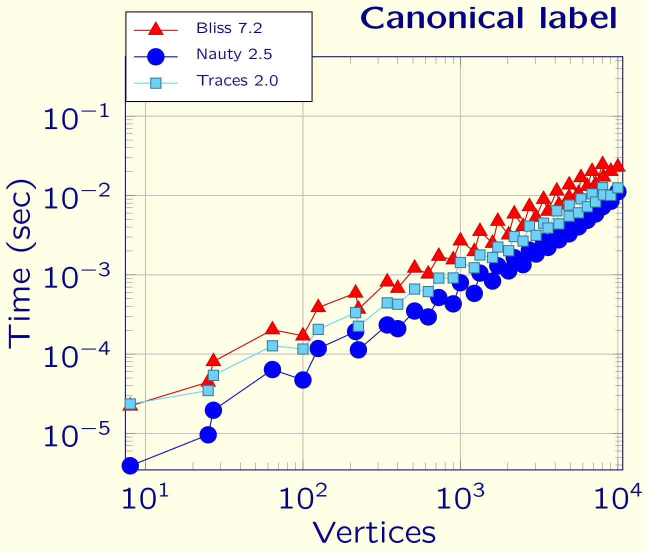
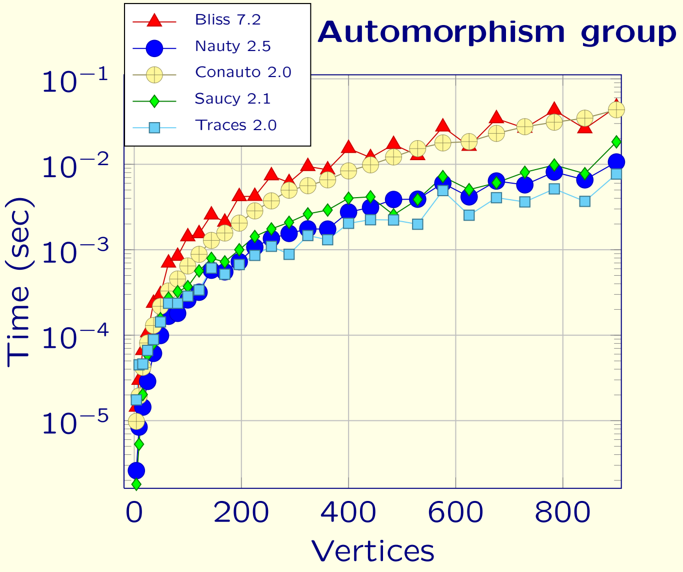


Figure 7: 2-dimensional and 3-dimensional grids



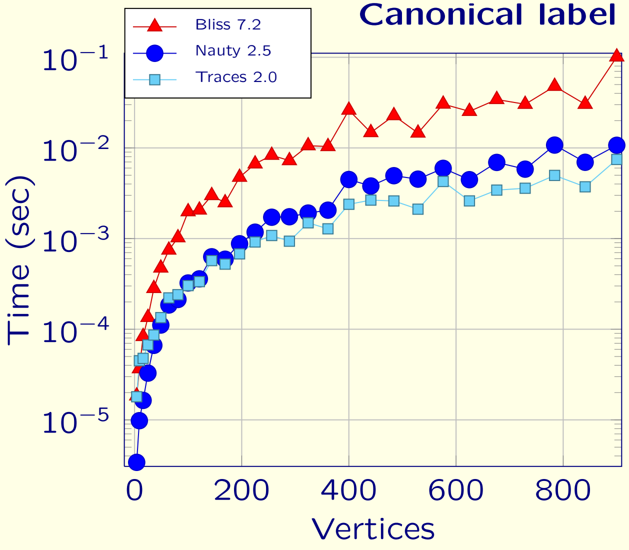
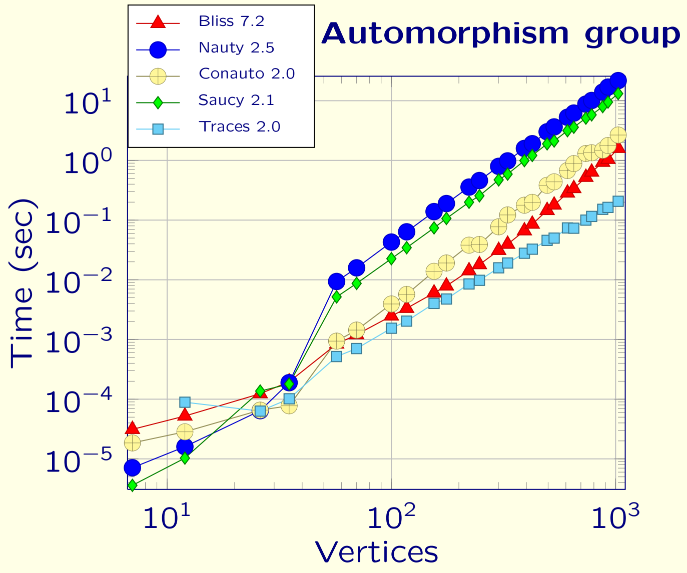


Figure 8: Latin square graphs



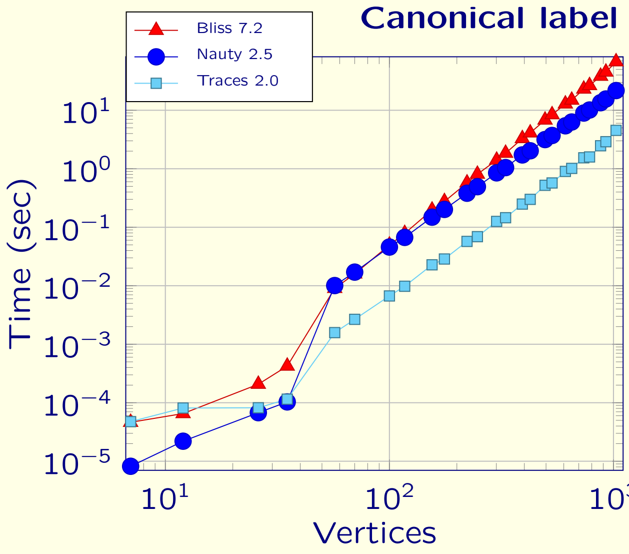
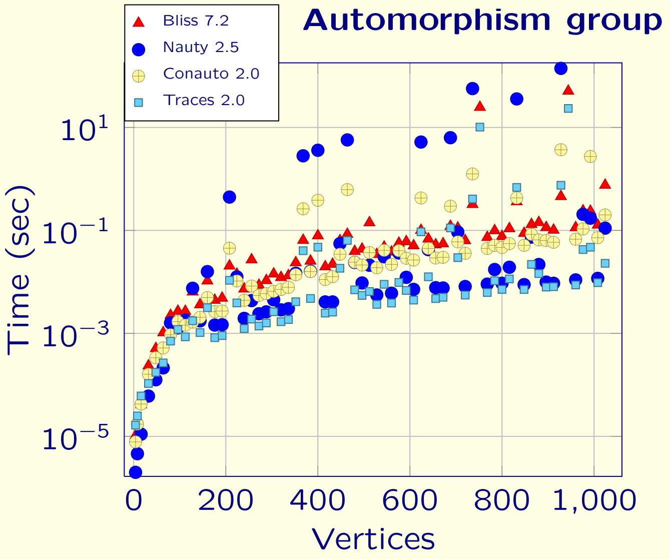


Figure 9: Strongly regular graphs



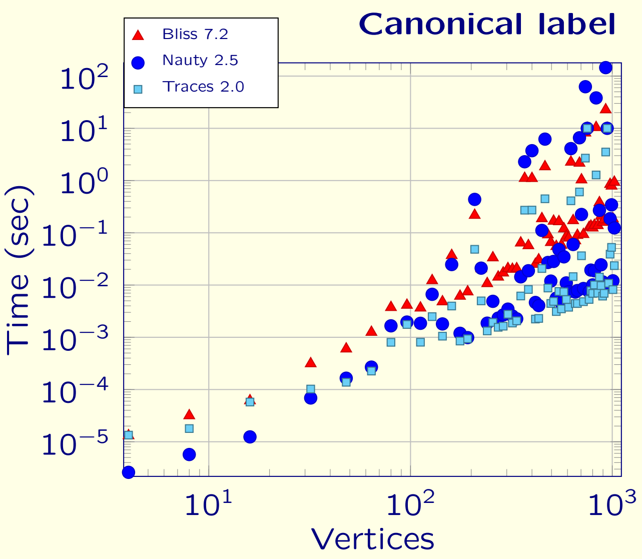
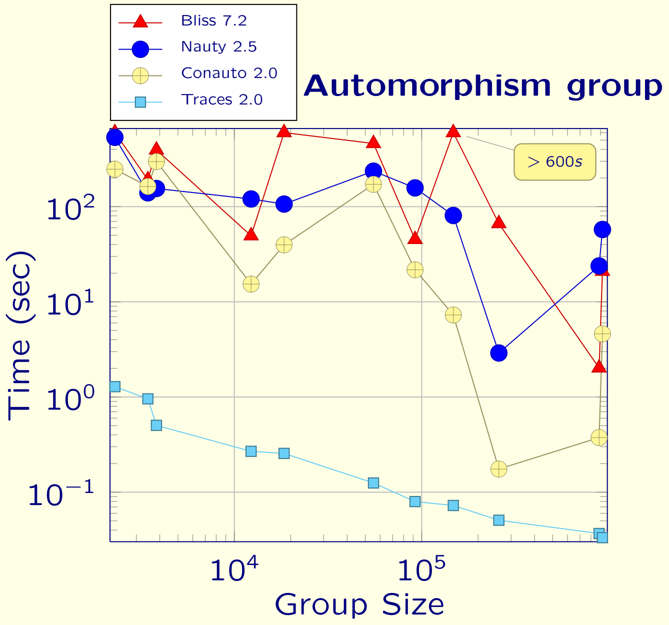


Figure 10: Hadamard matrix graphs



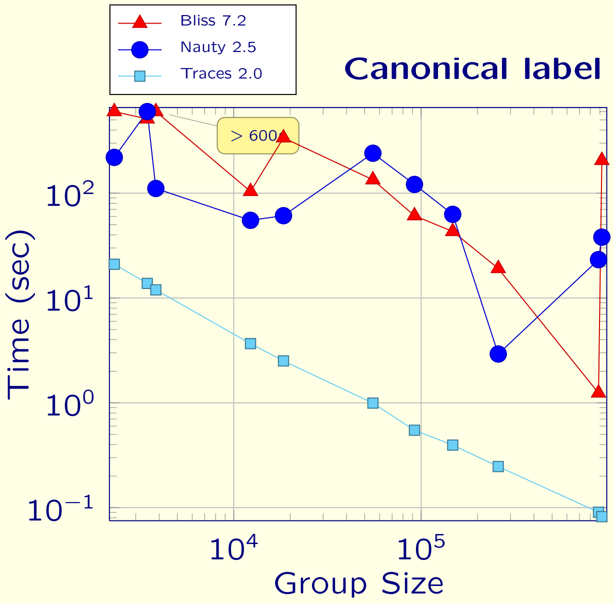
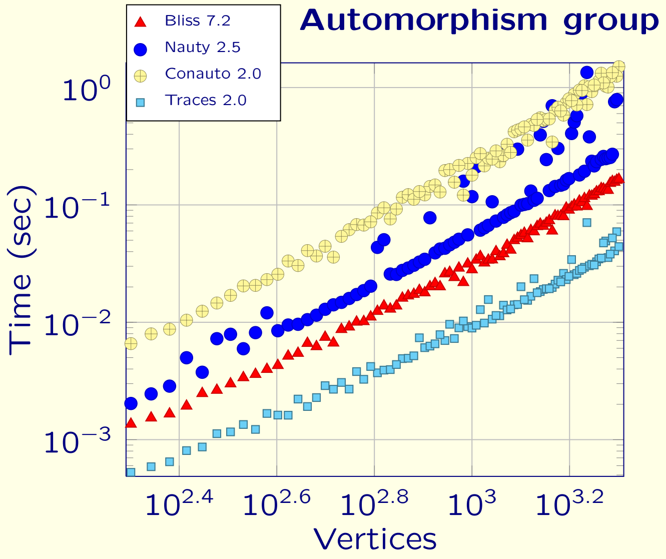


Figure 11: Projective planes of order 16



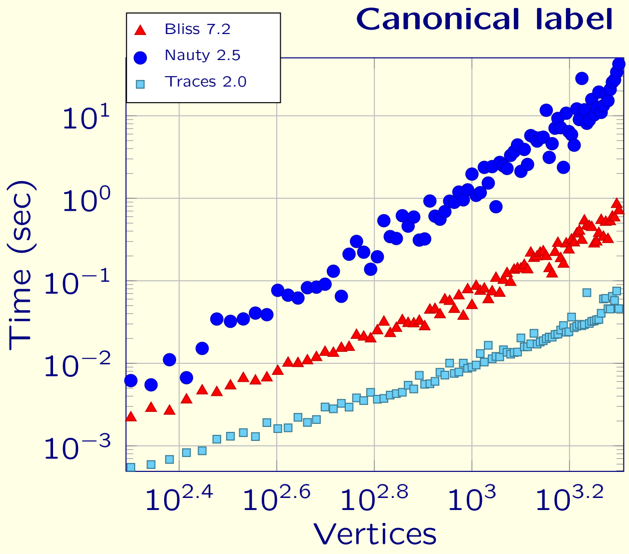
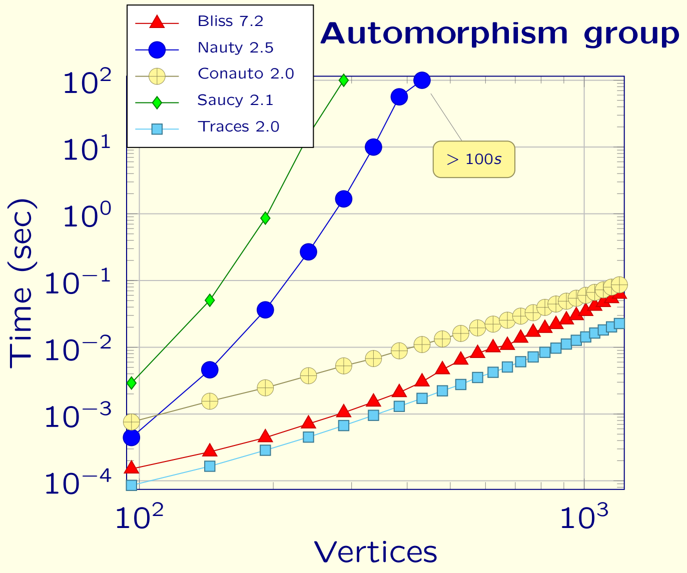


Figure 12: Cai-Fürer-Immermann graphs



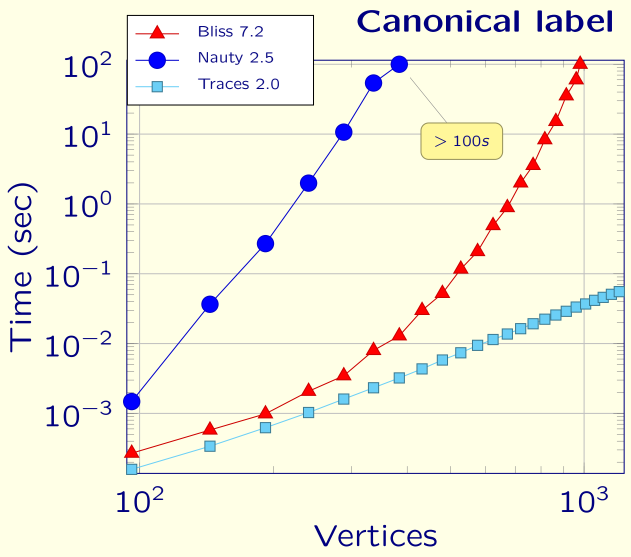


Figure 13: Miyazaki graphs