nauty User's Guide (Version 2.2)

Brendan D. McKay
Computer Science Department
Australian National University
ACT 0200, Australia
`bdm@cs.anu.edu.au`

Contents.

0. How to use this Guide.

The **dreadnaut** program provides sufficient functionality that most simple applications can be managed without the need to write any programs. Section 12 is intended to be a fairly self-contained introduction to that level of use. You should start reading there; it will direct you to any necessary information which appears elsewhere.

If you wish to write C programs which call **nauty**, you don't have much choice but to read this Guide from start to finish. However, it isn't really as hard as it sounds; see the example in Appendix A for an constructive proof.

The current version of **nauty** is available at `http://cs.anu.edu.au/~bdm/nauty`.

1. Introduction.

nauty (**n**o **aut**omorphisms, **y**es?) is a set of procedures for determining the automorphism group of a vertex-coloured graph. It provides this information in the form of a set of generators, the size of the group, and the orbits of the group. It is also able to produce a canonically-labelled isomorph of the graph, to assist in isomorphism testing. The mathematical basis for the algorithm is described in [5]; only a broad outline is given here. Note, however, that a great number of improvements have been made since the implementation described in [5].

Useful ideas received from Greg Butler, Aaron Grosky, Andrew Kirk, Bill Kocay, Rudi Mathon, Kevin Malysiak, Mark Henderson, Gordon Royle, Carsten Saager, Neil Sloane, Don Taylor, Gunnar Brinkmann, Yann Kieffer, Wendy Myrvold, Günter Stertenbrink, and several others, are gratefully acknowledged.

The author would appreciate receiving any comments about the program and/or this Guide, especially about apparent bugs.

nauty is written in a highly portable subset of the language C. Modern C compilers for most types of computer should be able to handle **nauty** without difficulty.

2. The Algorithm.

Throughout this document, a *graph* is a simple graph with n vertices labelled $0, 1, \ldots, n{-}1$. Digraphs, and graphs with loops, can also be handled correctly (see Section 4), but we will not mention them much. The vertex set of a graph G is denoted by $V = V(G)$.

The terms *colouring* and *partition* will be used interchangeably to denote a partition of V into disjoint non-empty *colour classes* or *cells*. The order of the cells is significant, but the order of the vertices within each cell is not. If π_1 and π_2 are partitions, then π_1 is *finer* than π_2, and π_2 is *coarser* than π_1, if every cell of π_1 is a subset of some cell of π_2. (Note that partitions are both finer and coarser than themselves.) A *singleton cell* is a cell with cardinality one, while a *discrete* partition is one with only singleton cells.

Let G be a graph, γ a permutation of V, $v \in V$, $W \subseteq V$, and $\pi = (V_0, V_1, \ldots, V_k)$ a partition of V. Then v^γ is the image of v under γ, $W^\gamma = \{w^\gamma \mid w \in W\}$, G^γ is the graph in which vertices x^γ and y^γ are adjacent if and only if x and y are adjacent in G, and π^γ is the partition $(V_0^\gamma, V_1^\gamma, \ldots, V_k^\gamma)$.

The *automorphism group* of a coloured graph (G, π) is the set of all permutations γ such that $G^\gamma = G$ and $\pi^\gamma = \pi$. Since the order of cells in partitions is significant, the last condition means that γ fixes each cell of π setwise (i.e., γ is *colour preserving*). In the majority of applications, π has only one cell V, so we get the usual automorphism group.

If $\pi = (V_0, V_1, \ldots, V_k)$ is a partition of $\{0, 1, \ldots, n{-}1\}$, then $c(\pi)$ is the partition $(\{0, 1, \ldots, |V_0|{-}1\}, \{|V_0|, \ldots, |V_0| + |V_1| - 1\}, \ldots, \{n - |V_k|, \ldots, n{-}1\})$. Thus, $c(\pi)$ has the same cell sizes as π, in the same order, but is otherwise independent of π.

A *canonical labelling map* is a function $\mathcal{C}$ such that, for any graph G, partition π of V, and permutation γ of V, we have
(a) $\mathcal{C}(G, \pi) = G^\delta$ for some permutation δ such that $\pi^\delta = c(\pi)$, and
(b) $\mathcal{C}(G^\gamma, \pi^\gamma) = \mathcal{C}(G, \pi)$.
Informally, $\mathcal{C}$ relabels the vertices of G in order of colour, ignoring the original vertex labels. The usefulness of a canonical labelling map is as follows.

Theorem. *Suppose the graphs G_1 and G_2 are coloured using the same number of vertices of each colour. Then $\mathcal{C}(G_1, \pi_1) = \mathcal{C}(G_2, \pi_2)$ iff $G_1^\gamma = G_2$ for some colour-preserving permutation γ. (Here, π_1 and π_2 are the colourings, with the colours in the same order in each.)*

Let G be a graph and π a partition of V with cells $V_0, V_1, \ldots, V_k$. Then π is *equitable* (with respect to G) if there are numbers d_{ij} such that each vertex in V_i is adjacent to precisely d_{ij} vertices in V_j, for $0 \le i, j \le k$. Up to the order of the cells, there is a unique coarsest equitable partition which is finer than any given partition.

A *refinement function* is a function $\mathcal{R}$ such that, for any graph G, partition π of V, and permutation γ of V, we have
(a) $\mathcal{R}(G,\pi)$ is a partition of V which, up to the order of the cells, is the coarsest equitable partition finer than π, and
(b) $\mathcal{R}(G^\gamma,\pi^\gamma) = \mathcal{R}(G,\pi)^\gamma$.

The algorithm used by **nauty** is a backtrack program which can be described in terms of the usual associated search tree. We will refer to the *nodes* of the tree to avoid confusion with the *vertices* of G. The root of the tree is associated with the initial colouring π of G and the equitable partition $\pi' = \mathcal{R}(G,\pi)$. If π' is discrete, the automorphism group is trivial and we can obtain $\mathcal{C}(G,\pi)$ by labelling the vertices of G in the order that they appear in π'. Suppose more generally that the equitable partition π' is associated with some node ν of the tree. If π' is discrete, then ν has no children. If π' is not discrete, let C be a non-singleton cell of it. This is called the *target cell* for this node (chosen by **nauty** according to some rule). For each vertex $v \in C$ we have a child of ν associated with the partition got from π' by replacing the cell C by the pair of cells $\{v\}$ and $C-\{v\}$, in that order, and the equitable partition obtained by applying $\mathcal{R}$ to it. The children of ν are generated in ascending order of the labels on the vertices of C.

Any node of the tree for which the equitable partition is discrete corresponds to a labelling of G, as described above. Automorphisms of the graph are found by noticing that two such labellings give the same labelled graph. The canonical labelling map corresponds to one of these labellings, chosen according to a complicated scheme for which you will have to consult [5] or the source code.

Except in particularly simple cases, only some of the tree is actually generated. The other parts of the tree are either shown to be equivalent to parts already generated, or shown to be uninteresting. Again, see [5] for details.

In Figure One, we show an example of the part of the tree which is actually generated. The nodes are represented by their equitable partitions, assuming that the original colouring only used one colour. The target cells are underlined and the numbers on the tree edges give the elements of the target cells which are being fixed. In this example, all the leaves are equivalent and correspond to the automorphisms (1), (1 2)(4 5), and (0 1)(3 4), respectively.

3. Data Structures and Size Limits.

A `setword` is an unsigned integer type of either 16, 32 or 64 bits, depending on the compile-time parameter WORDSIZE. (By default, WORDSIZE is the largest of 32 and the size of type `int`.)

A `set` (by which we always mean a subset of $V = \{0,1,\ldots,n{-}1\}$) is represented by an array of m `setword`s, where m is some number such that WORDSIZE $\times\, m \geq n$. The bits of a `set` are numbered $0,1,\ldots,n{-}1$ left to right (within each `setword`: high order to low order). Bits which don't get numbers are called "unnumbered" and are assumed permanently zero. A `set` represents the subset $\{\, i \mid \text{bit } i \text{ is } 1 \,\}$.

A `graph` is represented by an array of n `set`s (so it has mn `setword`s altogether). The i-th `set` gives the vertices to which vertex i is adjacent, for $0 \leq i < n$.

The C types `setword`, `set` and `graph` are actually the same, so a graph is really represented by a 1-dimensional array of length mn, not by an array of arrays.

A permutation of V is represented by an array of n integers, the i-th entry giving the image of i under the permutation. The type of the entries, `permutation` is either the same as `int` or `short int`, depending on the circumstances (see below).

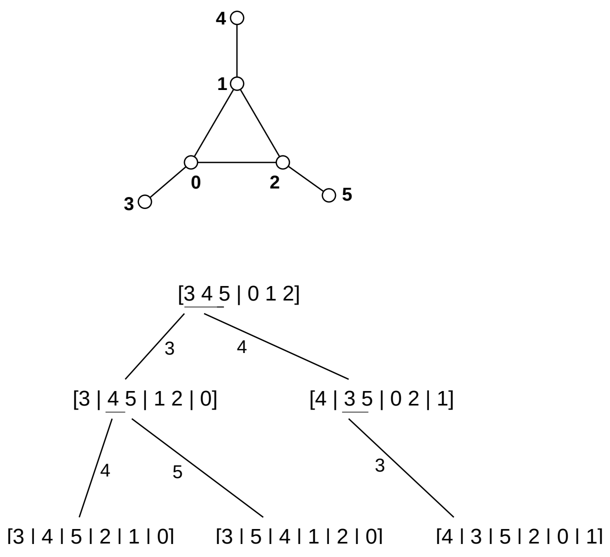


Figure One

The type `boolean` is a synonym for `int`, but the different name is intended to encourage you to restrict the values to either TRUE or FALSE (which are defined as 1 and 0, respectively).

The structured types `optionblk` and `statsblk` are described below. All these types are defined in the file `nauty.h`.

Note that types like `set` actually refer to the elements of the arrays (in this case `setword`) rather than the arrays themselves. This is done because the lengths of the arrays are not known in advance. We use `set` rather than `setword` purely for self-documentation purposes.

There are several ways to compile **nauty**, leading to differences in types and the size of graph that can be processed. These are selected by preprocessor variables.

(1) If BIGNAUTY is not defined, `permutation` is defined to be `short int` and there is an absolute limit of $2^{15} - 3 = 32765$ on the order of a graph. This is the default.

(2) If BIGNAUTY is defined, `permutation` is defined to be `int` and there is an absolute limit of $2^{24} - 3 = 16777213$ on the order of a graph.

In addition, there is a choice between static and dynamic memory allocation for the larger data objects. This is selected by the value of the preprocessor variable MAXN.

(a) If MAXN is defined as 0, the limit on the order of a graph is given in (1)–(2) above and objects are dynamically allocated. Of course, if you don't have enough memory, dynamic allocation may fail.

(b) If MAXN is defined as a positive integer, that is the limit on the order of a graph. It can't be greater than the absolute limit given in (1)–(2) above. In this case objects are statically allocated, so space is wasted if MAXN is much larger than what is actually used.

A special case of option (b) is $0 <$ MAXN $\leq$ WORDSIZE, which implies that a `set` consists of a single `setword`. Some of the critical routines in **nauty** have special code to optimize performance in that case. The recommended way to compile for this case is to define MAXN to be the name WORDSIZE.

4. Parameters.

A call to **nauty** has the form

nauty (*g*, *lab*, *ptn*, *active*, *orbits*, *options*, *stats*, *workspace*, *worksize*, *m*, *n*, *canong*)

where the parameters have meanings as defined below.

`graph` $*g$: The input graph. Read-only.

`int` $*$*lab*,$*$*ptn*: Two arrays of n entries. Their use depends on the values of several options. If *options.defaultptn* = TRUE, the input values are ignored; otherwise, they define the initial colouring of the graph (see below). If *options.getcanon* = TRUE, the value of *lab* on return is the canonical labelling of the graph. Precisely, it lists the vertices of g in the order in which they need to be relabelled to give *canong*. Irrespective of *options.getcanon*, neither *lab* nor *ptn* is changed by enough to change the colouring. (Recall that the order of the vertices within the cells is irrelevant.) Read-Write.

`set` $*$*active*: An array of m `setword`s specifying the colours which are initially active. A brief outline of what this means is given below. This argument is rarely used; **nauty** will always work correctly if given the nil pointer NULL. Read-only.

`int` $*$*orbits*: An array of n entries to hold the orbits of the automorphism group. When **nauty** returns, *orbits*$[i]$ is the number of the least-numbered vertex in the same orbit as i, for $0 \leq i \leq n{-}1$. Write-only.

`optionblk` $*$*options*: A structure giving a list of options to the procedure. See below for their meanings. Read-only.

`statsblk` $*$*stats*: A structure used by **nauty** to provide some statistics about what it did. See below for their meanings. Write-only.

`setword` $*$*workspace, worksize*: The address and length of an integer array used by **nauty** for working storage. There is no minimum requirement for correct operation, but the efficiency may suffer if not much is provided. A value of *worksize* $\geq 50m$ is recommended. Write-only and read-only, respectively.

`int` *m, n*: The number of `setword`s in `set`s and the number of vertices, respectively. It must be the case that $1 \leq n \leq m \times$ WORDSIZE. If **nauty** is compiled with MAXN $>$ 0, it must also be the case that $n \leq$ MAXN and $m \leq$ MAXM, where MAXM $=$ $\lceil$MAXN/WORDSIZE$\rceil$. Read-only.

`graph` $*$*canong*: The canonically labelled isomorph of g produced by **nauty**. This argument is ignored if *options.getcanon* = FALSE, in which case the nil pointer NULL can be given as the actual parameter. Write-only.

The initial colouring of the graph is determined by the values of the arrays *lab*, *ptn* and the flag *options.defaultptn*. If *options.defaultptn* = TRUE, the contents of *lab* and *ptn* are set by **nauty** so that every vertex has the same colour. If not, they are assumed to have been set by the user. In this case, *lab* should contain a list of all the vertices in some order such that vertices with the same colour are contiguous. The ends of the colour-classes are indicated by zeros in *ptn*. In super-precise terms, each cell has the form $\{lab[i], lab[i{+}1], \ldots, lab[j]\}$ where $[i, j]$ is a maximal subinterval of $[0, n{-}1]$ such that $ptn[k] > 0$ for $i \leq k < j$ and $ptn[j] = 0$. (In the terminology defined in Section 7, this is the "partition at level 0".) The order of the vertices within each cell has no effect on the behaviour of **nauty**. An example is given in Section 6.

The concept of *active cells* is used by the procedure which implements the partition refinement function $\mathcal{R}$ defined above. The details are given in [5], where the active cells are

in a sequence called α. In this implementation, a set rather than a sequence is used. If *options.defaultptn* = TRUE, or *active* = NULL, every colour is active. This will always work, and so is recommended if you don't want to be a smart-arse. If *options.defaultptn* = FALSE and *active* $\neq$ NULL, the elements of *active* indicate the indices $(0\,..\,n{-}1)$ where the active cells start in *lab* and *ptn* (see above). Theorem 2.7 of [5] gives some sufficient conditions for *active* to be valid. If these conditions are not met, anything might happen. The most common places where this feature may save a little time are:
(a) If the initial colouring is known to be already equitable, *active* can be the empty set. (Don't confuse this with NULL, which causes **nauty** to set the active set to include every cell.)
(b) If the graph is regular and the colouring has exactly two cells, *active* can indicate just one of them (the smallest for best efficiency).
If **nauty** is used to test two graphs for isomorphism, it is essential that exactly the same value of *active* be used for each of them.

The various fields of the structure *options* fine-tune the behaviour of **nauty**. The recommended way to assign values to these options is to start with the declaration
`DEFAULTOPTIONS(options);`
This defines the static variable `options` of type `optionblk`, initialized to sensible values for most circumstances. Changes in those values can then be made using assignment statements, for example
`options.linelength = 100;`
This practice will protect your code from breaking if additional fields are added to *options* in future editions of **nauty**, which is quite likely. (You should just need to recompile.)

All of these fields are read-only.

`boolean` *getcanon*: If this is TRUE, the canonically labelled isomorph *canong* is produced, and *lab* is set to indicate the canonical label, as described above. Otherwise, only the automorphism group is determined. Sometimes, different generators of the automorphism group are found if this option is selected; of course, the group they generate is the same. Default FALSE.

`boolean` *digraph*: This <u>must</u> be TRUE if the graph has any directed edges or loops. It has the effect of turning off some heuristics which are only valid for simple graphs. If no directed edges or loops are present, selecting this option is legal but may degrade the performance slightly. Default FALSE.

`boolean` *writeautoms*: If this is TRUE, generators of the automorphism group will be written to the file *outfile* (see below). The format will depend on the settings of options *cartesian* and *linelength* (see below, again). More details on what is written can be found in Section 5. Default FALSE (changed with version 2.1).

`boolean` *writemarkers*: If this is TRUE, extra data about the automorphism group generators will be written to the file *outfile* (see below). An explanation of what these data are can be found in Section 5. Default FALSE (changed with version 2.1).

`boolean` *defaultptn*: This has been fully explained above. Default TRUE.

`boolean` *cartesian*: If *writeautoms* = TRUE, the value of this option effects the format in which automorphisms are written. If *cartesian* = FALSE, the output is the usual cyclic representation of γ, for example "(2 5 6)(3 4)". If *cartesian* = TRUE, the output for an automorphism γ is the sequence of numbers "1^γ 2^γ ... $(n{-}1)^\gamma$", for example "1 5 4 3 6 2". Default FALSE.

`int` *linelength*: The value of this variable specifies the maximum number of characters per line (excluding end-of-line characters) which may be written to the file *outfile* (see below). Actually, it is ignored for the output selected by the option *writemarkers*, but that never has more than about 65 characters per line anyway. A value of 0 indicates no limit. Default CONSOLWIDTH, which can be defined when compiling but is set to 78 otherwise.

`FILE` **outfile*: This is the file to which the output selected by the options *writeautoms* and *writemarkers* is sent. It must be already open and writable. The nil pointer NULL is equivalent to `stdout` (the standard output). Default NULL.

`void` (**userrefproc*)(): This is a pointer to a user-defined procedure which is to be called in place of the default refinement procedure. Section 7 has details. If the value is NULL, the default refinement procedure is used. Default NULL.

`void` (**userautomproc*)(): This is a pointer to a user-defined procedure which is to be called for each generator. Section 7 has details. No calls will be made if the value is NULL. Default NULL.

`void` (**userlevelproc*)(): This is a pointer to a user-defined procedure which is to be called for each node in the leftmost path downwards from the root, in bottom to top order. Section 7 has details. No calls will be made if the value is NULL. Default NULL.

`void` (**usernodeproc*)(): This is a pointer to a user-defined procedure which is to be called for each node of the tree. Section 7 has details. No calls will be made if the value is NULL. Default NULL.

`void` (**usertcellproc*)(): This is a pointer to a user-defined procedure which is to be called in place of the default routine which chooses a target cell. Section 7 has details. If the value is NULL, the default routine is used. Default NULL.

`void` (**invarproc*)(): This is a pointer to a vertex-invariant procedure. See Section 8 for a discussion of vertex-invariants. No calls will be made if the value is NULL. Default NULL.

`int` *tc_level*: Two rules are available to choose target cells. On levels up to level *tc_level*, inclusive, an expensive but (empirically) highly effective rule is used. (The root of the search tree is at level one.) At deeper levels, a cheaper rule is used. For difficult graphs, a large value is recommended. For easier graphs, use 0. Default 100.

`int` *mininvarlevel*: The absolute value gives the minimum level at which *invarproc* will be applied. (The root of the search tree is at level one.) If *options.getcanon* = FALSE, a negative value indicates that the minimum level will be automatically set by **nauty** to the least level in the left-most path in the search tree where *invarproc* is applied and refines the partition. If *options.getcanon* = TRUE, the sign is ignored. A value of 0 indicates no minimum level. Default 0.

`int` *maxinvarlevel*: The absolute value gives the maximum level at which *invarproc* will be applied. (The root of the search tree is at level one.) If *options.getcanon* = FALSE, a negative value indicates that the maximum level will be automatically set by **nauty** to the least level in the left-most path in the search tree where *invarproc* is applied and refines the partition. If *options.getcanon* = TRUE, the sign is ignored. A value of 0 effectively disables *invarproc*. Default 1 (changed with version 2.1).

`int` *invararg*: This level is passed by **nauty** to the vertex-invariant procedure *invarproc*, which might use it for any purpose it pleases. Default 0.

`dispatchvec` **dispatch*: This is a vector of procedure pointers used to apply **nauty** to objects other than graphs. Version 2.2 only has full support for graphs. The value NULL is equivalent to the structure *dispatch_graph* defined in `naugraph.c`, which is what this option needs for graphs. Default NULL.

Some of the fields in the *options* argument may change the canonical labelling produced by **nauty**. These are fields *digraph*, *defaultptn*, *tc_level*, *userrefproc*, *usertcellproc*, *invarproc*, *mininvarlevel*, *maxinvarlevel*, *invararg* and *dispatch*. If **nauty** is used to test two graphs for isomorphism, it is important that the same values of these options be used for both graphs.

The various fields of the structure *stats* are set by **nauty**. Their meanings are as follows:

`double` *grpsize1*, `int` *grpsize2*: The order of the automorphism group is equal to $grpsize1 \times 10^{grpsize2}$, within rounding error. If the exact size of a very large group is needed, it can be calculated from the output selected by the *writemarkers* option. See Section 5.

`int` *numorbits*: The number of orbits of the automorphism group.

`int` *numgenerators*: The number of generators found.

`int` *errstatus*: If this is nonzero, an error was detected by **nauty**. Possible values are:
• MTOOBIG: m is too big; i.e., the maximum is 16777215/WORDSIZE+1 if MAXN=0 and BIGNAUTY is defined, 32765/WORDSIZE+1 if MAXN=0 and BIGNAUTY is not defined, and $\lceil$MAXN/WORDSIZE$\rceil$ otherwise.
• NTOOBIG: n is too big. $n >$ MAXN and MAXN > 0, or $n > 32765$ (16777213 if BIGNAUTY is defined), or $n >$ WORDSIZE $\times\, m$
• CANONGNIL: *canong* = NULL, but *options.getcanon* = TRUE.
nauty also writes a message to `stderr` in these cases, so there is no real need to test this parameter in most applications.

`long` *numnodes*: The total number of tree nodes generated.

`long` *numbadleaves*: The number of leaves of the tree which were generated but were useless in the sense that no automorphism was thereby discovered and the current-best-guess at the canonical labelling was not updated.

`int` *maxlevel*: The maximum level of any generated tree node. The root of the tree is on level one.

`long` *tctotal*: The total size of all the target cells in the search tree. The difference between this value and *numnodes* provides an estimate of the efficiency of **nauty**'s search-tree pruning.

`long` *canupdates*: The number of times the program's idea of the "best candidate for canonical label" was updated, including the original one.

`long` *invapplics*: The number of nodes at which the vertex-invariant was applied.

`long` *invsuccesses*: The number of nodes at which the vertex-invariant succeeded in refining the partition more than the refinement procedure did.

`int` *invarsuclevel*: The least level of the nodes in the tree at which the vertex-invariant succeeded in refining the partition more than the refinement procedure did. The value is zero if the vertex-invariant was never successful.

The values corresponding to node counts might overflow in a long computation, but this is not a serious problem as they are not used during the computation.

In addition to their parameters, the output routines of **nauty** respect the value of the global `int` variable *labelorg*. If the value of *labelorg* is k, the output routines pretend that the vertices of the graph are numbered $k, k+1, \ldots, n+k-1$, even though they are internally numbered $0, 1, \ldots, n-1$. By default, $k = 0$. Only non-negative values are supported.

5. Output.

If *options.writeautoms* = TRUE or *options.writemarkers* = TRUE, information concerning the automorphism group is written to the file *options.outfile*.

Let Γ be the automorphism group, and let $\Gamma_{v_1,v_2,\ldots,v_i}$ denote the point-wise stabiliser in Γ of $v_1, v_2, \ldots, v_i$. The output has the following general form:

```
    γ_1^(k)
    γ_2^(k)
    .
    .
    .
    γ_{t_k}^(k)
level k:   c_k cells; r_k orbits; v_k fixed; index i_k/j_k
    γ_1^(k−1)
    γ_2^(k−1)
    .
    .
    .
    γ_{t_{k−1}}^(k−1)
level k−1:   c_{k−1} cells; r_{k−1} orbits; v_{k−1} fixed; index i_{k−1}/j_{k−1}

    .
    .
    .

level 2:   c_2 cells; r_2 orbits; v_2 fixed; index i_2/j_2
    γ_1^(1)
    γ_2^(1)
    .
    .
    .
    γ_{t_1}^(1)
level 1:   c_1 cells; r_1 orbits; v_1 fixed; index i_1/j_1
```

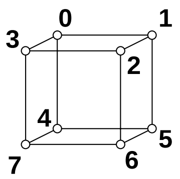
Here, $v_1, v_2, \ldots, v_k$ is a sequence of vertices such that $\Gamma_{v_1,v_2,\ldots,v_k}$ is trivial. The $\gamma_i^{(j)}$ are automorphisms. For $1 \le l \le k$, the following are true.
(a) $\Gamma_{v_1,v_2,\ldots,v_{l-1}}$ is generated by the automorphisms $\gamma_i^{(j)}$ for $l \le j \le k$ and $1 \le i \le t_j$.
(b) $\Gamma_{v_1,v_2,\ldots,v_{l-1}}$ has r_l orbits and order $i_1 i_2 \cdots i_l$.
(c) c_l is the number of cells in the equitable partition at the ancestor at level l of the first leaf of the tree, j_l is the number of vertices in the target cell of the same node, v_l is the first vertex in that cell, and i_l is the number of vertices of that cell which are equivalent to v_l.
(d) $\sum_{i=l}^{k} t_i \le n - r_l$. This follows from the fact that the number of orbits of the group generated by all the automorphisms found to up to any moment decreases as each new automorphism is found. In particular, this means that the total number of generators found is at most $n-1$. Usually, it is much less.

The markers “`level...`” are only written if *options.writemarkers* = TRUE. In the common circumstance that $c_l = r_l$, “c_l `cells;`” is omitted. Similarly, “`/`j_l” is omitted if $j_l = i_l$. Note that $i_l = 1$ is possible for more difficult graphs. Further information about the generators can be found in Theorem 2.34 of [5].

6. Examples.

All of the following examples were run without the use of a vertex-invariant.

Example 1:

$g =$ 

options[*getcanon* = FALSE, *digraph* = FALSE, *writeautoms* = TRUE, *writemarkers* = TRUE, *defaultptn* = TRUE, *cartesian* = FALSE, *tc_level* = 0].

output:

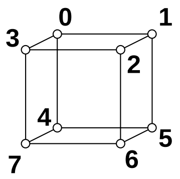
```
(2 5)(3 4)
level 3:   6 orbits; 3 fixed; index 2
(1 3)(5 7)
level 2:   4 orbits; 1 fixed; index 3
(0 1)(2 3)(4 5)(6 7)
level 1:   1 orbit; 0 fixed; index 8
```

orbits = (0,0,0,0,0,0,0,0), *stats*[*grpsize1* = 48.0, *grpsize2* = 0, *numorbits* = 1, *numgenerators* = 3, *numnodes* = 10, *numbadleaves* = 0, *maxlevel* = 4].

Explanation of output: Let γ_1, γ_2 and γ_3 be the three automorphisms found, in the order written. Let $\mathit{\Gamma}$ be the automorphism group. Then

$$\begin{aligned} \mathit{\Gamma}_{0,1,3} &= \{(1)\} \\ \mathit{\Gamma}_{0,1} &= \langle\gamma_1\rangle \ \text{ with 6 orbits and order 2} \\ \mathit{\Gamma}_{0} &= \langle\gamma_1,\gamma_2\rangle \ \text{ with 4 orbits and order } 2\times 3 = 6 \\ \mathit{\Gamma} &= \langle\gamma_1,\gamma_2,\gamma_3\rangle \ \text{ with 1 orbit and order } 6\times 8 = 48. \end{aligned}$$

Example 2:

$g =$ 

lab = (2,0,1,3,4,5,6,7), *ptn* = (0,1,1,1,1,1,1,0), *active* = NULL,
options[*getcanon* = FALSE, *digraph* = FALSE, *writeautoms* = TRUE, *writemarkers* = TRUE, *defaultptn* = FALSE, *cartesian* = TRUE, *tc_level* = 0].

output:

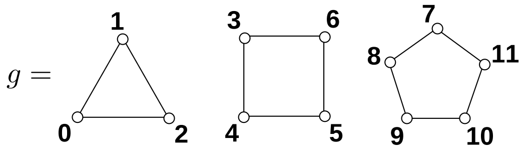
```
5 1 2 6 4 0 3 7
level 2:   6 orbits; 3 fixed; index 2
```

```
        0 3 2 1 4 7 6 5
        level 1:   4 orbits; 1 fixed; index 3
```

orbits = (0,1,2,1,4,0,1,0), *stats*[*grpsize1* = 6.0, *grpsize2* = 0, *numorbits* = 4, *numgenerators* = 2, *numnodes* = 6, *numbadleaves* = 0, *maxlevel* = 3].

In this example we have set *lab*, *ptn* and *options.defaultptn* so that vertex 2 is fixed. The automorphisms were written in the "cartesian" representation, which would probably only be useful if they were going to be fed to another program. The value of *orbits* on return indicates that the orbits of the group are $\{0, 5, 7\}$, $\{1, 3, 6\}$, $\{2\}$ and $\{4\}$.

Example 3:

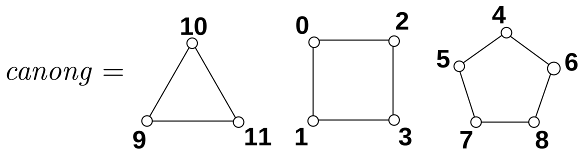


options[*getcanon* = TRUE, *digraph* = FALSE, *writeautoms* = TRUE, *writemarkers* = TRUE, *defaultptn* = TRUE, *tc_level* = 0].

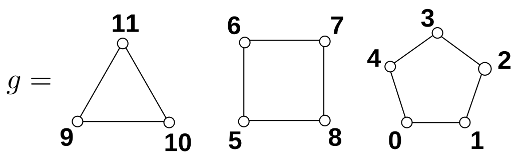
output:

```
        (8 11)(9 10)
        level 6:   10 orbits; 8 fixed; index 2
        (7 8)(9 11)
        level 5:   8 orbits; 7 fixed; index 5
        (4 6)
        level 4:   7 orbits; 4 fixed; index 2
        (3 4)(5 6)
        level 3:   4 cells; 5 orbits; 3 fixed; index 4/9
        (1 2)
        level 2:   3 cells; 4 orbits; 1 fixed; index 2
        (0 1)
        level 1:   1 cell; 3 orbits; 0 fixed; index 3/12
```

orbits = (0,0,0,3,3,3,3,7,7,7,7,7), *stats*[*grpsize1* = 480.0, *grpsize2* = 0, *numorbits* = 3, *numgenerators* = 6, *numnodes* = 40, *numbadleaves* = 2, *maxlevel* = 7], *lab* = (3,4,6,5,7,8,11,9,10,0,1,2).

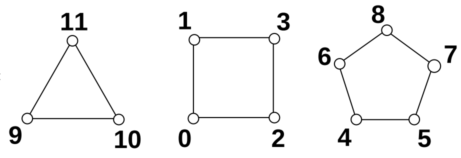


Example 4:



options[*getcanon* = TRUE, *digraph* = FALSE, *writeautoms* = FALSE, *writemarkers* = FALSE, *defaultptn* = TRUE, *tc_level* = 0].
No output written.

orbits = (0,0,0,0,0,5,5,5,5,9,9,9), *stats*[*grpsize1* = 480.0, *grpsize2* = 0, *numorbits* = 3, *numgenerators* = 6, *numnodes* = 41, *numbadleaves* = 3, *maxlevel* = 7],
lab = (5,6,8,7,0,1,4,2,3,9,10,11).

$g =$  ,

which is identical to the resulting *canong* in Example 3.

7. User-defined procedures.

Provision is made for up to five procedures specified by the user to be called at various times during the processing. This will be done if pointers to them are passed in the *userrefproc*, *userautomproc*, *usernodeproc*, *userlevelproc* and/or *usertcellproc* fields of *options* (see Section 4). In all cases, a value of NULL will result in sensible default action.

These procedures have many parameters in common; we will describe the most important of these here. Unless the individual procedure descriptions specify otherwise, they should be treated as read-only.

`graph` $*g$; `int` m, n: These are the arguments of the same name passed to **nauty**. **nauty** has not changed them. See Section 4 for their meanings.

`int` *level*: The level of the current node. The root of the search tree has level one.

`int` $*lab$, ∗ptn: Arrays of length n giving partitions associated with each of the nodes along the path from the root of the tree to the current node. These are the parameters of the same name passed to **nauty**, but **nauty** has modified their contents as described below.

Suppose that we are currently at level l of the search tree. Let $\nu_1, \nu_2, \ldots, \nu_l$ be the path in the tree from the root ν_1 to the current node ν_l. The "partition at level i" is a partition π_i associated with node ν_i. The partition originally passed to **nauty**, implicitly or explicitly, is the "partition at level 0", denoted by π_0. The complete partition nest $\pi_0, \pi_1, \ldots, \pi_l$ is held in *lab* and *ptn* thus:
(a) *lab* holds a permutation of $\{0, 1, \ldots, n-1\}$.
(b) For $0 \le t \le l$, the partition π_t has as cells all the sets of the form $\{lab[i], lab[i+1], \ldots, lab[j]\}$, where $[i, j]$ is a maximal subinterval of $[0, n-1]$ such that $ptn[k] > t$ for $i \le k < j$ and $ptn[j] \le t$.
(c) Every entry of *ptn* which is not less than or equal to l is equal to NAUTY_INFINITY. (NAUTY_INFINITY is a large constant defined in `nauty.h`.)

For example, say $n = 10$, $l = 3$, $\pi_0 = [0, 2, 4, 5, 6, 7, 8, 9|1, 3]$, $\pi_1 = [0, 2, 4, 6|5, 7, 8, 9|1, 3]$, $\pi_2 = [0, 2, 4, 6|8|5, 7, 9|3|1]$, and $\pi_3 = [4, 6|0, 2|8|5, 7, 9|3|1]$. Then the contents of *lab* and *ptn* may be

lab:	4	6	2	0	8	7	5	9	3	1
ptn:	∞	3	∞	1	2	∞	∞	0	2	0

The order of the vertices within the cells of π_l is arbitrary.

We will refer to the partition at level l as "the current partition".

(a) *userrefproc* (*g*, *lab*, *ptn*, *level*, *numcells*, *count*, *active*, *code*, *m*, *n*)

This is a procedure to replace the default partition-refinement procedure, and is called for each node of the tree. The partition associated with the node is the "partition at level *level*", which is defined above.

The parameters passed are as follows.

g,m,n,lab,ptn,level: As above. The parameters *lab* and *ptn* may be altered by this procedure to the extent of making the current partition finer. The partitions at higher levels must not be altered.

`int` ∗*numcells*: The number of cells in the current partition. This must be updated if the number of cells is increased.

`permutation` ∗*count*: This is the address of an array of length at least n which can be used as scratch space. It can be changed at will.

`set` ∗*active*: The set of active cells. This is *not* the same as the parameter of the same name passed to **nauty**, but has the same meaning and purpose. It can be changed which affecting **nauty** behaviour. See Section 4.

`int` ∗*code*: This must be set to a labelling-independent value which is an invariant of the partition at this level before or after refinement. (Example: the number of cells.) It is essential that equivalent nodes have the same code. The value assigned must be less than NAUTY_INFINITY.

The operation of refining the current partition involves permuting the vertices (i.e., entries of *lab*) within a cell, and then breaking it into subcells by changing the appropriate entries of *ptn* to *level.*

The validity of **nauty** requires that the operation performed be entirely independent of the labelling of the graph. Thus, if *userrefproc* is called with *g* and *lab* relabelled consistently and the same values of *ptn* and *active*, then the final values of *ptn* and *active* should be the same, and the final value of *lab* should be the same but relabelled in the same way (remembering always that the order of vertices within the cells doesn't matter). It is also necessary that nodes of the tree which may be equivalent must be treated equivalently. To be safe, regard any nodes on the same level as possibly equivalent.

It is desirable (but not compulsory) that the partition returned is equitable. If necessary, this can be done by calling the default refinement procedure *refine*, which has the same parameter list. If equitablility cannot be ensured, make sure that *options.digraph* = TRUE.

The usefulness of *userrefproc* has declined since vertex-invariants were introduced (see Section 8).

(b) *usernodeproc* (*g*, *lab*, *ptn*, *level*, *numcells*, *tc*, *code*, *m*, *n*)

This is called once for every node of the tree, after the partition has been refined.

The parameters passed are as follows. Treat all of them as read-only.

g,m,n,lab,ptn,level: As above.

`int` *numcells*: The number of cells in the current partition.

`int` *tc*: If **nauty** has determined that children of this node need to be explored, *tc* is the index in *lab* of where the target cell starts. Otherwise, it is −1.

`int` *code*: This is the code produced by the refinement and vertex-invariant procedures while refining this partition.

(c) *userautomproc*(*count*, *perm*, *orbits*, *numorbits*, *stabvertex*, *n*)

This is called once for each generator of the automorphism group, in the same order as they are written (see Section 5). It is provided to facilitate such tasks as storing the generators for later use, writing them in some unusual manner, or converting them into another representation (for example, into their actions on the edges).

Suppose the generator is $\gamma = \gamma_i^{(j)}$, in the notation of Section 5. Then the parameters have meanings as below. Treat them all as read-only.

`int` *count*: The ordinal of this generator. The first is number 1.

`permutation` $*$*perm*: The generator γ itself. For $0 \le i < n$, $perm[i] = i^\gamma$.

`int` $*$*orbits*; `int` *numorbits*: The orbits and number of orbits of the group generated by all the generators found so far, including this one. See Section 4 for the format of *orbits*.

`int` *stabvertex*: The value v_j, as defined in Section 5.

`int` *n*: The number of vertices, as usual.

(d) *userlevelproc*(*lab*, *ptn*, *level*, *orbits*, *stats*, *tv*, *index*, *tcellsize*, *numcells*, *childcount*, *n*)

This is called once for each node on the leftmost path downwards from the root, in bottom to top order. It corresponds to the markers "`level ...`", which are described in Section 5, except that an additional, initial, call is made for the first leaf of the tree. The purpose is to provide more information than is provided by the markers, in a manner which enables it to be stored for later use, etc.. The parameters passed are as follows. Treat them all as read-only.

lab,ptn,level,n: As above. The values of *level* will decrease by one for each call, reaching one for the final call.

Suppose that the value of *level* is l.

`int` $*$*orbits*: The orbits of the group generated by all the automorphisms found so far. See Section 4 for the format. In the notation of Section 5, *orbits* gives the orbits of the stabiliser $\Gamma_{v_1,v_2,\ldots,v_{l-1}}$.

`statsblk` $*$*stats*: The meaning is as given in Section 4, except that it applies to the group generated by all the automorphisms found so far, that is to $\Gamma_{v_1,v_2,\ldots,v_{l-1}}$. Only the fields which refer to the group can be assumed correct.

`int` *tv*, *index*, *tcellsize*, *numcells*: In the notation of Section 5, these are the values of v_l, i_l, j_l and c_l, respectively. For the first call, their values are 0, 1, 1 and n, respectively.

`int` *childcount*: This is the number of children of the node at level *level* on the first path down the tree which were actually generated.

The condition $numcells = n$ can be used to identify the first call.

(e) *usertcellproc*(*g*, *lab*, *ptn*, *level*, *numcells*, *tcell*, *size*, *cellpos*, *tc_level*, *hint*, *bestcellproc*, *m*, *n*)

This is a replacement for the default procedure called on to choose a target cell. It is called for every node for which **nauty** has decided children must be generated, after the partition has been refined.

The parameters are as follows. Only *tcell*, *tcellsize* and *cellpos* may be altered.

g,m,n,lab,ptn,level: As above.

`int` *numcells*: The number of cells in the current partition.

`set` **tcell*: This is the address of a `set` of m `setword`s which must be set by the procedure to contain just those vertices in the target cell.

`int` **size*: This must be set by the procedure to the size of the target cell.

`int` **cellpos*: This must be set by the procedure to the position in *lab* where the target cell starts.

`int` *tc_level*: The value of the field of the same name in the *options* parameter passed to **nauty**.

`int` *hint*: If this is ≥ 0, it is a suggestion from **nauty** of a good value for *cellpos* (and thus for *tcell* and *tcellsize*). There is no compulsion to take the hint, but taking it is almost always a good idea. However, you must first verify that the hint is valid in the sense that there is a non-singleton cell which starts at the specified place. If there is not, you must choose a valid cell.

`int` *bestcellproc*`()`: If not NULL, this is a procedure which can be called with argument list `(graph *`*g*`, int *`*lab*`, int *`*ptn*`, int` *level*`, int` *tc_level*`, int` *m*`, int` *n*`)`. The procedure passed by **nauty** is the one given in *options.dispatch–>bestcell*, which for graphs is normally the procedure *bestcell* in `naugraph.c`. It returns the index in *lab/ptn* of a non-trivial cell, or n if there is one.

It is quite central to the validity of the algorithm that a non-singleton cell be chosen (it will always exist). The choice must be entirely independent of the labelling of the vertices. It must also be independent of the position of the node in the search tree to the extent that equivalent nodes are treated equivalently.

The standard way to write *usertcellproc* is like this:

(i) if *hint* indicates a non-trivial cell, choose it; else
(ii) if *level* $\leq$ *tc_level* use *bestcellproc* to choose a cell; else
(iii) choose a non-trivial cell in some other manner (for example, the first such cell).

The *bestcellproc* parameter was added at version 2.0.

8. Vertex-invariants.

As described in Section 2, the operation of **nauty** is driven by a procedure which accepts partitions and attempts to make them strictly finer without separating equivalent vertices. For some families of difficult graphs, the built-in refinement procedure is insufficiently powerful, resulting in excessively large search trees. In many cases, this problem can be dramatically reduced by using some sort of invariant to assist the refinement procedure.

Formally, let $\mathcal{G}$ be the set of labelled graphs (or digraphs) with vertices $V = \{0, 1, \ldots, n{-}1\}$, and let $\mathit{\Pi}$ be the set of partitions of V. As always, the order of the cells of a partition is significant, but the order of the elements of the cells is not. Let $\mathcal{Z}$ be the integers. A *vertex-invariant* is defined to be a mapping

$$\phi \,:\, \mathcal{G} \times \mathit{\Pi} \times V \rightarrow \mathcal{Z}$$

such that $\phi(G^\gamma, \pi^\gamma, v^\gamma) = \phi(G, \pi, v)$ for every $G \in \mathcal{G}$, $\pi \in \mathit{\Pi}$, $v \in V$ and permutation γ. Informally, this says that the values of ϕ are independent of the labelling of G.

A great number of vertex-invariants have been proposed in the literature, but very few of them are suitable for use with **nauty**. Most of them are either insufficiently powerful or require excessive amounts of time or space to compute. Even amongst the vertex-invariants which are known to be useful, their usefulness varies so much with the type of graph they are applied to, or the levels of the search tree at which they are applied, that intelligent automatic selection of a vertex-invariant by **nauty** would seem to be a task beyond our current capabilities. Consequently, the choice of vertex-invariant (or the choice not to use one) has been left up to the user.

The *options* parameter of **nauty** has four fields relevant to vertex-invariants, namely *invarproc*, *mininvarlevel*, *maxinvarlevel* and *invararg*. These are fully described in Section 4. The `I` command in **dreadnaut** may be useful in investigating which of the supplied vertex-invariants are useful for your problem. Experience shows that it is nearly always best to apply the invariant at just one level in the search tree, with levels 1 and 2 being the most likely candidates.

We now describe the vertex-invariants which are provided with **nauty**. Information on how to write a new vertex-invariant procedure can be found in the file `nautinv.c`. We will assume that g is a graph on $V = \{0, 1, \ldots, n{-}1\}$, and that $\pi = (V_0, V_1, \ldots, V_k)$ is a partition of V. This partition will be equitable unless *options.digraph* = TRUE. One of the cells of π will be designated V^*. If the procedure is called by **nauty** at level 1 (i.e. at the root of the search tree), or directly by **dreadnaut** (`I` command), this will be the first cell V_0; otherwise, V^* will be the singleton cell containing the vertex fixed in order to create this node from its parent.

twopaths. Each vertex v is given a code depending on the cells to which belong the vertices reachable from v along a path of length 2. *invararg* is not used. This is a cheap invariant suitable for graphs which are regular but otherwise have no particular structure (for example).

adjtriang. Each vertex v is given a code depending on the number of common neighbours between each pair $\{v_1, v_2\}$ of neighbours of v, and which cells v_1 and v_2 belong to. v_1 must be adjacent to v_2 if *invararg = 0* and not adjacent if *invararg = 1*. This is a fairly cheap invariant which can often break up the vertex sets of strongly-regular graphs.

triples. Each vertex v is given a code depending on the set of weights $w(v, v_1, v_2)$, where $\{v_1, v_2\}$ ranges over the set of all pairs of vertices distinct from v such that at least one of $\{v, v_1, v_2\}$ lies in V^*. The weight $w(v, v_1, v_2)$ depends on the number of vertices adjacent to an odd number of $\{v, v_1, v_2\}$ and to the cells that v, v_1 and v_2 belong to. *invararg* is not used. This invariant often works on strongly-regular graphs that *adjtriang* fails on, but is more expensive.

quadruples. Each vertex v is given a code depending on the set of weights $w(v, v_1, v_2, v_3)$, where $\{v_1, v_2, v_3\}$ ranges over the set of all pairs of vertices distinct from v such that at least one of $\{v, v_1, v_2, v_3\}$ lies in V^*. The weight $w(v, v_1, v_2, v_3)$ depends on the number of vertices adjacent to an odd number of $\{v, v_1, v_2, v_3\}$ and to the cells that v, v_1, v_2 and v_3 belong to. *invararg* is not used. This is an expensive invariant which can sometimes be of use for graphs with a particularly regular structure.

celltrips. Each vertex v is given a code depending on the set of weights $w(v, v_1, v_2)$, where $w(v, v_1, v_2)$ depends on the number of vertices adjacent to an odd number of $\{v, v_1, v_2\}$. These three vertices are constrained to belong to the same cell. The cells of π are tried in increasing order of size until one splits. *invararg* is not used. This invariant can sometimes split the bipartite graphs derived from block designs, and other graphs of moderate difficulty.

cellquads. Each vertex v is given a code depending on the set of weights $w(v, v_1, v_2, v_3)$, where $w(v, v_1, v_2, v_3)$ depends on the number of vertices adjacent to an odd number of $\{v, v_1, v_2, v_3\}$. These four vertices are constrained to belong to the same cell. The cells of π are tried in increasing order of size until one splits. *invararg* is not used. This invariant is powerful enough to split many difficult graphs, such as hadamard-matrix graphs (where it is best applied at level 2).

cellquins. Each vertex v is given a code depending on the set of weights $w(v, v_1, v_2, v_3, v_4)$, where $w(v, v_1, v_2, v_3, v_4)$ depends on the number of vertices adjacent to an odd number of $\{v, v_1, v_2, v_3, v_4\}$. These five vertices are constrained to belong to the same cell. The cells of π are tried in increasing order of size until one splits. *invararg* is not used. We know of no good use for this very powerful but very expensive invariant.

distances. Each vertex v is given a code depending on the number of vertices at each distance from v, and what cells they belong to. If a cell is found that splits, no further cells are tried. *invararg* specifies an upper bound on which distance to investigate, with 0 indicating no limit. This is a fairly cheap invariant suitable for things like regular graphs for which *twopaths* doesn't work.

indsets. Each vertex v is given a code depending on the number of independent sets of size *invararg* which include v, and the cells containing the other vertices of those sets. The value of *invararg* is limited to 10. This can often split the vertex sets of strongly-regular graphs and bipartite design graphs, though it becomes expensive if *invararg* is large. A value of 4 is sometimes sufficient.

cliques. Each vertex v is given a code depending on the number of cliques of size *invararg* which include v, and the cells containing the other vertices of those cliques. The value of *invararg* is limited to 10. This can often split the vertex sets of strongly-regular graphs, though it becomes expensive if *invararg* is large. A value of 4 is sometimes sufficient.

cellcliq. Each vertex v is given a code depending on the number of cliques of size *invararg* which include v and lie within the cell containing v. The value of *invararg* is limited to 10. The cells are tried in increasing order of size, and the process stops as soon as a cell splits. This invariant applied at level 2 can be very successful on difficult vertex-transitive graphs. A value of 3 can sometimes work even on strongly-regular graphs.

cellind. Each vertex v is given a code depending on the number of independent sets of size *invararg* which include v and lie within the cell containing v. The value of *invararg* is limited to 10. The cells are tried in increasing order of size, and the process stops as soon as a cell splits. This invariant applied at level 2 can be very successful on difficult vertex-transitive graphs.

adjacencies. This is an invariant for digraphs and is not useful for graphs. The standard refinement procedure alone can sometimes give very poor performance for directed graphs, especially those which are not strongly connected. This invariant tries to correct the poor behaviour. Applying it to multiple levels may be necessary.

cellfano. This invariant is intended for projective plane graphs but can be applied to any graphs. It is very expensive.

cellfano2. This invariant is intended for projective plane graphs but can be applied to any graphs. It is very expensive, but maybe less than *cellfano* for genuine projective plane graphs. In the latter case, it can be thought of as counting the Fano subplanes according to which cells they involve.

9. Writing programs which call *nauty*. A complete example of a program calling **nauty** can be found in Appendix A. There are two versions, one using an explicit positive value of MAXN, and one using dynamic allocation.

Programs which call **nauty** should include the file `nauty.h`. As well as defining the relevant types and parameters, this file also declares macros and procedures which are of use in constructing the arguments, and declares some useful tables.

Suppose that m and n have meanings as usual.

There are two general approaches. The first, the simplest if a prior limit is known on the graph size, is to define MAXN to be that limit before `nauty.h` is included. `nauty.h` will define MAXM, and then MAXN and MAXN can be used to declare variables. For example:

```
set   s[MAXM];  /* a set */
graph   g[MAXN*MAXM];  /* a graph */
int   xy[MAXN];  /* an array */
```

The second method is more complicated but does not require a prior bound on the graph size. In this method, each variable whose size is unknown is dynamically allocated. Of course you can do this yourself using `malloc()` but `nauty.h` provides macros for doing it in a convenient and efficient way. First there are static declarations:

```
DYNALLSTAT(set,s,s_sz);
DYNALLSTAT(graph,g,g_sz);
DYNALLSTAT(int,xy,xy_sz);
```

Before the variables are used, they are set to the right size using the dynamic allocation macros:

```
DYNALLOC1(set,s,s_sz,m,"malloc");
DYNALLOC2(graph,g,g_sz,m,n,"malloc");
DYNALLOC1(int,xy,xy_sz,n,"malloc");
```

To take the first variable as an example, the result of the macro will be that `s` has a value of type `set*` which points to an array of length at least m. If DYNALLSTAT is used again for the same variable, it is freed and allocated again only if the new requested size is larger than the previous size. This is intended to be more efficient that repeated unnecessary calls to `malloc()` and `free()`. In case it is desired to free the object allocated by DYNALLOC1, use, for example, `DYNFREE(s,s_sz)`. There is also CONDYNFREE that frees objects if they are bigger than a given size.

In the case of g, we used `DYNALLOC2` instead of `DYNALLOC1`. This is slightly better as it covers the possibility that mn is too large for an `int`. We could also use

```
DYNALLOC1(graph,g,g_sz,m*(size_t)n,"malloc");
```

`nauty.h` also defines a number of macros that are useful for programming with the **nauty** data structures. Some of the more useful macros are as follows.

ADDELEMENT(s,i) : add element i to set s.

DELELEMENT(s,i) : delete element i from set s.

FLIPELEMENT(s,i) : delete element i from set s if it is present, or insert it if it is absent.

ISELEMENT(s,i) : test if i is an element of the set s ($0 \le i \le n{-}1$).

EMPTYSET(s,m) : make the set s equal to the empty set.

POPCOUNT(x) : the number of 1-bits in the `setword` x. Use `(x ? POPCOUNT(x) : 0)` in circumstances where x is most often zero.

FIRSTBIT(x) : the position (0 to WORDSIZE − 1) of the first (least-numbered) 1-bit in the `setword` x, or WORDSIZE if there is none.

TAKEBIT(i,x) : If the `setword` x is not 0, set i to the position (0 to WORDSIZE − 1) of the first (least-numbered) 1-bit in x, and remove that bit from x.

ALLBITS : A `setword` constant with the first WORDSIZE bits set (this is usually all the bits).

BITMASK(i) : A `setword` constant with the first i+1 bits unset and the other WORDSIZE$-i-1$ numbered bits set, for $0 \leq i <$ WORDSIZE. Thus, *ANDing* a `setword` with BITMASK(i) deletes bits $0..i$.

Some of the procedures in `nautil.c` or `naugraph.c` may be useful. They are declared in `nauty.h`. See the source code for the parameter list and semantics of these:

nextelement : find the position of the next element in a set following a specified position. The recommended way to do something for each element of the set s is like this:

```
  for (i = -1; (i = nextelement(s,m,i)) >= 0;)
      Process element i.
```

permset : apply a permutation to a set.

orbjoin : update the orbits of a group according to a new generator.

writeperm : write a permutation to a file.

isautom : test if a permutation is an automorphism.

updatecan : (for *samerows* = 0) relabel a graph.

refine : find coarsest equitable partition not coarser than given partition.

refine1 : produces exactly the same results as *refine*, but assumes $m = 1$ for greater speed.

The file `naututil.c` contains procedures which are used by the **dreadnaut** program (see Section 12). Many of these are also useful to programs which call **nauty**. If your program uses them, include `naututil.h` instead of `nauty.h`.

Some of the more useful procedures are:

setsize : find cardinality of set.

setinter : find cardinality of intersection of two sets.

putset : write a set to a file.

putgraph : write a graph to a file.

putorbits : write a set of orbits to a file.

putptn : write a partition to a file.

readgraph : read a graph from a file.

readptn : read a partition from a file.

ranperm : generate a random permutation.

rangraph : generate a random graph.

mathon : perform a doubling operation, as defined in [3].

complement : take the complement of a graph.

converse : take the converse of a digraph.

cellstarts : find the places where the cells at a given level begin.

sublabel : extract an induced subgraph of a graph.

In addition, the file `nautaux.c` contains a few procedures which manipulate graphs or partitions, but which are not currently used by **nauty** or **dreadnaut**.

It is recommended that programs which call **nauty** use the call
nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);
which will verify that a compatible verion of **nauty** is being used.

10. Installing *nauty* and *dreadnaut*.

First, read the file `README` to see if there is information more recent than this manual.

There are a number of source files provided. **nauty** by itself requires at least the files `nauty.h`, `nauty.c`, `nautil.c` and `naugraph.c`. The provided invariants are in `nautinv.c`. The **dreadnaut** program requires, in addition, files `rng.c`, `naututil.h`, `naututil.c` and `dread-naut.c`.

Starting at version 2.1, **nauty** does not try too hard to support very old or broken compilers. In particular, the basic facilities of ANSI C such as *void* and function prototypes are assumed. The files of **nauty** itself should compile with any ANSI-compliant compiler. On the other hand, **dreadnaut** expects a few library functions that might not be available in non-Unix systems.

If are using a Unix-like system (one that has a working shell and preferably `make`), the preferred way to compile **nauty** is to run the shell script

```
./configure
```

This will examine your system and create the files `nauty.h`, `naututil.h`, `gtools.h` and `make-file` in a way that is (hopefully) compatible. Then you can compile **nauty** using

```
make nauty
```

If you are on a system where these tools are not available, but you have a half-decent C compiler, you should start by editing the definitions near the start of `nauty.h`, `naututil.h` and `gtools.h`. (Most should be OK already.) Then you can compile using the commands in `makefile` as a guide. If you have trouble, please advise the author of the details. Similarly, please tell us if you can improve the operation on non-Unix systems.

This procedure will create two editions of **dreadnaut**, namely `dreadnaut` (n limited to 32765) and `dreadnautB` (n limited to 16777213). The makefile also knows how to make other variants of **nauty**; here is a list of all of them:

`nauty.o` etc: MAXN = 0 (n limited to 32765)

`nauty1.o` etc: MAXN = WORDSIZE (n limited to the word size)

`nautyB.o` etc: MAXN = 0, BIGNAUTY (n limited to 16777213)

`nautyW.o` etc.: MAXN = 0, WORDSIZE = 32

`nautyW1.o` etc.: MAXN = WORDSIZE = 32

`nautyS.o` etc.: MAXN = 0, WORDSIZE = 16

`nautyS1.o` etc.: MAXN = WORDSIZE = 16

`nautyL.o` etc.: MAXN = 0, WORDSIZE = 64

`nautyL1.o` etc: MAXN = WORDSIZE = 64

The last two variants can only be used if your compiler supports either `unsigned long` or `unsigned long long` type as a 64-bit integer.

There are some test files included in the package. To run these, just use

```
make checks
```

which should not produce any output that looks like an error message.

11. Efficiency.

We give some sample execution times for a Pentium III processor at 1 GHz, using gcc with full optimisation, using default options unless otherwise specified.

For random graphs with edge probability 1/2, experimental execution times for large n are about n^2 nanoseconds with *options.getcanon* = FALSE, and $12\,n^2$ nanoseconds with *options.getcanon* = TRUE. The large difference between these times for large n is almost entirely taken up by the process of permuting the entries of g to get *canong*. Except for very small n, nearly all random graphs have only discrete equitable partitions, and thus have trivial automorphism groups. All **nauty** does in this case is one refinement operation followed, if *options.getcanon* = TRUE, by one relabelling operation.

The 46308 vertex-transitive graphs of order 30 require 0.14 milliseconds each on average, irrespective of the value of *options.getcanon*. An average of about 4 generators each are found for the automorphism groups.

A list of difficult graphs is given by Mathon in [3]. Using his notation for them, we find the following times with *options.getcanon* = TRUE:
A_{25}–B_{25}: 0.00015 seconds;
A_{50} and B_{50}: 0.029 seconds (0.006 seconds using vertex-invariant *cellquads* at level 1);
A^1_{25}: 0.0008 seconds (0.00014 seconds using vertex-invariant *adjtriang* at level 1);
B^1_{25}: 0.0026 seconds (0.00014 seconds using vertex-invariant *adjtriang* at level 1);
A_{35}–D_{35}: 0.008 seconds (0.00018 seconds using vertex-invariant *cliques* with parameter 4 at level 1);
A_{52}: 0.009 seconds (0.0014 seconds using vertex-invariant *cliques* with parameter 5 at level 1);
B_{52}: 0.055 seconds (0.0016 seconds using vertex-invariant *cliques* with parameter 5 at level 1);
A_{72}–D_{72}: 0.58 seconds (0.15 seconds using vertex-invariant *quadruples* applied at level 1).
The execution time for these graphs varies somewhat with the initial labelling.

Amongst the most difficult known graphs for this algorithm, and probably for most other similar algorithms, are certain bipartite graphs derived from Hadamard matrices. For example, some of these graphs on 96 vertices require more than 5 seconds to process. However, with the vertex-invariant *cellquads* applied at level two, this time is reduced to less than 1 second.

A family of strongly-regular graphs with 155 vertices and trivial automorphism group require 0.76 seconds with no vertex-invariant and 0.01 seconds with the vertex-invariant *adjtriang* (or *cliques* with parameter 4) applied at level 1. A similar family with 1027 vertices require 264 seconds with no vertex-invariant and 1.3 seconds with the vertex-invariant *adjtriang* (or *cliques* with parameter 4) applied at level 1.

As examples of how **nauty** performs for very rich automorphism groups, we mention $L(K_{30})$ (435-vertex linegraph of complete graph; group size 30!; execution time 0.1 seconds; 29 generators) and the 1-skeleton of the 11-cube (2048-vertex graph; group size 81,749,606,400; execution time 11 seconds; 11 generators).

12. *dreadnaut*.

dreadnaut is a simple program which can read graphs and execute **nauty**. It is only a very primitive interface with few facilities. If you want to use **nauty** in a richer interactive environment, some of your choices are:

(a) Magma: `http://magma.maths.usyd.edu.au/magma/`

(b) GAP with GRAPE: `http://www-gap.dcs.st-and.ac.uk/~gap/Share/grape.html`
(c) LINK: `http://dimacs.rutgers.edu/~berryj/LINK.html`
(d) Vega: `http://vega.ijp.si/Htmldoc/Vega03.html`

Input is taken from the standard input and output is sent to the standard output, but this can be changed by using the "`<`" and "`>`" commands. Commands may appear any number per line separated by white space, commas, semicolons or nothing. They consist of single characters, sometimes followed by parameters.

At any point of time, **dreadnaut** knows the following information:
(a) The number of vertices, n.
(b) The "current graph" g, if defined.
(c) The "current partition" π, if defined.
(d) The orbits of the (coloured) graph (g, π), if defined.
(e) The canonically labelled isomorph of g, called h, if defined. (Also called *canong.*)
(f) An extra graph called h', if defined. (Also called *savedg.*)
(g) Values for each of a variety of options.

In the following '`#`' is an integer and '`=`' is optional.

(A) Commands which define or examine the graph g.

`n=#` Set value of n. The maximum value is installation-defined.

`g` Read the graph g.
There is always a "current vertex" which is initially the first vertex. (Vertices are numbered from 0 unless you have used the `$` command.) The number of the current vertex is displayed as part of the prompt, if any. Available subcommands:
`#` : add an edge from the current vertex to the specified vertex. (Unless you have selected the option *digraph*, edges only need to be entered in one direction.)
`-#` : delete the edge, if any, from the current vertex to the specified vertex.
`;` : increment the current vertex. If it becomes too high for a vertex label, stop.
`#:` : make the specified vertex the current vertex.
`?` : display the neighbours of the current vertex.
`.` : stop.
`!` : ignore the rest of this input line.
`,` : ignored.

`e` Edit the graph g. The available subcommands are the same as for the "`g`" command.

`r ... ;` Relabel the graph g, where '...' is a permutation of $\{0, 1, \ldots, n-1\}$, specifying the order in which to relabel the vertices, followed by a semicolon. Missing numbers are filled in at the end in numerical order. For example, for $n = 5$, "`r 4,1;`" is equivalent to "`r 4,1,0,2,3;`". The partition π is permuted consistently.

`R ... ;` This is the same as `r` except that unspecified vertices are not filled in. Instead, a subgraph corresponding to the given vertices is formed and replaces g. If the command is given as `-R`, the given vertices are deleted instead. The partition π is reset to have only one cell.

`j` Relabel the graph g at random. The partition π is permuted consistently.

`%` Perform the doubling operation $E(g)$ defined in [3]. The result in g is a regular graph with order $2n + 2$ and degree n.

`s=#` Generate graph (or digraph) g at random with independent edge probabilities $1/i$, where i is the integer specified.

`_` (underscore) Replace the graph g by its complement. If there are any loops, the set of loops is complemented too; otherwise, no loops are introduced.

`__` (two underscores) If g is a digraph, take its converse (which reverses the direction of all the edges). Otherwise do the same as `_`.

`t` Type the graph g, in an obvious format. The value of option *linelength* is taken into account. The format used is consistent with the input format allowed by the "`g`" command. To examine just some of the graph, you can use the "`?`" subcommand within the "`e`" command.

`T` This is exactly like "`t`" except that a line of the form "`n=`n `$=`l `g`" is written first, where n is the number of vertices and l is the number of the first vertex, and a line of the form "`$$`" is written afterwards. This enables you to save a graph to a file and easily restore it later: "`>newgraph.dre T ->`" will save g to the file *newgraph.dre*, while "`<newgraph.dre`" will restore it.

`v` Display the degrees of each vertex of the graph g, if defined. For digraphs, the outdegrees are displayed.

(B) Commands which define the partition π.

`f` Specify an initial partition.
"`-f`" selects the partition with only one cell, which is the default.
"`f=#`" selects the partition with one cell containing just the vertex named and one cell containing every other vertex.
"`f=[ ... ]`" selects an arbitrary partition. Replace "`...`" by a list of cells separated by "`|`". You can use the abbreviation "x`:`y" for the range $x, x+1, \ldots, y$. Any vertices not named are put in a cell of their own at the end.
Example: If $n = 10$, then "`f=[3:7 | 0,2]`" establishes the partition $[3,4,5,6,7 \mid 0,2 \mid 1,8,9]$.

`i` Perform a refinement operation, replacing the partition π by its refinement. The *active* set initially contains every cell.

`I` Perform a refinement operation, an application of the vertex-invariant (if one has been selected using the `*` command), and (if any cells were split) another refinement operation. The final partition becomes π. The behaviour may be modified by the `K` command, but not by the `k` command.
This is useful for determining whether an invariant is effective for a particular graph. Note that you need to restore the partition between repeated tests.

(C) Commands which establish or examine options.

`$=#` Establish an origin for vertex numbering. The default is 0. Only non-negative values are permitted. All the input-output routines used by **nauty** or **dreadnaut** respect this value, even though internally vertices are always numbered from 0. (The value given is copied into the global `int` variable *labelorg*, which is described in Section 4.)

`$$`	Restore the vertex numbering origin to what it was just before the last `$` command. Only one previous value is remembered.
`l=#`	Set value of option *linelength* : the length of the longest line permitted for output. The default value is installation-dependent (typically 78).
`w=#`	Set value of *worksize* : the amount of space provided for **nauty** to store automorphism data. The maximum value is installation-defined, and the default is the same as the maximum. There's little reason to ever use this command.
`+`	Ignored. Provided for contrast with "-".
`d,-d`	Set option *digraph* to TRUE or FALSE, respectively. You must set it to TRUE if you wish to define g to be a digraph or a graph with loops. The default is FALSE. Changing it from TRUE to FALSE also causes the graph g to become undefined, as a safety measure.
`c,-c`	Set option *getcanon* to TRUE or FALSE, respectively. This tells **nauty** whether to find a canonical labelling or just the automorphism group. The default is FALSE.
`a,-a`	Set option *writeautoms* to TRUE or FALSE, respectively. This tells **nauty** whether to display the automorphisms it finds. The default is TRUE.
`m,-m`	Set option *writemarkers* to TRUE or FALSE, respectively. This tells **nauty** whether to display the level markers "`level` ...". See Section 5 for their meaning. The default is TRUE.
`p,-p`	Set option *cartesian* to TRUE or FALSE, respectively. This tells **nauty** to use the "cartesian" form when writing automorphisms. Precisely, the automorphism γ is displayed as a list $v_1^\gamma\ v_2^\gamma\ \ldots v_n^\gamma$, where $v_1, v_2, \ldots, v_n$ are the vertices of g. The default is FALSE.
`y=#`	Set the value of option *tc_level*. A value of `#` tells **nauty** to use an advanced, but expensive, algorithm for choosing target cells in the top k levels of the search tree. See Section 4 for a more detailed description. The default is 100, but setting it to 0 might speed up the average time for easy graphs.
`*=#`	Select a vertex-invariant. One user-defined vertex-invariant can be linked with **dreadnaut** if its name is provided in the preprocessor variable INVARPROC. The argument to the $*$ command is interpretted thus:

`-1`	: the user-defined procedure (if any)
`0`	: no vertex-invariant (this is the default)
`1`	: *twopaths*
`2`	: *adjtriang*
`3`	: *triples*
`4`	: *quadruples*
`5`	: *celltrips*
`6`	: *cellquads*
`7`	: *cellquins*
`8`	: *distances*
`9`	: *indsets*
`10`	: *cliques*
`11`	: *cellcliq*
`12`	: *cellind*
`13`	: *adjacencies*
`14`	: *cellfano*

`15` : *cellfano2*
These procedures are described in Section 8. The default behaviour is for the invariant to be applied only at the root of the tree, but this can be modified using the `k` command. The `K` command can be used to change the invariant parameter, if there is one. The default is `K=3` for *indsets*, *cliques*, *cellind* and *cellcliq*; and `K=0` for everything else.

`k=# #` (Two integer arguments.) Define values for the options *mininvarlevel* and *maxinvarlevel*. These tell **nauty** the minimum and maximum levels of the tree at which it is to apply the vertex-invariant. The root of the tree is at level 1. See Section 4 for a little more information about these options. The default is `k = 0 1`, which causes the invariant to be applied only at the top of the search tree.

`K=#` Give a value to the *invararg* option. This number is passed to the vertex-invariant by the `I` command and by **nauty**. See Section 8 for the meaning of this option for each available vertex-invariant. The default value depends on the invariant; see the $*$ command.

`u=#` Request calls to user-defined functions. The value is
1 for *usernodeproc*,
2 for *userautomproc*,
4 for *userlevelproc*,
8 for *usertcellproc*,
16 for *userrefproc*.
These can be added together to select more than one procedure. The procedures called are those named by the compile-time symbols USERNODE, USERAUTOM, USERLEVEL, USERTCELL and USERREF defined in `dreadnaut.c`. The default values are:
USERNODE: For each node, print a number of dots equal to the depth, then (*numcells*/*code*/*tc*) where *numcells* is the number of cells, *code* is the code produced by the refinement procedure, and *tc* is the position in *lab* where the target cell starts. For the first path down the tree, the partition is displayed as well.
USERAUTOM: For each automorphism, display the arguments *numorbits* and *stabvertex* (see Section 7).
USERLEVEL: For each level, display the arguments *tv*, *index*, *tcellsize*, *numcells* and *childcount*, as well as the fields *numnodes*, *numorbits* and *numgenerators* of *stats*. See Section 7 for what they mean.
USERTCELL: Do nothing.
USERREF: Do nothing.

? Type the current values of m, n, *worksize*, most of the options, the number of edges in g, and the number of cells in π. If output has been directed away from `stdout` using the ">" command, some of this information is also written to `stdout`.

& Type the current partition π, unless it is has only one cell.

&& Same as &, except that the quotient of g with respect to π is also written. Say $\pi = (V_0, V_1, \ldots, V_m)$ and let v_i be the least numbered vertex in V_i for $0 \le i \le m$. Then, for each i, this command writes v_i, then $|V_i|$ in brackets, then the numbers $k_0, k_1, \ldots, k_m$, where k_j is the number of edges from v_j to V_i. The value 0 is written as "-", while the value $|V_i|$ is written as "$*$".

(D) Commands which execute **nauty** or use the results.

`x` Execute **nauty**. Depending on the values of the *writeautoms* and *writemarkers* options, the automorphism group will be displayed while **nauty** is running. See Section 5 for an explanation of the output. When **nauty** returns, **dreadnaut** will display some statistics about it. See Section 4 for the meanings; the important ones are the order of the group and the number of orbits. Depending on your system, the execution time is also displayed.

`@` Copy h, if defined, to h'. See the description of the `#` command for more.

`b` Type the canonical label and the canonically labelled graph. The canonical label is given in the form of a list of the vertices of g in canonical order. Only possible after `x` with option *getcanon* selected.

`z` Type two 8-digit hex numbers whose value depends only on h. This allows quick comparison between graphs. Isomorphic graphs give the same value. In principle, non-isomorphic graphs may also give the same value but we don't know any examples (please tell us if you find one). Only possible after `x` with option *getcanon* selected.

`#` Compare the labelled graphs h and h'. Both must have been already defined (using `x` and `@`). The complete process for testing two graphs g_1 and g_2 for isomorphism is this:
enter g_1;
`c x @` (select *getcanon* option, execute **nauty**, copy h to h');
enter g_2;
`x #` (execute **nauty**, compare h to h').

`##` This is the same as `#` except that, if h is identical to h', you will also be given an isomorphism from g_1 to g_2. This is in the form of a sequence of pairs v_i-w_i, where v_i is a vertex of g_1 and w_i is a vertex of g_2. The vertex-numbering origin in force when h' was created is used for g_1, whilst the origin now in force is used for g_2.

`o` Type the orbits of the group. Only possible after `x`.

(F) Miscellaneous commands.

`h,H` Help: type a summary of **dreadnaut** commands.

`"..."` Anything between the quotes is simply copied to the output. The ligatures '`\n`' (newline), '`\t`' (tab), '`\b`' (backspace), '`\r`' (carriage return), '`\f`' (formfeed), '`\\`' (backslash), '`\'`' (single quote) and '`\"`' (double quote) are recognised. Other occurrences of '`\`' are ignored.

`!` Ignore anything else on this input line. Note that this is a command, not a comment character in the usual sense, so you can't use it in the middle of other commands.

`<` Begin reading input from another file. The name of the file starts at the first non-white character after the "`<`" and ends before the next white character. If such a file cannot be found, another attempt is made with the string "`.dre`" appended to the name. When end-of-file is encountered on that file, continue from the current input file. The allowed level of nesting is system-dependent (usually 8).

`>,>>` Close the existing output file unless it is the standard output, then begin writing output to another file. The name of the file starts at the first non-white character after the "`>`" and ends before the next white character. For "`>`" the file starts off empty. For "`>>`", if an existing file of the right name exists, it is written to starting at the current end-of-file. Use "`->`" to direct output back to the standard output.

`M=#` Each call to **nauty** is performed `#` times and the average cpu time is then reported accurately. This is for doing timing tests with easy graphs.

`q` Quit. **dreadnaut** will exit irrespective of which level of input nesting it is on.

The canonical labellings produced by **dreadnaut** can depend on the values of many of the options. If you are testing two or more graphs for isomorphism, it is important that you use the same values of these options for all your graphs. In general, h is a function of all these:
(a) option *digraph* (`d` command)
(b) all the vertex-invariant options ($*$, `k` and `K` commands)
(c) the value of *tc_level* (`y` command)
(d) the use of *usertcellproc* or *userrefproc* (`u` command)
(e) the version of **nauty** used

Beginning at version 2.1, the canonical labelling does not depend on the compiler, the system, or the word size.

Several sample **dreadnaut** sessions are shown below. The first problem solved is the second example in Section 6. The underlined characters are those typed by the user.

```
>  n=8 g                             8 vertices
0:  1 3 4;                           enter the graph
1:  2 5;
2:  3 6;
3:  7;
4:  5 7;
5:  6;
6:  7.
>  f=2 x                             fix vertex 2; execute
[fixing partition]
(0 5)(3 6)
level 2:  6 orbits; 3 fixed; index 2
(1 3)(5 7)
level 1:  4 orbits; 1 fixed; index 3
4 orbits; grpsize=6; 2 gens; 6 nodes; maxlev=3
tctotal=7; cpu time = 0.00 seconds
>  o                                 show the orbits
 0 5 7; 1 3 6; 2; 4;
>  q                                 quit
```

The next problem solved is to determine an isomorphism between the graphs of examples 3 and 4 of Section 6. We turn off the writing of automorphisms to save some space.

```
>  c -a -m                  turn  getcanon on, group writing off
>  n=12 g                   enter the first graph
0:  1; 2; 0;
3:  4; 5; 6; 3;
7:  8; 9; 10; 11; 7.
>  x @                       execute, save the result
```

```
3 orbits; grpsize=480; 6 gens; 31 nodes (3 bad leaves); maxlev=7
tctotal=88; canupdates=1; cpu time = 0.00 seconds
> g                         enter the second graph
0:  1; 2; 3; 4; 0;
5:  6; 7; 8; 5;
9:  10; 11; 9.
> x                         execute
3 orbits; grpsize=480; 6 gens; 50 nodes (2 bad leaves); maxlev=7
tctotal=124; canupdates=4; cpu time = 0.00 seconds
> ##                        compare to saved graph
h and h' are identical.
 0-9 1-10 2-11 3-5 4-6 5-7 6-8 7-0 8-1 9-2 10-3 11-4
```

As a third example, we consider a simple block design. **nauty** can compute automorphisms and canonical labellings of block designs by the common method of converting the design to an equivalent coloured graph. Suppose a design D has varieties x_1, x_2, …, x_v and blocks B_1, B_2, …, B_b. Define $G(D)$ to be the the graph with vertex set $\{x_1, \ldots, x_v, B_1, \ldots, B_b\}$, with each x-vertex having one colour and each B-vertex having a second colour, and edge set $\{x_iB_j \mid x_i \in B_j\}$. The following theorem is elementary.

Theorem.

(a) The automorphism group of D is isomorphic to the automorphism group of $G(D)$.

(b) If D_1 and D_2 are designs, D_1 and D_2 are isomorphic if and only if $G(D_1)$ and $G(D_2)$ are isomorphic. ∎

Consider the design $D = \{\{1,2,4\},\{1,3\},\{2,3,4\}\}$. Label $G(D)$ so that the varieties of D correspond to vertices 1–4, while the blocks correspond to vertices 5–7.

```
> $=1                             label vertices starting at 1
> n=7 g
1: 5:                             go to vertex 5 (block 1)
5: 1 2 4;
6: 1 3;
7: 2 3 4.
> f=[1:4]                         fix the varieties setwise
> cx                              run nauty
[fixing partition]
(2 4)                                           group generators
level 2:  6 orbits; 2 fixed; index 2
(1 3)(5 7)
level 1:  4 orbits; 1 fixed; index 2
4 orbits; grpsize=4; 2 gens; 6 nodes; maxlev=3
tctotal=6; canupdates=1; cpu time = 0.00 seconds
> o                               display the orbits
 1 3; 2 4; 5 7; 6;
> b                               display the canonical labelling
```

```
 1 3 2 4 6 5 7                                the vertices in canonical order
  1 :  5 6;                                   the relabelled graph
  2 :  5 7;
  3 :  6 7;
  4 :  6 7;
  5 :  1 2;
  6 :  1 3 4;
  7 :  2 3 4;
> q                          quit
```

For many families of block designs, it can be proved that the isomorphism class of each design is uniquely determined by the isomorphism class of its block-intersection graph, where that graph has the blocks as vertices and pairs of intersecting blocks as edges. For $(v, k, 1)$-designs, a sufficient condition for this is that $v > k(k^2 - 2k + 2)$. On the occasions when this is true, **nauty** can usually process the block-intersection graphs more quickly than it can process the designs directly. Also, the vertex-invariants described in Section 8 are more likely to be successful with the block-intersection graphs.

13. *gtools.*

A package of programs called **gtools** is distributed along with **nauty**. The main purpose of the package is to provide efficient processing of files of graphs stored in `graph6` or `sparse6` format. These formats are defined in the file `formats.txt`. Support for **gtools** is limited to Unix, but they may run on other systems which provide basic Unix system calls. The program `shortg` requires a program compatible with the Unix `sort` program, as well as pipes.

To compile **gtools** under Unix, just use `make gtools`. All the **gtools** programs are self-documenting: just execute with the option `-help` to see an explanation of all the features. We only list the basic functions of the programs here:

`geng`	: generate small graphs
`genbg`	: generate small bicoloured graphs
`directg`	: generate small digraphs with given underlying graph
`multig`	: generate small multigraphs with given underlying graph
`genrang`	: generate random graphs
`copyg`	: convert format and select subset
`labelg`	: canonically label graphs
`shortg`	: remove isomorphs from a file of graphs
`listg`	: display graphs in a variety of forms
`showg`	: a stand-alone version of `listg`
`amtog`	: read graphs in adjacency matrix form
`dretog`	: read graphs in dreadnaut form
`complg`	: complement graphs
`catg`	: concatenate files of graphs
`addedgeg`	: add an edge in each possible way
`deledgeg`	: delete an edge in each possible way

`newedgeg` : in each possible way, subdivide two non-adjacent edges and join the two new vertices

`NRswitch` : switch the edges between the neighbourhood and the complementary neighbourhood, for each vertex

`countg` : count graphs according to a variety of properties

`pickg` : select graphs according to a variety of properties

`biplabg` : label bipartite graphs so the colour classes are contiguous

Further programs will be added. Requests are welcome.

14. Recent changes.

This section lists all the significant changes made to **nauty** or **dreadnaut** since version 1.5. For a complete list of even trivial changes, see the source code.

(A) <u>Changes to the user view of **dreadnaut**</u>.

(a) The commands `R`, `_ _` and `M` have been added.

(b) Several new invariants have been added.

(B) <u>Changes affecting programs which call **nauty**</u>.

(a) The graph-specific contents of the file `nautil.c` have been moved into the new source file `naugraph.c`. This is to support the use of **nauty** with non-graph objects.

(b) The *options* parameter has grown some extra fields. To ease future changes like this, use the DEFAULTOPTIONS macro to declare the actual parameter.

(f) Dynamic allocation for all large variables has been added in the case that MAXN=0.

(g) Support for 64-bit compilation using `long long` if available.

(j) Some features of ANSI C are now assumed, including function prototypes. If you want to use **nauty** with a very old compiler, try version 1.9 or earlier.

(k) The set macros like ADDELEMENT now work for sets of arbitrary size even if you are compiling with MAXN $\leq$ WORDSIZE. To get the old behaviour (where a more efficient definition is used in the latter case), define the preprocessor variable ONE_WORD_SETS.

(l) Minor changes: extra parameter in *usertcellproc*, no need to use EXTDEFS, null pointers like NILSET are obsolete (just use NULL), INFINITY has been renamed to NAUTY_INFINITY.█

15. References.

[1] B. W. Kernighan and D. M. Ritchie, The C programming language (Prentice-Hall, Englewood Cliffs, 1978).

[2] A. Kirk, Efficiency considerations in the canonical labelling of graphs, Technical report TR-CS-85-05, Computer Science Department, Australian National University (1985).

[3] K. E. Malysiak, Graph Isomorphism, Canonical Labelling and Invariants, Honours Thesis, Computer Science Department, Australian National University (1987).

[4] R. Mathon, Sample graphs for isomorphism testing, *Congressus Numerantium* **21** (1978) 499–517.

[5] B. D. McKay, Practical graph isomorphism, *Congressus Numerantium* **30** (1981) 45–87. Available at `http://cs.anu.edu.au/~bdm/nauty/PGI/`.

Appendix. A Sample programs which call **nauty**.

We give two sample programs which generate a graph (a polygon) and call **nauty** to display its automorphism group.

The first program uses a fixed positive value of MAXN so is limited to that size. The second program uses dynamic allocation and so works with much larger sizes.

In each case, you need `nauty.c`, `nautil.c` and `naugraph.c` as well.

Note the calls `nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);` These are not essential, but have the desirable feature of checking that you linked the program with a compatible version of **nauty**.

These programs are in the **nauty** distribution as `nautyex.c` and `nautyex2.c`.

```
/* This program prints generators for the automorphism group of an
   n-vertex polygon, where n is a number supplied by the user.
   It needs to be linked with nauty.c, nautil.c and naugraph.c.

   This version uses a fixed limit for MAXN.
*/

#define MAXN 100
#include "nauty.h"   /* which includes <stdio.h> */

main()
{
    graph g[MAXN*MAXM];
    int lab[MAXN],ptn[MAXN],orbits[MAXN];
    static DEFAULTOPTIONS(options);
    statsblk(stats);
    setword workspace[50*MAXM];

    int n,m,v;
    set *gv;

    options.writeautoms = TRUE;

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
            if (n > MAXN)
            {
                printf("n must be in the range 1..%d\n",MAXN);
                exit(1);
            }

            m = (n + WORDSIZE - 1) / WORDSIZE;
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

            for (v = 0; v < n; ++v)
            {
                gv = GRAPHROW(g,v,m);

                EMPTYSET(gv,m);
                ADDELEMENT(gv,(v+n-1)%n);
                ADDELEMENT(gv,(v+1)%n);
            }

            printf("Generators for Aut(C[%d]):\n",n);
            nauty(g,lab,ptn,NULL,orbits,&options,&stats,
                                        workspace,50*MAXM,m,n,NULL);
        }
        else
            break;
    }
}
```

```
/* This program prints generators for the automorphism group of an
   n-vertex polygon, where n is a number supplied by the user.
   It needs to be linked with nauty.c, nautil.c and naugraph.c.
   This version uses dynamic allocation.
*/

#include "nauty.h"    /* which includes <stdio.h> */

main()
{
    DYNALLSTAT(graph,g,g_sz);
    DYNALLSTAT(int,lab,lab_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    static DEFAULTOPTIONS(options);
    statsblk(stats);
    setword workspace[100];

    int n,m,v;
    set *gv;

    options.writeautoms = TRUE;

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
            m = (n + WORDSIZE - 1) / WORDSIZE;
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

            DYNALLOC2(graph,g,g_sz,m,n,"malloc");
            DYNALLOC1(int,lab,lab_sz,n,"malloc");
            DYNALLOC1(int,lab,lab_sz,n,"malloc");
            DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
            DYNALLOC1(int,orbits,orbits_sz,n,"malloc");

            for (v = 0; v < n; ++v)
            {
                gv = GRAPHROW(g,v,m);

                EMPTYSET(gv,m);
                ADDELEMENT(gv,(v+n-1)%n);
                ADDELEMENT(gv,(v+1)%n);
            }

            printf("Generators for Aut(C[%d]):\n",n);
            nauty(g,lab,ptn,NULL,orbits,&options,&stats,
                                        workspace,100,m,n,NULL);
        }
        else
            break;
    }
}
```