nauty and **Traces** User's Guide (Version 2.5)

Brendan D. McKay*
Research School of Computer Science
Australian National University
Canberra ACT 0200, Australia
bdm@cs.anu.edu.au

Adolfo Piperno
Departimento di Informatica
Sapienza Università di Roma
Rome, Italy
piperno@di.uniroma1.it

January 17, 2013

Contents

*Research supported by the Australian Research Council.

0 How to use this Guide

nauty (**no aut**omorphisms, **y**es?) is a set of procedures for determining the automorphism group of a vertex-coloured graph, and for testing graphs for isomorphism. **Traces** is an alternative program for these operations.

The **dreadnaut** program provides sufficient functionality that most simple applications can be managed without the need to write any programs. Section 2 is intended to be a fairly self-contained introduction to that level of use. You should start by reading Section 1 and Section 2.

nauty also comes with a set of utilities suitable for processing files of graphs; these are described in Section 15.

For other serious purposes, you will need to write a program that calls **nauty** or **Traces**. In that case 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 sample programs in this guide for a constructive proof.

The current versions of **nauty** and **Traces** are available at
`http://cs.anu.edu.au/~bdm/nauty` and `http://pallini.di.uniroma1.it`. There is a mailing list you can subscribe to if you want to discuss **nauty** and **Traces** and receive upgrade notices:
`http://dcsmail.anu.edu.au/cgi-bin/mailman/listinfo/nauty-list`.

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

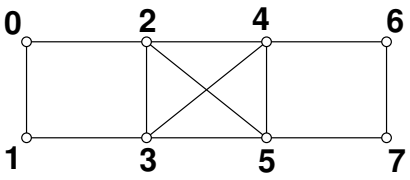
The theoretical basis of the original edition of **nauty** first appeared in [9]. An updated account, and a detailed description of **Traces** appears in [10].

1 Introduction

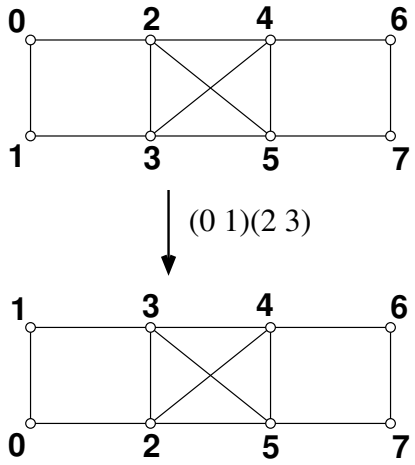
nauty and **Traces** come with a primitive interactive interface **dreadnaut** which will suffice for most one-off computations. This chapter describes the basic concepts and gives examples of **dreadnaut** usage. Later chapters will describe the programming interface.

A *graph* for our purposes has a finite set of *vertices*, and a finite set of *edges*. Most of the time when we write "graph" we mean "simple undirected graph", which implies that each edge is an unordered pair vw of distinct vertices (so multiple edges and loops are not included).

The following shows a graph with 8 vertices and 12 edges.



An *automorphism* of a graph is a permutation of the vertex labels so that the set of edges remains the same. In the above graph we can interchange vertex labels 0,1 and interchange vertex labels 2,3, and this preserves the edge set (for example, 2 is adjacent to 5 before and after, while 0 is not adjacent to 4 before or after). This means that $(0\,1)(2\,3)$ is an automorphism.



The application of two automorphisms one after the other is an automorphism too. The set of all automorphisms, including the trivial one (that moves no labels at all), is called the *automorphism group* of the graph. The automorphism group of the graph above has 8 automorphisms:

(1)	$(0\,6)(1\,7)(2\,4)(3\,5)$
$(0\,1)(2\,3)$	$(0\,7)(2\,5)(1\,6)(3\,4)$
$(4\,5)(6\,7)$	$(0\,6\,1\,7)(2\,4\,3\,5)$
$(0\,1)(2\,3)(4\,5)(6\,7)$	$(0\,7\,1\,6)(2\,5\,3\,4)$

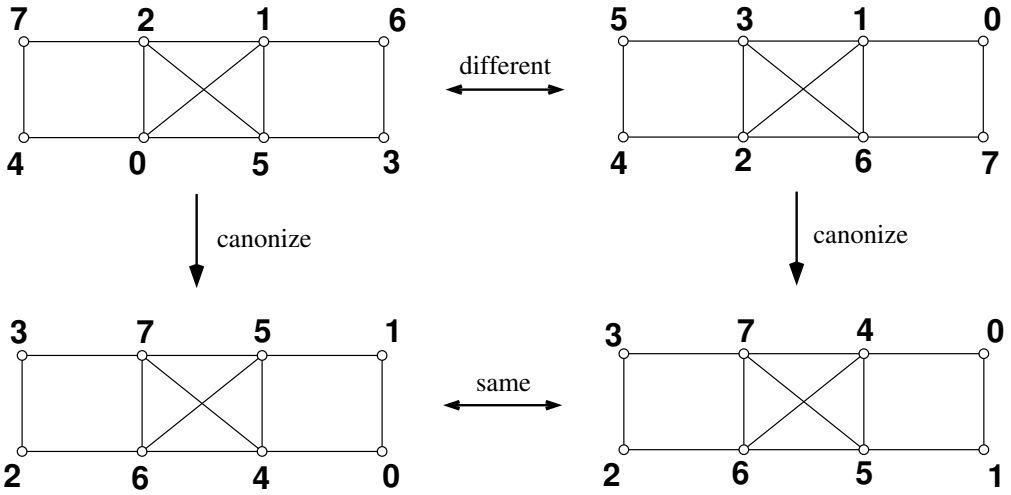
Because the number of automorphisms can be extremely large, it is more efficient to work with a set of *generators* of the automorphism group. This is a set of automorphisms such that every automorphism can be expressed as a combination of them. In the example, a set of generators is $\{(4\,5)(6\,7), (0\,6)(1\,7)(2\,4)(3\,5)\}$.

The automorphisms also define an equivalence relationship on the vertices of the graph: two vertices are equivalent if there is an automorphism taking one to the other. For example, vertices 6 and 7 are equivalent since the automorphism $(4\,5)(6\,7)$ takes 6 onto 7. The sets of equivalent vertices are called *orbits*; in the example they are $\{0, 1, 6, 7\}$ and $\{2, 3, 4, 5\}$.

Another function that **nauty** and **Traces** can perform is *canonical labelling*. This is an operation of placing the vertex labels in a way that does not depend on where they were before. Graphs that are *isomorphic* (the same except for vertex labels) become *identical* (exactly the same) after canonical labelling (canonizing).

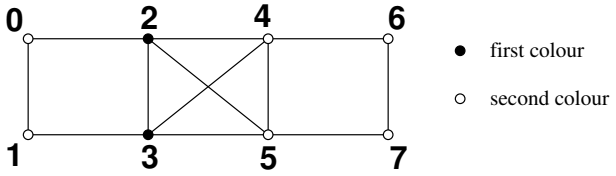
In the figure below, the two graphs in the upper row are clearly isomorphic, though they are not identical (for example 0 and 4 are adjacent in the left graph but not in the

right graph). However, when the graphs are canonized, producing the graphs in the lower row, the results are identical (note that the edges of the two graphs are the same, even though the drawings differ).



The purpose of canonical labelling is to test isomorphism: two isomorphic graphs become identical when they are canonically labelled.

Sometimes the vertices of a graph are distinguished from each other according to some criterion coming from the application. To handle this situation, vertices in **nauty** and **Traces** can be *coloured*. The definition of "automorphism" respects colours: each vertex can only be mapped onto a vertex of the same colour. The example below has two vertex colours, black (•) first and white (◦) second.

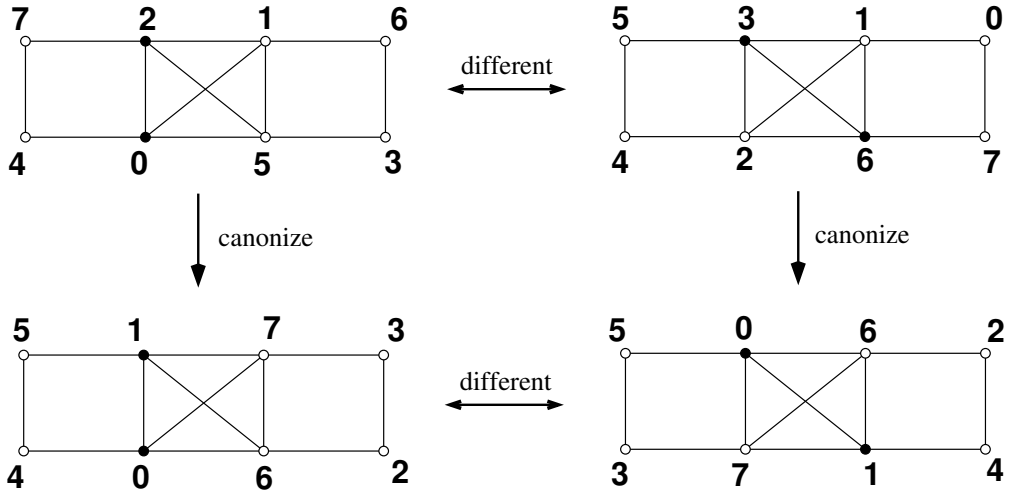


There are now only 4 automorphisms, namely those which preserve the colouring:

(1)	(0 1)(2 3)
(4 5)(6 7)	(0 1)(2 3)(4 5)(6 7)

nauty and **Traces** consider the colours to come in some order; i.e., there is a 1st colour, a 2nd colour, etc.. This doesn't matter with regard to automorphisms, but it plays an important part in canonical labelling: the new vertex labels are in order of colour. The vertices of the first colour are labelled first, of the second colour next, and so on. This rule means that the canonical labelling can be used to determine if two coloured graphs are isomorphic via an isomorphism that maps each vertex of one graph onto a vertex of the same colour in the other graph.

A colouring of the vertices is also referred to as a *partition*, and the colour classes as the *cells* of the partition.



nauty can also handle directed graphs and loops, but **Traces** currently only handles simple undirected graphs.

2 dreadnaut

dreadnaut is a simple program which can read graphs and execute **nauty** or **Traces**. It is a rather primitive interface with few facilities.

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, except when they consist of two characters. Sometimes commands are followed by parameters.

At any point of time, **dreadnaut** knows the following information:

(a) The "mode", which is one of **dense** (for using the dense version of **nauty**; this is the default), **sparse** (for using the sparse version of **nauty**) and **Traces** (for using **Traces**).

(b) The number of vertices, n.

(c) The "current graph" g, if defined.

(d) The "current partition" π. If it is not defined, it is assumed equal to the partition with every vertex in the same cell (i.e., with the same colour).

(e) The orbits of the (coloured) graph (g, π), if defined.

(f) The canonically labelled isomorph of g, called h, if defined. (Also called `canong`.)

(g) An extra graph called h', if defined. (Also called `savedg`.)

(h) Values for each of a variety of options.

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

(A) Commands that set the mode.

`Ad` or `An` Change the mode to **dense**. The packed adjacency matrix data structure will be used for g and the dense version of **nauty** will be used for the `x` command. The graphs g, h, h' and the partition π become undefined.

`As` Change the mode to **sparse**. The adjacency list data structure will be used for g and the sparse version of **nauty** will be used for the `x` command. The graphs g, h, h' and the partition π become undefined.

`At` Change the mode to **Traces**. The adjacency list data structure will be used for g and **Traces** will be used if the `x` command. The graphs g, h, h' and the partition π become undefined.

`Ad+,As+,At+` As well as changing the mode, if g is defined it is converted to the data structure required by the new mode. The graphs h and h' become undefined, but the partition π and the orbits of g are maintained if they are defined.

The initial mode is **dense**.

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

`n=#` Set value of n. The maximum value depends on available memory.

`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 (**dense** mode only).
`.` : 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. This is only available in **dense** mode.

`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 [8]. 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.

`sr=#` Generate random regular graph g of degree i, where i is the integer specified. This is only available in **sparse** and **traces** mode, and i cannot be more than 8.

`_` (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` (see `l` command) 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. Loops count as 1.

(C) Commands which define the partition π.

`f` Specify a 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 $[3,4,5,6,7 \mid 0,2 \mid 1,8,9]$.

`F=#` Make the partition π finer by placing the specified vertex in a cell of its own just before the remains of the cell it was in before.

`FF` Identify the cell which would be first individualized by the chosen algorithm (according to the mode), and place one vertex of that cell in a cell of its own just before the remains of the cell it was in before. This is not likely to be useful unless the partition has been refined first (see the `i` command. This is currently not correct in **Traces** mode.

`i` Perform a refinement operation, replacing the partition π by its refinement. The

refinement procedure used depends on the mode.

`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, for example by using the `f` command.

`O` If the orbits of the automorphism group are known (as by executing the `x` command), they are converted into a partition π. The cells of π are the orbits, and they are arranged in order of their least elements.

`OO` This is like `O`, except that the orbits are placed in increasing order of size and equal-sized orbits are combined into single cells. This means the resulting partition is an isomorphism invariant.

(D) 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.

`$$` 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). A value of 0 indicates no limit.

`w=#` Set value of `worksize` : the amount of space provided for **nauty** to store automorphism data. The amount provided is enough to store i automorphisms, where i is the integer provided. **Traces** does not use this option.

`+` 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** or **Traces** 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** or **Traces** 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 8 for their meaning. The default is TRUE. **Traces** does not use this option.

`p,-p` Set option `cartesian` to TRUE or FALSE, respectively. This tells **nauty** or **Traces** 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 7 for a more detailed description. The default is 100, but setting it to 0 might speed up the average time for easy graphs. **Traces** does not use this option.

`G=#` Set a parameter that effects a probabilistic method (random Schreier algorithm). `G=0` turns off the method altogether (acceptable for **nauty** but disastrous for **Traces**). Larger values make the method more precise but more expensive. The default value is 10, which is adequate for most purposes.

`S=#` Specify a strategy for **Traces**. The default is 0, which is optimal for most cases. However, for some particularly difficult graphs the memory consumption of **Traces** might be excessive. In that event, non-zero values of this parameter will cause **Traces** to sacrifice some time in order to save space. The canonical labelling is not affected.

`*=#` 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 interpreted 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` (all modes)
`9` : `indsets`
`10` : `cliques`
`11` : `cellcliq`
`12` : `cellind`
`13` : `adjacencies` (all modes)
`14` : `cellfano`
`15` : `cellfano2`
`16` : `refinvar`

These procedures are described in Section 10. 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. Except where indicated, the invariants are only available in **dense** mode. See the `k` command for restrictions in **Traces** mode.

`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 7 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. **Traces** does not use invariants itself, but invariants will be applied before calling **Traces** if `mininvarlevel` $\le 1 \le$ `maxinvarlevel`.

`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 10 for the meaning of this option for each available vertex-invariant. The default value depends on the invariant; see the $*$ command.

`V=#` Specify a verbosity level for **Traces**. A value of 0 means that no output will be written except for the summary at the end. A value of 1 produces some output during execution, see Section 8. The default is 0.

`u=#` Request calls to user-defined functions (**nauty** only). The value is
1 for `usernodeproc`,
2 for `userautomproc`,
4 for `userlevelproc`,
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 9).
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 9 for what they mean.
USERREF: Do nothing.

`?` Type the mode, and 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 "$*$".

`P,-P` Turn on, respectively off, the facility to provide known automophisms to **Traces**.

If this is turned on, the `PP` command (below) can be used to input automorphisms, and the group generators (including the extra ones found by **Traces**) are kept when **Traces** exits. The generators will be deleted if the graph changes. To delete them manually, use "`-P P`" (with the space!).

`PP...;` Set on the facility to provide known automophisms to **Traces** (if it isn't on already) and read in one automorphism. Use of this command one or more times before command `x` in **Traces** mode allows known automorphisms to be given to **Traces**. When **Traces** runs, it checks whether the permutation is in fact an automorphism, and dies if it is not. The format of the input is a list of n distinct integers comprising a permutation. Note that the automorphisms you give to **Traces** are not written by it; only extra generators are written.
Example: If $n = 7$ then "`PP 4 0 2 3 6 5 1;`" is valid input.

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

`x` Execute **nauty** or **Traces**. The program to execute depends on the mode. Depending on the values of the `writeautoms` and `writemarkers` options, the automorphism group will be displayed while **nauty** or **Traces** is running. See Section 8 for an explanation of the output. If `getcanon` is TRUE, a canonically labelled graph is computed too. When **nauty** or **Traces** returns, **dreadnaut** will display some statistics about its execution. See Section 7 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 three 8-digit hex numbers whose value depends only on h. This allows quick comparison between graphs. Isomorphic graphs give the same value. Non-isomorphic graphs may also give the same value, though this is rare. 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** or **Traces**, copy h to h');
enter g_2
`x #` (execute **nauty** or **Traces**, 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`. For orbits longer than one

vertex, the orbit size is shown in parentheses.

`M=#,M=#/#,-M` Each call to **nauty** or **Traces** is repeated until either the number of repetitions exceeds the first value or (if the second part is included) the cpu time exceeds the number of seconds in the second value. Each limit can be turned off by setting the value to 0. The cpu time is then reported accurately. This is for doing timing tests with easy graphs. Output is suppressed except for the first execution. This also effects the `i` command. `-M` and `M=1` turn this feature off.

(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, unless the first non-white character is "`"`" in which case the name ends at the next "`"`" or newline. 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 configurable (usually 10).

`>,>>` 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, unless the first non-white character is "`"`" in which case the name ends at the next "`"`" or newline. 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.

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

(G) Command line options

Some **dreadnaut** commands can be given on the command line using the `-o` switch; for example "`dreadnaut -o "At c -a G=20"`". Only one such string of commands is allowed. The available commands are:
`Ad, As, At, a, c, d, G, l, m, M, p, P, V, w, y, $`,
and negatives of those such as `-c`.

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) the mode

(b) option `digraph` (`d` command)

(c) all the vertex-invariant options (∗, `k` and `K` commands)

(d) the value of `tc_level` (`y` command)

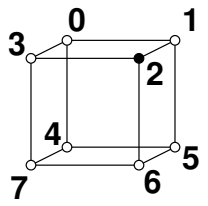
(e) the version of **nauty** or **Traces** used

Assuming you don't have a particularly ancient or broken compiler, the canonical labelling does not depend on the compiler, the operating system, the hardware, or the word size.

2.1 Sample dreadnaut sessions

Several sample **dreadnaut** sessions are shown below. The underlined characters are those typed by the user.

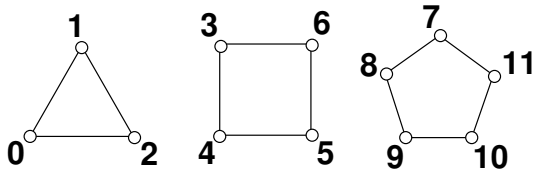
g =



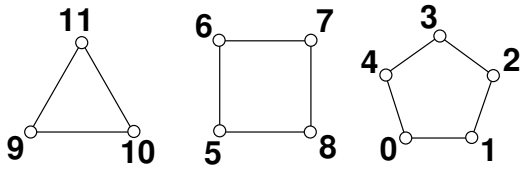

```
> n=8 g                          8 vertices
0: 1 3 4;                        enter the graph
1: 2 5;
2: 3 6;
3: 7;
4: 5 7;
5: 6;
6: 7.
> f=2 x                          fix vertex 2; execute
[fixing partition]
(0 5)(3 6)
level 2:  6 orbits; 3 fixed; index 2
(1 3)(5 7)
level 1:  4 orbits; 1 fixed;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

The next problem solved is to determine an isomorphism between the following two graphs. We turn off the writing of automorphisms to save some space, and this time we will use **Traces**.

g_1 =



g_2 =




```
> At                    use Traces mode
> c -a V=0              turn getcanon on, group writing and verbosity 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; 4 gens; 41 nodes (1 interrupted); maxlev=6;
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; 5 gens; 35 nodes (2 interrupted); maxlev=6;
canupdates=1; 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
> q                          quit
```

As a third example, we consider a simple block design. **nauty** and **Traces** 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 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 $\{x_i B_j \,|\, x_i \in B_j\}$. The following theorem is elementary.

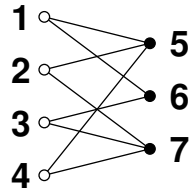
Theorem 1.

(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. This time we will do it with the sparse version of **nauty**.

g =




```
> $=1 As                  label vertices starting at 1, sparse mode
> n=7 g
1: 5:                     go to vertex 5 (block 1), the character is a colon
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
canupdates=1; cpu time = 0.00 seconds
> o                       display the orbits
 1 3 (2); 2 4 (2); 5 7 (2); 6;
> b                       display the canonical labelling
 2 4 1 3 6 7 5                     the vertices in canonical order
  1 :  6 7;                        the relabelled graph
  2 :  6 7;
  3 :  5 7;
  4 :  5 6;
  5 :  3 4;
  6 :  1 2 4;
  7 :  1 2 3;
> q                       quit
```

Looking at the vertices 5,6,7 which represent the blocks, we see that the canonically labelled block design is $\{\,\{3,4\},\{1,2,4\},\{1,2,3\}\,\}$.

3 Data Structures

In this section we will describe the basic data structures required for programs that call **nauty** or **Traces**.

Data structure for graphs.

There are two graph data structures supported. One is the *packed form* used only by the dense version of **nauty**. The other is the *sparse form* used by the sparse version of **nauty** and by **Traces**. The vertices of a graph are numbered $0, 1, \ldots, n-1$.

The *packed form* of a graph is an adjacency matrix with one bit per entry. 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 32 unless the size of type `long int` is greater than 32, in which case WORDSIZE is 64, but this test can be overridden at configuration time, see Section 16.)

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 represented in packed form uses the type `graph`. It is stored as 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 in dense form is really represented by a 1-dimensional array of length mn, not by an array of arrays.

A graph represented in *sparse form* uses the type `sparsegraph`. It is stored as a structure with the following fields:

`int nv`: the number of vertices

`size_t nde`: the number of directed edges (loops count as 1)

`size_t *v`: pointer to an array of length at least `nv`

`int *d`: pointer to an array of length at least `nv`

`int *e`: pointer to an array of length at least `nde`

`SG_WEIGHT *w`: not used in this version, should be NULL

`size_t vlen`, `dlen`, `elen`, `wlen`: the actual lengths of the arrays `v`, `d`, `e` and `w`. The unit is the element type of the array in each case (so `vlen` is the number of entries of type `size_t` in the array `v`, etc.)

For each vertex $i = 0 \ldots n-1$, `d[i]` is the degree (out-degree for a digraph) of that vertex. `v[i]` is an index into the array `e` such that `e[v[i]],e[v[i]+1],...,e[v[i]+d[i]-1]` are the vertices to which vertex `i` is joined. It is not necessary that this list of neighbours be sorted. These neighbour lists can be present in the array `e` in any order and may have gaps between them, but cannot overlap. If `d[i]`$=0$ for some `i`, `v[i]` is not used.

In Figure 1, the graph on the left is represented in packed form by the array in the

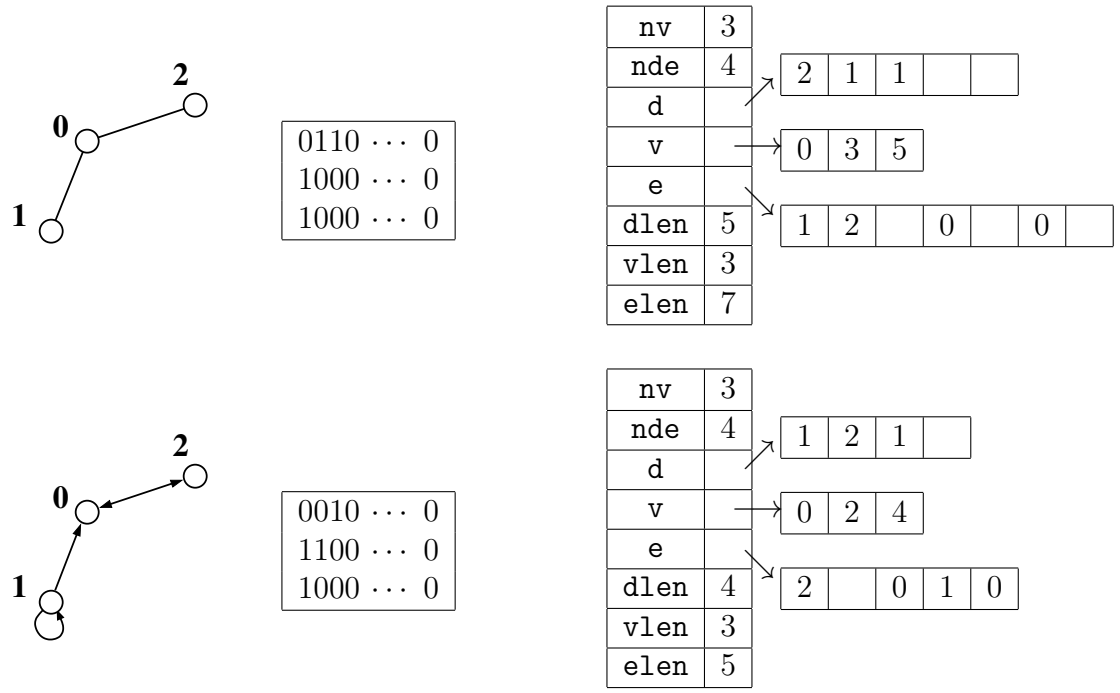


Figure 1: Packed and sparse data structures for graphs and digraphs.

centre (we show three words of type `setword`). On the right is a possible sparse form for the same graph. Note that loops are only allowed for directed graphs, and contribute 1 to the vertex degree.

Before the `sparesgraph` structure can be used, it needs to be initialised. Fields `d,v,e,w` should be set to NULL, and `dlen,vlen,elen,wlen` should be set to 0. After initialisation, the sizes of the fields will be automatically adjusted as required, so you don't need to initialise it again.

Data structure for permutations, orbits, and colourings.

A permutation of V is represented by an array of n integers, type `int`, with the i-th entry giving the image of i under the permutation.

The orbits of the automorphism group is also represented by an array of n integers, type `int`. The value of the i-th entry is the smallest number of a vertex in the same orbit as vertex i.

Examples of a permutation and a set of orbits for $n = 9$ are given below.

2	3	5	6	1	0	4	7	8

is the permutation $(0\,2\,5)(1\,3\,6\,4)$

0	0	2	2	0	0	6	2	6

gives the orbits $\{0,1,4,5\},\{2,3,7\},\{6,8\}$

A colouring (partition) of the vertices is specified by a pair of arrays, usually called `lab` and `ptn`. The array `lab` contains a list of the vertices in some order. The array `ptn` indicates the division into colours: if `ptn[i]` $= 0$, then a cell (colour class) ends at position `i`. The following example for $n = 9$ shows one way to represent the partition

$[\{2\},\{3\},\{0,1,5,6\},\{4,7,8\}]$.

lab:	2	3	5	6	1	0	4	7	8	all vertices in some order
ptn:	0	0	1	1	1	0	1	1	0	cells end where the zeros are

Note that colours come in a particular order. In the example, there are 4 colours listed left to right. However, each colour class by itself is an unordered set of vertices so it makes no difference which order they are listed in. Also, values in `ptn` which are not 0 can be any positive value. (Advanced hint: this is not true internally to **nauty**, probably you don't need to know this.) So, for example, exactly the same partition is represented by the following.

lab:	2	3	1	6	5	0	7	8	4	all vertices in some order
ptn:	0	0	1	2	2	0	8	3	0	cells end where the zeros are

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).

4 Size limits

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 type `int` has less than 32 bits (rare these days), there is an absolute limit of $2^{15} - 3 = 32765$.

(2) If type `int` has at least 32 bits, there is an absolute limit of $2^{30} = 1073741824$ 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. This is the default.

(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 most, but not all 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 $0 < \text{MAXN} \leq \text{WORDSIZE}$, 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.

Traces is limited to the same number of vertices as **nauty**, including the restriction to MAXN if that is non-zero.

5 Options and statistics

Various options are provided to **nauty** or **Traces** by means of options structures. The type `optionblk` is used for **nauty** and the type `TracesOptions` for **Traces**.

In all cases, it is strongly recommended that the values be first set to their defaults by using one of the provided macros:
DEFAULTOPTIONS_GRAPH : for undirected graphs in dense **nauty**
DEFAULTOPTIONS_DIGRAPH : for digraphs in dense **nauty**
DEFAULTOPTIONS_SPARSEGRAPH : for undirected graphs in sparse **nauty**
DEFAULTOPTIONS_SPARSEDIGRAPH : for digraphs in sparse **nauty**
DEFAULTOPTIONS_TRACES : for undirected graphs in **Traces**
If any of the defaults are not suitable, change them using assignment statements. In this way you will only need to recompile if the option structures change in the future.

We first describe `optionblk`, used by **nauty** (both dense and sparse versions).

`boolean getcanon`: If this is TRUE, the canonically labelled graph is produced as well as the automorphism group. Otherwise, only the automorphism group is determined. Default FALSE.

`boolean digraph`: This must be TRUE if the graph has any directed edges or loops. If no directed edges or loops are present, selecting this option is legal but may degrade the performance slightly and the canonical labelling might be different. Default TRUE for DEFAULTOPTIONS_DIGRAPH and DEFAULTOPTIONS_SPARSEDIGRAPH, otherwise 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 8. Default FALSE.

`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 8. Default FALSE.

`boolean defaultptn`: If this is TRUE, it is assumed that all vertices of the graph have the same colour (so the initial values of the parameters `lab` and `ptn` are ignored). If it is FALSE, the initial colouring of the vertices is determined by `lab` and `ptn` as described 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 78.

`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 null 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 9 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 9 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 9 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 9 has details. No calls will be made if the value is NULL. Default NULL.

`void (*invarproc)()`: This is a pointer to a vertex-invariant procedure. See Section 10 for a discussion of vertex-invariants. No calls will be made if the value is NULL. The default is `adjacencies` for DEFAULTOPTIONS_DIGRAPH, `adjacencies_sg` for DEFAULTOPTIONS_SPARSEDIGRAPH, and NULL otherwise.

`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 option `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 `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 option `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 option `getcanon` = TRUE, the sign is ignored. A value of 0 effectively disables `invarproc`. Default 1.

`int invararg`: This value 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 implement different versions of **nauty**. The defaults depend on which DEFAULTOPTIONS variant

is used.

`boolean schreier`: If this is TRUE, pruning of the search tree will be enhanced by use of the random Schreier algorithm. Default FALSE (but **dreadnaut** sets it to TRUE). The setting of this parameter does not effect the canonical labelling.

We now describe `TracesOptions`, used by **Traces**.

`boolean getcanon`: If this is TRUE, the canonically labelled graph is produced as well as the automorphism group. Otherwise, only the automorphism group is determined. 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 8. Default FALSE.

`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.

`boolean digraph`: Unused, must be FALSE. This release of **Traces** cannot handle digraphs.

`boolean defaultptn`: If this is TRUE, it is assumed that all vertices of the graph have the same colour (so the initial values of the parameters `lab` and `ptn` are ignored). If it is FALSE, the initial colouring of the vertices is determined by `lab` and `ptn` as described above. Default TRUE.

`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). Default 0.

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

`int strategy`: A value of 0 causes **Traces** to employ a breadth-first strategy to expand the search tree. A value greater than 1 causes it to use a depth-first-like strategy. In that case, the actual value of the parameter is used as an upper bound on the number of children of each node to generate before proceeding to processing of the first child. The value of this parameter does not effect the canonical labelling and in general the cpu time will increase at the benefit of reduced memory consumption. Default 0.

`int verbosity`: A level of verbosity of messages while **Traces** is running. A value of 0 means that no output will be written (except that automorphisms are written if the `writeautoms` option requests them). Larger values produce greater information about the execution, though its interpretation requires some knowledge of the algorithm. Default 0.

`permnode **generators`: This can be used to provide known automorphisms to **Traces** and receive the automorphisms from **Traces** when it is finished. If it is NULL when **Traces** is called, **Traces** does not change it. If it is non-NULL, it is expected to point to a (perhaps empty) circular list of known automorphisms. (It is an error to give a permutation that is not an automorphism of the input coloured graph.) In this case, **Traces** will add automorphisms to the list so that the whole automorphism group is generated. See Section 18 for detailed instructions and examples. Default NULL.

`void (*userautomproc) (int,int*,int)`: This is a pointer to a user-defined procedure which is to be called for each generator. Section 9 has details. No calls will be made if the value is NULL. Default NULL.

Structured types for receiving statistics

nauty and **Traces** provide some statistics on output. These values do not play a part in the computation, so it isn't an error if some of the counts exceed the capacity of the fields they are stored in.

The various fields of a structure of type `statsblk` are set by **nauty**. Their meanings are as follows:

`double grpsize1`, `int grpsize2`: Within rounding error, the order of the automorphism group is equal to $\texttt{grpsize1} \times 10^{\texttt{grpsize2}}$. If the exact size of a very large group is needed, it can be calculated from the output selected by the `writemarkers` option, or you can compute it with your own multiprecision arithmetic using the `userlevelproc` feature. See Section 8.

`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. The maximum is 1073741826/WORDSIZE+1 if MAXN=0 and `int` has at least 32 bits, 32765/WORDSIZE+1 if MAXN=0 and `int` has at least 32 bits, and $\lceil$MAXN/WORDSIZE$\rceil$ otherwise.
- NTOOBIG: n is too big. Either $n >$ WORDSIZE $\times m$ or n exceeds its absolute limit as in Section 4.
- 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.

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

`unsigned 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.

`unsigned 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.

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

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

`unsigned 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.

Traces returns some statistics in an argument of type `TracesStats`.

`double grpsize1`, `int grpsize2`: Within rounding error, the order of the automorphism group is estimated to be $\texttt{grpsize1} \times 10^{\texttt{grpsize2}}$.

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

`int treedepth`: The depth of the search tree.

`int numgenerators`: The number of generators found for the automorphism group.

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

`unsigned long interrupted`: The number of refinement operations aborted early.

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

`unsigned long peaknodes`: The maximum number of tree nodes simultaneously existing at any moment during the execution.

6 Calling nauty and Traces

In this section, we describe simplified interfaces to **nauty** and the main interface to **Traces**. The hairy details of calling **nauty** directly will be left to Section 7.

A call to the dense version of **nauty** can be made as follows.

`densenauty`(`g`, `lab`, `ptn`, `orbits`, `options`, `stats`, `m`, `n`, `canong`)

`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 above for the format). 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.

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

`optionblk *options`: A structure giving a list of options to the procedure. See above for their meanings. It should be declared using DEFAULTOPTIONS_GRAPH or DEFAULTOPTIONS_DIGRAPH, but options other than `dispatch` can be changed. Read-only.

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

`int m, n`: The number of `setword`s in `set`s and the number of vertices, respectively. It must be the case that $1 \le n \le m \times$ WORDSIZE. If **nauty** is compiled with MAXN > 0, it must also be the case that $n \le$ MAXN and $m \le$ 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.

A call to the sparse version of **nauty** can be made as follows.

`sparsenauty`(`g`, `lab`, `ptn`, `orbits`, `options`, `stats`, `canong`)

The parameters are the same as for `densenauty` except:

(a) Parameters `g` and `canong` have type `sparsegraph*`. If `options.getcanon` = TRUE, then `canong` should have been initialised (see Section 12). The fields will be automatically expanded if they aren't large enough.

(b) `options` should be declared using either DEFAULTOPTIONS_SPARSEGRAPH or DEFAULTOPTIONS_SPARSEDIGRAPH, but options other than `dispatch` can be changed.

A call to **Traces** looks like this:

`Traces`(`g`, `lab`, `ptn`, `orbits`, `toptions`, `tstats`, `canong`)

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

`int *lab,*ptn`: Two arrays of n entries. Their use depends on the values of several options. If `toptions.defaultptn` = TRUE, the input values are ignored (every vertex has the same colour); otherwise, they define the initial colouring of the graph (see above for the format). If `toptions.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`. Read-write.

`int *orbits`: Returns the orbits of the automorphism group, as described above. Write-only.

`TracesOptions *toptions`: A structure giving a list of options to the procedure. See above for their meanings. Read-only.

`TracesStats *tstats`: A structure used by **Traces** to provide some statistics about

what it did. See above for their meanings. Write-only.

`sparsegraph *canong`: The canonically labelled graph, if `toptions.getcanon` = TRUE. Otherwise it can be NULL. Write-only.

7 Low level nauty calls

For most applications, the simplified interface to **nauty** described in the previous section is recommended. This section describes the low-level interface used both for dense graphs and sparse graphs. **nauty** knows which type you are using from the `dispatch` field of the `options` argument.

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` or `sparsegraph *g`: The input graph. Read-only. The actual parameter has type `graph*`, so your compiler might like you to cast a `sparsegraph*` argument to that type if you are using a sparse graph.

`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 $0 \le i \le n-1$. Write-only.

`optionblk *options`: A structure giving a list of options to the procedure. See Section 5 for their meanings. Read-only.

`statsblk *stats`: A structure used by **nauty** to provide some statistics about what it did. See Section 5 for their meanings. Write-only.

`setword *workspace`, `worksize`: The address and length of an array used by **nauty** for working storage. The length is given in units of `setword` (differently from the `w` command in **dreadnaut**). There is no minimum requirement for correct operation, but the efficiency may suffer if not much is provided. A value of `worksize` $\ge 50m$ 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 $1 \le n \le m \times \text{WORDSIZE}$. If **nauty** is compiled with MAXN > 0, it must also be the case that $n \le$ MAXN and $m \le$ MAXM, where MAXM $= \lceil$MAXN/WORDSIZE$\rceil$. Read-only.

`graph` or `sparsegraph` `*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 type must be the same as that of parameter g.

The `C` type of the parameters `g` and `canong` is `graph*`. If another type of pointer is passed (for example `sparsegraph*`), it should be cast to type `graph*`.

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 $\{\texttt{lab}[i], \texttt{lab}[i{+}1], \ldots, \texttt{lab}[j]\}$ where $[i, j]$ is a maximal subinterval of $[0, n{-}1]$ such that $\texttt{ptn}[k] > 0$ for $i \le k < j$ and $\texttt{ptn}[j] = 0$. (In the terminology defined in Section 9, this is the "partition at level 0".) The order of the vertices within each cell has no effect on the behaviour of **nauty**.

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 [9], 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` $\ne$ 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 [9] 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.

Some of the fields in the `options` argument may change the canonical labelling produced by **nauty**. These are fields `digraph`, `defaultptn`, `tc_level`, `userrefproc`, `invarproc`, `mininvarlevel`, `maxinvarlevel`, `invararg` and `dispatch`. The canonical labelling also depends on whether the graph is in packed or sparse form. 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.

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 always internally numbered $0, 1, \ldots, n{-}1$. By default, $k = 0$. Only non-negative values are supported.

8 Interpretation of the output

8.1 nauty output

If `options.writeautoms` = TRUE or `options.writemarkers` = TRUE, **nauty** writes 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:

$\gamma_1^{(k)}$
$\gamma_2^{(k)}$
$\vdots$
$\gamma_{t_k}^{(k)}$
level k: c_k cells; r_k orbits; v_k fixed; index i_k/j_k
$\gamma_1^{(k-1)}$
$\gamma_2^{(k-1)}$
$\vdots$
$\gamma_{t_{k-1}}^{(k-1)}$
level $k{-}1$: c_{k-1} cells; r_{k-1} orbits; v_{k-1} fixed; index i_{k-1}/j_{k-1}
$\vdots$
level 2: c_2 cells; r_2 orbits; v_2 fixed; index i_2/j_2
$\gamma_1^{(1)}$
$\gamma_2^{(1)}$
$\vdots$
$\gamma_{t_1}^{(1)}$
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.

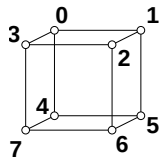
The markers "`level...`" are only written if `options.writemarkers` = TRUE. In the common circumstance that $c_l = r_l$, "c_l `cells;`" is omitted. Similarly, "`/`j_l" is omitted if $j_l = i_l$. Note that $i_l = 1$ is possible for more difficult graphs. Further information about the generators can be found in Theorem 2.34 of [9].

Examples of nauty output

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

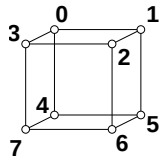
written. Let Γ be the automorphism group. Then

$$\begin{aligned}\Gamma_{0,1,3} &= \{(1)\}\\ \Gamma_{0,1} &= \langle\gamma_1\rangle \text{ with 6 orbits and order 2}\\ \Gamma_0 &= \langle\gamma_1,\gamma_2\rangle \text{ with 4 orbits and order } 2\times 3 = 6\\ \Gamma &= \langle\gamma_1,\gamma_2,\gamma_3\rangle \text{ with 1 orbit and order } 6\times 8 = 48.\end{aligned}$$

The values of `stats.grpsize1` and `stats.grpsize2` show that $|\Gamma| = 48\times 10^0 = 48$. The value of `stats.numorbits` shows there is only one orbit, which is also seen from `orbits`.

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:

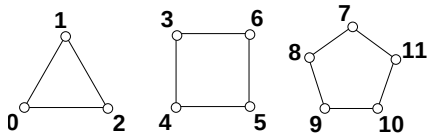
```
5 1 2 6 4 0 3 7
level 2:   6 orbits; 3 fixed; index 2
0 3 2 1 4 7 6 5
level 1:   4 orbits; 1 fixed; index 3
```

`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:

g =



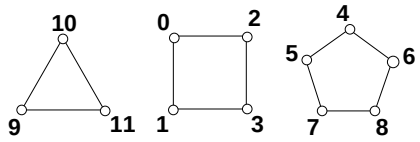
`options`[`getcanon` = TRUE, `digraph` = FALSE, `writeautoms` = TRUE,
`writemarkers` = TRUE, `defaultptn` = TRUE, `tc_level` = 0].
output:

```
(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
```

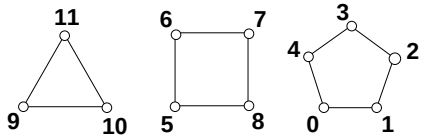
orbits = (0,0,0,3,3,3,3,7,7,7,7,7), stats[grpsize1 = 480.0, grpsize2 = 0, numorbits = 3, numgenerators = 6, numnodes = 40, numbadleaves = 2, maxlevel = 7], lab = (3,4,6,5,7,8,11,9,10,0,1,2).

h =



Example 4:

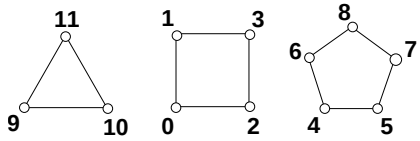
g =



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).

h =

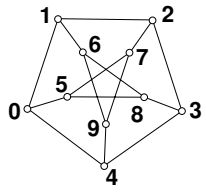


which is identical to the resulting canong in Example 3.

8.2 Traces output

Output from **Traces** is controlled by the writeautoms, cartesian, and verbosity options.

g =



`toptions`[`getcanon` = FALSE, `writeautoms` = TRUE,
`defaultptn` = TRUE, `verbosity` = 1].

output:

```
  Gen #1:  (0 4 3 7 5 1)(2 9 8)
  Gen #2:  (0 2)(3 4)(6 9)(7 8)
  level 1:  3 cells; target cell: 10; 1 orbit; 3 nodes (1 kept); 1 update;
  Gen #3:  (1 4 9)(2 8 7)(3 6 5)
  Gen #4:  (3 6)(4 9)(7 8)
  Gen #5:  (1 9 4)(2 6 8 5 7 3)
  level 2:  7 cells; target cell: 6; 1 orbit; 5 nodes (2 kept); 1 update;
  level 3:  10 cells; target cell: 2; 1 orbit; 3 nodes (1 kept); 1 update;
```

The lines starting `Gen` are generators for the group. Note that in **Traces** there is no base and the generators do not in general form a strong generating set. The lines starting `level`, which are turned on by the `verbosity` option, describe the levels in the search tree:

(a) the level number (level 1 has the children of the root of the tree);

(b) the number of cells, and the size of the target cell, for the nodes on this level that are the best;

(c) the number of orbits in the group generated by the generators found by the time this level is finished;

(d) the number of nodes created on this level, and the number which are judged to be best;

(e) the number of times a node was found on this level better than the previous nodes on this level.

If known automorphisms are given to **Traces** via `toptions.generators`, these are not written.

9 User-defined procedures

nauty makes provision for up to four 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` and/or `usernodeproc`, `userlevelproc` fields of `options` (see Section 7). In all cases, a value of NULL will cause no call to be made.

Traces currently has one such procedure, `userautomproc`.

Proceedures for nauty.

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 7 for their meanings. If the sparse version of **nauty** is being used, the argument passed to parameter `g` actually has type

`sparsegraph*`. Correct practice is declare it as type `graph*` but cast it to type `sparsegraph*` before use.

`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.

`int *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

without affecting **nauty** behaviour. See Section 7.

`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 **nauty** is called with `options.digraph` = TRUE.

The usefulness of `userrefproc` has declined since vertex-invariants were introduced (see Section 10).

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

This is called once for every node of the search 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 8). 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 8. 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.

`int *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 7 for the format of `orbits`.

`int stabvertex`: The value v_j, as defined in Section 8.

`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 8, 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.

`n`,`lab`,`ptn`,`level`: 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 7 for the format. In the notation of Section 8, `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 7, 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 8, 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.

Proceedures for Traces.

`userautomproc`(`count`, `perm`, `n`)

This is called once for each generator of the automorphism group, in the same order as they are written. 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).

The parameters have meanings as below. Treat them all as read-only.

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

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

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

10 Vertex-invariants

The operation of **nauty** and **Traces** 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.

Traces does not have the facility to use invariants during its operation, though **dreadnaut** allows an invariant to be applied before **Traces** is called.

Formally, let $\mathcal{G}$ be the set of all labelled graphs (or digraphs) with vertex set $V = \{0, 1, \ldots, n-1\}$, and let Π 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

$$\phi \ : \ \mathcal{G} \times \Pi \times V \to \mathcal{Z}$$

such that $\phi(G^\gamma, \pi^\gamma, v^\gamma) = \phi(G, \pi, v)$ for every $G \in \mathcal{G}$, $\pi \in \Pi$, $v \in V$ 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 7. 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.

Unless otherwise specified, these invariants are only available for graphs in packed form. Trying to use them with sparse form will cause a disaster.

`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. Use the smallest value of `invararg` that gives satisfactory results.

`distances_sg`. This is like `distances` but works for sparse form rather than packed form. It is in the file `nausparse.c` rather than `nautyinv.c`.

`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.

`adjacencies_sg`. This is like `adjacencies` but works for sparse form rather than packed form. It is in the file `nausparse.c` rather than `nautyinv.c`.

`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. Another class of graph that this invariant can help with is the graphs derived from Latin squares as in Section 14.

`refinvar.` Each vertex is given a code that depends on the result of refining the partition resulting from individualization of that vertex. The refinement is not necessarily complete; its completeness depends on the value of `invararg`. Use the smallest value of `invararg` that gives satisfactory results. This is good for regular graphs that are not strongly regular, and similar graphs.

11 Writing programs which call dense nauty

Programs which call the dense version of **nauty** should include the file `nauty.h` and be linked with `nauty.c`, `nautil.c`, `naugraph.c`, `schreier.c`, and `naurng.c`. If a built-in invariant is used, the file `nautinv.h` should be included too, and `nautinv.c` should be linked.

The simplest way to link with the necessary object files is to use the library `nauty.a`.

Suppose that m and n have meanings as usual.

There are two general approaches to storage management. 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 MAXM 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 DYNALLOC1 or DYNALLOC2 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. Otherwise the same space is reused. 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");`

The last parameter of DYNALLOC1 and DYNALLOC2 is a string used in an error message in the event that the allocation fails.

`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.

SETWORDSNEEDED(`n`) : the least number of `setword`s needed for n bits.

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 $0 \le i <$ WORDSIZE. Thus, *ANDing* a `setword` with BITMASK(`i`) deletes bits $0..i$.

ALLMASK(`i`) : A `setword` constant with the first i bits set and all other bits unset, for $0 \le i \le$ WORDSIZE.

GRAPHROW(`g`,`v`,`m`) : The address of the row of graph `g` corresponding to the neighbours of vertex `v`.

EMPTYGRAPH(`g`,`m`,`n`) : Makes a graph empty (i.e., no edges).

ADDONEARC(`g`,`v`,`w`,`m`) : Add one directed edge to a graph.

ADDONEEDGE(`g`,`v`,`w`,`m`) : Add one undirected edge to a graph.

SETWORD_SHORT, SETWORD_INT, SETWORD_LONG, SETWORD_LONGLONG : Exactly one of these is defined, according to which unsigned integer type is the same as `setword`.

SETWORDSNEEDED(`n`) : Calculates $\lceil n/\text{WORDSIZE} \rceil$.

SETWORD_FORMAT : A string suitable for writing a value of type `setword` with `fprintf`. For example it might be `"%08lx"`.

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 }
```

If you just want to do something for each bit in a `setword x`, it is more efficient to do it like this:

```
tmp = x;
while (tmp)
{
    TAKEBIT(i,tmp);
    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 2). Many of these are also useful to programs which call **nauty**. If your program uses them, include `naututil.h` as well as `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.

`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 files `nautaux.c`, `gutil1.c` and `gutil2.c` contain a few procedures which manipulate graphs or partitions, or compute properties of them, 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. This only needs to be done once.

We next give some programs which illustrate simple use of dense **nauty**. The source files are included in the **nauty** distribution.

11.1 nautyex1.c : Packed form with static allocation

```
/* This program prints generators for the automorphism group of an
   n-vertex polygon, where n is a number supplied by the user.

   This version uses a fixed limit for MAXN.
*/

#define MAXN 1000    /* Define this before including nauty.h */
#include "nauty.h"   /* which includes <stdio.h> and other system files */

int
main(int argc, char *argv[])
{
    graph g[MAXN*MAXM];
    int lab[MAXN],ptn[MAXN],orbits[MAXN];
    static DEFAULTOPTIONS_GRAPH(options);
    statsblk stats;

    int n,m,v;

 /* Default options are set by the DEFAULTOPTIONS_GRAPH macro above.
    Here we change those options that we want to be different from the
    defaults.  writeautoms=TRUE causes automorphisms to be written.     */

    options.writeautoms = TRUE;

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) != 1 || n <= 0)    /* Exit if EOF or bad number */
            break;

        if (n > MAXN)
        {
            printf("n must be in the range 1..%d\n",MAXN);
            exit(1);
        }

     /* The nauty parameter m is a value such that an array of
        m setwords is sufficient to hold n bits.  The type setword
        is defined in nauty.h.  The number of bits in a setword is
        WORDSIZE, which is 16, 32 or 64.  Here we calculate
        m = ceiling(n/WORDSIZE).                                    */

        m = SETWORDSNEEDED(n);
```

```
     /* The following optional call verifies that we are linking
        to compatible versions of the nauty routines.            */

        nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

     /* Now we create the cycle.  First we zero the graph, than for
        each v, we add the edge (v,v+1), where values are mod n. */

        EMPTYGRAPH(g,m,n);
        for (v = 0; v < n; ++v)  ADDONEEDGE(g,v,(v+1)%n,m);

        printf("Generators for Aut(C[%d]):\n",n);

     /* Since we are not requiring a canonical labelling, the last
        parameter to densenauty() is not required and can be NULL. */

        densenauty(g,lab,ptn,orbits,&options,&stats,m,n,NULL);

     /* The size of the group is returned in stats.grpsize1 and
        stats.grpsize2. */

        printf("Automorphism group size = ");
        writegroupsize(stdout,stats.grpsize1,stats.grpsize2);
        printf("\n");
    }

    exit(0);
}
```

11.2 nautyex2.c : Packed form with dynamic allocation

```
/* This program prints generators for the automorphism group of an
   n-vertex polygon, where n is a number supplied by the user.

   This version uses dynamic allocation.
*/

#include "nauty.h"
/* MAXN=0 is defined by nauty.h, which implies dynamic allocation */

int
main(int argc, char *argv[])
{
  /* DYNALLSTAT declares a pointer variable (to hold an array when it
     is allocated) and a size variable to remember how big the array is.
     Nothing is allocated yet.  */

    DYNALLSTAT(graph,g,g_sz);
    DYNALLSTAT(int,lab,lab_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    static DEFAULTOPTIONS_GRAPH(options);
    statsblk stats;

    int n,m,v;
    set *gv;

/* Default options are set by the DEFAULTOPTIONS_GRAPH macro above.
   Here we change those options that we want to be different from the
   defaults.  writeautoms=TRUE causes automorphisms to be written. */

    options.writeautoms = TRUE;

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {

     /* The nauty parameter m is a value such that an array of
        m setwords is sufficient to hold n bits.  The type setword
        is defined in nauty.h.  The number of bits in a setword is
        WORDSIZE, which is 16, 32 or 64.  Here we calculate
        m = ceiling(n/WORDSIZE). */

            m = SETWORDSNEEDED(n);
```

```
        /* The following optional call verifies that we are linking
           to compatible versions of the nauty routines. */

           nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

        /* Now that we know how big the graph will be, we allocate
         * space for the graph and the other arrays we need. */

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

           EMPTYGRAPH(g,m,n);
           for (v = 0; v < n; ++v) ADDONEEDGE(g,v,(v+1)%n,m);

           printf("Generators for Aut(C[%d]):\n",n);
           densenauty(g,lab,ptn,orbits,&options,&stats,m,n,NULL);

           printf("order = ");
           writegroupsize(stdout,stats.grpsize1,stats.grpsize2);
           printf("\n");
       }
       else
           break;
   }

   exit(0);
}
```

11.3 nautyex8.c : Determining an isomorphism, dense form

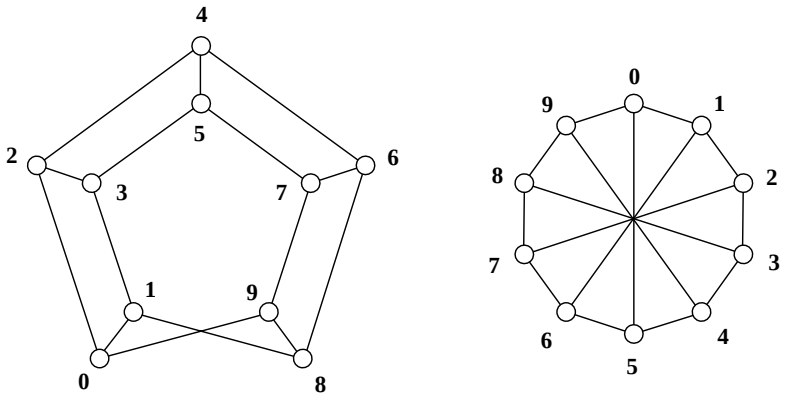


Figure 2: Two labellings of a Moebius graph.

```
/* This program demonstrates how an isomorphism is found between
   two graphs, using the Moebius graphs as an example.
   This version uses dense form with dynamic allocation.
*/

#include "nauty.h"

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,lab1,lab1_sz);
    DYNALLSTAT(int,lab2,lab2_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    DYNALLSTAT(int,map,map_sz);
    DYNALLSTAT(graph,g1,g1_sz);
    DYNALLSTAT(graph,g2,g2_sz);
    DYNALLSTAT(graph,cg1,cg1_sz);
    DYNALLSTAT(graph,cg2,cg2_sz);
    static DEFAULTOPTIONS_GRAPH(options);
    statsblk stats;

    int n,m,i;
    size_t k;

 /* Select option for canonical labelling */

    options.getcanon = TRUE;

    while (1)
    {
```

```
printf("\nenter n : ");
if (scanf("%d",&n) == 1 && n > 0)
{
    if (n%2 != 0)
    {
        fprintf(stderr,"Sorry, n must be even\n");
        continue;
    }

    m = SETWORDSNEEDED(n);
    nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

    DYNALLOC1(int,lab1,lab1_sz,n,"malloc");
    DYNALLOC1(int,lab2,lab2_sz,n,"malloc");
    DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
    DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
    DYNALLOC1(int,map,map_sz,n,"malloc");

 /* Now make the first graph */

    DYNALLOC2(graph,g1,g1_sz,n,m,"malloc");
    EMPTYGRAPH(g1,m,n);

    for (i = 0; i < n; i += 2)   /* Spokes */
        ADDONEEDGE(g1,i,i+1,m);

    for (i = 0; i < n-2; ++i)  /* Cycle */
        ADDONEEDGE(g1,i,i+2,m);
    ADDONEEDGE(g1,1,n-2,m);
    ADDONEEDGE(g1,0,n-1,m);

 /* Now make the second graph */

    DYNALLOC2(graph,g2,g2_sz,n,m,"malloc");
    EMPTYGRAPH(g2,m,n);

    for (i = 0; i < n; ++i)
    {
        ADDONEEDGE(g2,i,(i+1)%n,m);   /* Rim */
        ADDONEEDGE(g2,i,(i+n/2)%n,m); /* Diagonals */
    }

 /* Create canonical graphs */

    DYNALLOC2(graph,cg1,cg1_sz,n,m,"malloc");
    DYNALLOC2(graph,cg2,cg2_sz,n,m,"malloc");

    densenauty(g1,lab1,ptn,orbits,&options,&stats,m,n,cg1);
```

```
            densenauty(g2,lab2,ptn,orbits,&options,&stats,m,n,cg2);

        /* Compare canonically labelled graphs */

            for (k = 0; k < m*(size_t)n; ++k)
                if (cg1[k] != cg2[k]) break;

            if (k == m*(size_t)n)
            {
                printf("Isomorphic.\n");
                if (n <= 1000)
                {
                 /* Write the isomorphism.  For each i, vertex lab1[i]
                    of sg1 maps onto vertex lab2[i] of sg2.  We compute
                    the map in order of labelling because it looks better. */

                    for (i = 0; i < n; ++i) map[lab1[i]] = lab2[i];
                    for (i = 0; i < n; ++i) printf(" %d-%d",i,map[i]);
                    printf("\n");
                }
            }
            else
                printf("Not isomorphic.\n");
        }
        else
            break;
    }

    exit(0);
}
```

12 Writing programs which call sparse nauty

The basic data structure for sparse representation is the structure `sparsegraph` defined in Section 3. Programs using it should include `nausparse.h` and link with the file `nausparse.c`.

As described in Section 3, the sparse representation of a graph uses a structure of type `sparsegraph` with the following fields:

`int nv`: the number of vertices

`int nde`: the number of directed edges (loops count as 1)

`int *v`: pointer to an array of length at least `nv`

`int *d`: pointer to an array of length at least `nv`

`int *e`: pointer to an array of length at least `nde`

`SG_WEIGHT *w`: not implemented in this version, should be NULL

`size_t vlen`, `dlen`, `elen`, `wlen`: the actual lengths of the arrays v, d, e and w. The unit is the element type of the array in each case (so `vlen` is the number of `int`s in the array v, etc.)

For definiteness we will assume that such a graph is declared thus:

```
sparsegraph sg;
```

Before use this should be initialised, for which there is a macro:

```
SG_INIT(sg);
```

or alternatively you can declare and initialise it at once:

```
SG_DECL(sg);
```

To allocate the `v`, `d` and `e` arrays for a graph with n vertices and e directed edge, use

```
SG_ALLOC(sg,n,e,"message");
```

where the message is used if allocation fails, and to free this space use

```
SG_FREE(sg);
```

A particular graph can be stored in several different ways, since the lists of neighbours of vertex do not need to be contiguous in `sg.e`, nor do they need to be sorted. To tell if two sparse graphs are identical, there is a procedure `aresame_sg` in `nausparse.c`.

The canonically labelled graph produced by **nauty** is guaranteed to already have sorted contiguous adjacency lists. It also has a specific value of `sg.v`[i] if vertex i has degree 0, namely 0 for $i = 0$ and `sg.v`[$i-1$]+1 otherwise.

Some utilities for handling sparse form graphs can be found in `nausparse.c`:

`aresame_sg` : Test if two sparse graphs are the same. (Note: this is not an isomorphism test, just a labelled graph comparison.)

`sortlists_sg` : Sort the neighbourhood lists `sg.e`[`sg.v`[i] .. `sg.v`[i]+`sg.v`[i]-1] into ascending order.

`put_sg` : Write a sparse graph in human-readable format.

`copy_sg` : Make a copy of a sparse graph.

`sg_to_nauty` : Convert sparse form to packed form.

`nauty_to_sg` : Convert packed form to sparse form.

Now we give versions of the previous two programs that use sparse **nauty** instead of dense **nauty**.

12.1 nautyex4.c : Sparse form with dynamic allocation

```
/* This program prints generators for the automorphism group of an
   n-vertex polygon, where n is a number supplied by the user.
   This version uses sparse form with dynamic allocation.
*/

#include "nausparse.h"    /* which includes nauty.h */

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,lab,lab_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    static DEFAULTOPTIONS_SPARSEGRAPH(options);
    statsblk stats;
    sparsegraph sg;   /* Declare sparse graph structure */

    int n,m,i;

    options.writeautoms = TRUE;

 /* Initialise sparse graph structure. */

    SG_INIT(sg);

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

            DYNALLOC1(int,lab,lab_sz,n,"malloc");
            DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
            DYNALLOC1(int,orbits,orbits_sz,n,"malloc");

         /* SG_ALLOC makes sure that the v,d,e fields of a sparse graph
            structure point to arrays that are large enough.  This only
            works if the structure has been initialised. */

            SG_ALLOC(sg,n,2*n,"malloc");

            sg.nv = n;              /* Number of vertices */
            sg.nde = 2*n;           /* Number of directed edges */
```

```
            for (i = 0; i < n; ++i)
            {
                sg.v[i] = 2*i;
                sg.d[i] = 2;
                sg.e[2*i] = (i+n-1)%n;      /* edge i->i-1 */
                sg.e[2*i+1] = (i+n+1)%n;    /* edge i->i+1 */
            }

            printf("Generators for Aut(C[%d]):\n",n);
            sparsenauty(&sg,lab,ptn,orbits,&options,&stats,NULL);

            printf("Automorphism group size = ");
            writegroupsize(stdout,stats.grpsize1,stats.grpsize2);
            printf("\n");
        }
        else
            break;
    }

    exit(0);
}
```

12.2 nautyex5.c : Sparse form with dynamic allocation

```
/* This program demonstrates how an isomorphism is found between
   two graphs, using the Moebius graph as an example.
   This version uses sparse form with dynamic allocation.
*/

#include "nausparse.h"    /* which includes nauty.h */

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,lab1,lab1_sz);
    DYNALLSTAT(int,lab2,lab2_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    DYNALLSTAT(int,map,map_sz);
    static DEFAULTOPTIONS_SPARSEGRAPH(options);
    statsblk stats;
 /* Declare and initialize sparse graph structures */
    SG_DECL(sg1); SG_DECL(sg2);
    SG_DECL(cg1); SG_DECL(cg2);

    int n,m,i;

 /* Select option for canonical labelling */

    options.getcanon = TRUE;

 /* Read the number of vertices and process it */

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
            if (n%2 != 0)
            {
                fprintf(stderr,"Sorry, n must be even\n");
                continue;
            }

            m = SETWORDSNEEDED(n);
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

            DYNALLOC1(int,lab1,lab1_sz,n,"malloc");
            DYNALLOC1(int,lab2,lab2_sz,n,"malloc");
            DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
```

```
   DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
   DYNALLOC1(int,map,map_sz,n,"malloc");

/* Now make the first graph */

   SG_ALLOC(sg1,n,3*n,"malloc");
   sg1.nv = n;              /* Number of vertices */
   sg1.nde = 3*n;           /* Number of directed edges */

   for (i = 0; i < n; ++i)
   {
       sg1.v[i] = 3*i;     /* Position of vertex i in v array */
       sg1.d[i] = 3;       /* Degree of vertex i */
   }

   for (i = 0; i < n; i += 2)   /* Spokes */
   {
       sg1.e[sg1.v[i]] = i+1;
       sg1.e[sg1.v[i+1]] = i;
   }

   for (i = 0; i < n-2; ++i)  /* Clockwise edges */
       sg1.e[sg1.v[i]+1] = i+2;
   sg1.e[sg1.v[n-2]+1] = 1;
   sg1.e[sg1.v[n-1]+1] = 0;

   for (i = 2; i < n; ++i)  /* Anticlockwise edges */
       sg1.e[sg1.v[i]+2] = i-2;
   sg1.e[sg1.v[1]+2] = n-2;
   sg1.e[sg1.v[0]+2] = n-1;

/* Now make the second graph */

   SG_ALLOC(sg2,n,3*n,"malloc");
   sg2.nv = n;              /* Number of vertices */
   sg2.nde = 3*n;           /* Number of directed edges */

   for (i = 0; i < n; ++i)
   {
       sg2.v[i] = 3*i;
       sg2.d[i] = 3;
   }

   for (i = 0; i < n; ++i)
   {
       sg2.v[i] = 3*i;
       sg2.d[i] = 3;
       sg2.e[sg2.v[i]] = (i+1) % n;      /* Clockwise */
```

```
            sg2.e[sg2.v[i]+1] = (i+n-1) % n;  /* Anti-clockwise */
            sg2.e[sg2.v[i]+2] = (i+n/2) % n;  /* Diagonals */
        }

    /* Label sg1, result in cg1 and labelling in lab1; similarly sg2.
       It is not necessary to pre-allocate space in cg1 and cg2, but
       they have to be initialised as we did above.  */

       sparsenauty(&sg1,lab1,ptn,orbits,&options,&stats,&cg1);
       sparsenauty(&sg2,lab2,ptn,orbits,&options,&stats,&cg2);

    /* Compare canonically labelled graphs */

       if (aresame_sg(&cg1,&cg2))
       {
           printf("Isomorphic.\n");
           if (n <= 1000)
           {
            /* Write the isomorphism.  For each i, vertex lab1[i]
               of sg1 maps onto vertex lab2[i] of sg2.  We compute
               the map in order of labelling because it looks better. */

               for (i = 0; i < n; ++i) map[lab1[i]] = lab2[i];
               for (i = 0; i < n; ++i) printf(" %d-%d",i,map[i]);
               printf("\n");
           }
       }
       else
           printf("Not isomorphic.\n");
    }
    else
        break;
}

    exit(0);
}
```

13 Writing programs which call Traces

Traces uses the same data structures for graphs, partitions, permutations and orbits as sparse **nauty**, so the functions for manipulating sparse graphs can be used unchanged.

Here we give the previous program again, using **Traces**.

13.1 nautyex7.c : Determining an isomorphism using Traces

```
/* This program demonstrates how an isomorphism is found between
   two graphs, using the Moebius graph as an example.
   This version uses Traces.
*/

#include "traces.h"

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,lab1,lab1_sz);
    DYNALLSTAT(int,lab2,lab2_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    DYNALLSTAT(int,map,map_sz);
    static DEFAULTOPTIONS_TRACES(options);
    TracesStats stats;
 /* Declare and initialize sparse graph structures */
    SG_DECL(sg1); SG_DECL(sg2);
    SG_DECL(cg1); SG_DECL(cg2);

    int n,m,i;

 /* Select option for canonical labelling */

    options.getcanon = TRUE;

 /* Read a number of vertices and process */

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
            if (n%2 != 0)
            {
                fprintf(stderr,"Sorry, n must be even\n");
                continue;
            }

            m = SETWORDSNEEDED(n);
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

            DYNALLOC1(int,lab1,lab1_sz,n,"malloc");
            DYNALLOC1(int,lab2,lab2_sz,n,"malloc");
            DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
```

```
   DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
   DYNALLOC1(int,map,map_sz,n,"malloc");

/* Now make the first graph */

   SG_ALLOC(sg1,n,3*n,"malloc");
   sg1.nv = n;              /* Number of vertices */
   sg1.nde = 3*n;           /* Number of directed edges */

   for (i = 0; i < n; ++i)
   {
       sg1.v[i] = 3*i;     /* Position of vertex i in v array */
       sg1.d[i] = 3;       /* Degree of vertex i */
   }

   for (i = 0; i < n; i += 2)   /* Spokes */
   {
       sg1.e[sg1.v[i]] = i+1;
       sg1.e[sg1.v[i+1]] = i;
   }

   for (i = 0; i < n-2; ++i)  /* Clockwise edges */
       sg1.e[sg1.v[i]+1] = i+2;
   sg1.e[sg1.v[n-2]+1] = 1;
   sg1.e[sg1.v[n-1]+1] = 0;

   for (i = 2; i < n; ++i)  /* Anticlockwise edges */
       sg1.e[sg1.v[i]+2] = i-2;
   sg1.e[sg1.v[1]+2] = n-2;
   sg1.e[sg1.v[0]+2] = n-1;

/* Now make the second graph */

   SG_ALLOC(sg2,n,3*n,"malloc");
   sg2.nv = n;              /* Number of vertices */
   sg2.nde = 3*n;           /* Number of directed edges */

   for (i = 0; i < n; ++i)
   {
       sg2.v[i] = 3*i;
       sg2.d[i] = 3;
   }

   for (i = 0; i < n; ++i)
   {
       sg2.v[i] = 3*i;
       sg2.d[i] = 3;
       sg2.e[sg2.v[i]] = (i+1) % n;      /* Clockwise */
```

```
            sg2.e[sg2.v[i]+1] = (i+n-1) % n;  /* Anti-clockwise */
            sg2.e[sg2.v[i]+2] = (i+n/2) % n;  /* Diagonals */
        }

    /* Label sg1, result in cg1 and labelling in lab1; similarly sg2.
       It is not necessary to pre-allocate space in cg1 and cg2, but
       they have to be initialised as we did above.  */

       Traces(&sg1,lab1,ptn,orbits,&options,&stats,&cg1);
       Traces(&sg2,lab2,ptn,orbits,&options,&stats,&cg2);

    /* Compare canonically labelled graphs */

       if (aresame_sg(&cg1,&cg2))
       {
           printf("Isomorphic.\n");
           if (n <= 1000)
           {
            /* Write the isomorphism.  For each i, vertex lab1[i]
               of sg1 maps onto vertex lab2[i] of sg2.  We compute
               the map in order of labelling because it looks better. */

               for (i = 0; i < n; ++i) map[lab1[i]] = lab2[i];
               for (i = 0; i < n; ++i) printf(" %d-%d",i,map[i]);
               printf("\n");
           }
       }
       else
           printf("Not isomorphic.\n");
    }
    else
        break;
}

exit(0);
}
```

14 Variations

As mentioned, **nauty** and **Traces** can handle graphs with coloured vertices. In this section, we describe how several other types of isomorphism problem can be solved by mapping them onto a problem for vertex-coloured graphs. (But recall that **Traces** can't handle directed edges.)

Isomorphism of edge-coloured graphs. Isomorphism of two graphs, each with both vertices and edges coloured, is defined in the obvious way. An example of such a graph appears at the left of Figure 3.

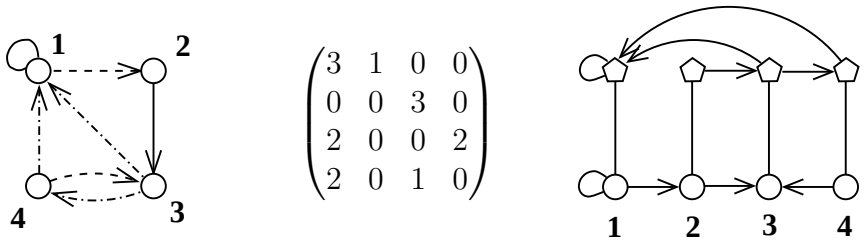


Figure 3: Graphs with coloured edges

In the center of the figure we identify the colours with the integers $1, 2, 3$. At the right of the figure we show an equivalent vertex-coloured graph. In this case there are two layers, each with its own colour. Edges of colour 1 are represented as an edge in the first (lowest) layer, edges of colour 2 are represented as an edge in the second layer, and edges of colour 3 are represented as edges in both layers. It is now easy to see that the automorphism group of the new graph (precisely, its action on the first layer) is the automorphism group of the original graph. Moreover, the order in which a canonical labelling of the new graph labels the vertices of the first layer can be taken to be a canonical labelling of the original graph.

More generally, if the edge colours are integers in $\{1, 2, \ldots, 2^d - 1\}$, we make d layers, and the binary expansion of each colour number tells us which layers contain edges. The vertical threads (each corresponding to one vertex of the original graph) can be connected using either paths or cliques. If the original graph has n vertices and k colours, the new graph has $O(n \log k)$ vertices. This can be improved to $O(n\sqrt{\log k})$ vertices by also using edges that are not horizontal, but this needs care.

Exchangeable vertex colours. The vertex colours known to **nauty** and **Traces** are distinguishable: vertices can only be mapped onto vertices of the same colour. In some applications, entire colour classes can also be exchanged.

In the left side of Figure 4 is a graph with three exchangeable vertex colours. To process this problem with **nauty** or **Traces**, we recolour the vertices to be all the same, then indicate the original colour classes with additional vertices of a new colour, as in the graph on the right. It is easy to see how this idea can be extended to allow some colours to be exchangeable and some not.

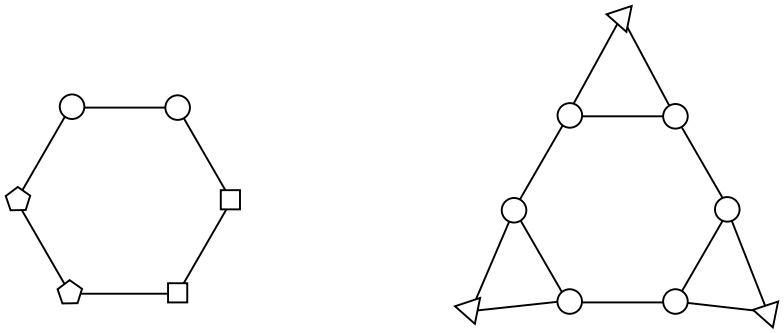


Figure 4: Graphs with exchangeable vertex colours.

Isomorphism of hypergraphs and designs. A *hypergraph* is similar to an undirected graph except that the edges can be vertex sets of any size, not just of size 2. Such an structure is also called a *design*.

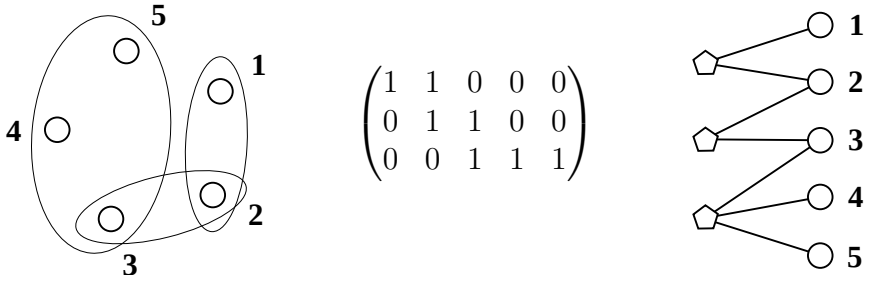


Figure 5: Hypergraph/design isomorphism as graph isomorphism

In the left of Figure 5 we see a hypergraph with 5 vertices, two edges of size 2, and one edge of size 3. On the right is an equivalent vertex-coloured graph. The vertices on the left, coloured with one colour, represent the hypergraph edges, while the edges on the right, coloured with a different colour, represent the hypergraph vertices. The edges of the graph indicate the hypergraph incidence (containment) relationship.

In the center of the figure, we show the edge-vertex incidence matrix. This can be any binary matrix at all, which prompts us to note that the problem under consideration is just that of determining 0-1 matrix equivalence under independent permutation of the rows and columns. By combining this idea with the previous construction, we can handle such an equivalence relation on the set of matrices with arbitrary entries.

Hadamard equivalence. Two matrices over $\{-1, +1\}$ are *Hadamard-equivalent* if one can be obtained from the other by permuting the rows, permuting the columns, and multiplying some of the rows and some of the columns by -1.

Suppose $A = (a_{ij})$ is a matrix over $\{-1, +1\}$ of order $m \times n$. Construct a graph $G(A)$ with vertices $v_1, \ldots, v_m, v'_1, \ldots, v'_m$ of one colour, and $w_1, \ldots, w_n, w'_1, \ldots, w'_n$ of another colour. Insert the edges are $\{v_i, w_j\}$ and $\{v'_i, w'_j\}$ if $a_{ij} = 1$ and $\{v_i, w'_j\}$ and $\{v'_i, w_j\}$ if $a_{ij} = -1$. Figure 6 gives an example.

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

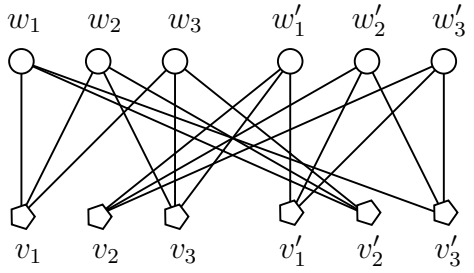


Figure 6: Hadamard equivalence as graph isomorphism

Permuting the rows of A corresponds to permuting $v_1, \ldots, v_m$ and $v'_1, \ldots, v'_m$ together, and similarly for permuting the columns. Multiplying row i by -1 corresponds to interchanging v_i with v'_i, and similarly with columns. Thus, the operations that define Hadamard equivalence map onto graph isomorphism operations. It is less obvious that the same holds in reverse: if B is a second matrix, $G(A)$ is isomorphic to $G(B)$ if and only if A is Hadamard-equivalent to B [3]. Similarly, $\mathrm{Aut}(G(A))$ consists of the operations corresponding the Hadarmard equivalences of A to itself, together with the central element $(v_1\ v'_1)\cdots(v_m\ v'_m)(w_1\ w'_1)\cdots(w_n\ w'_n)$ and a canonical labelling of $G(A)$ can be used to make one of A. We omit the details.

Isotopy of matrices. Two matrices over some symbol set S are called *isotopic* if one can be obtained from the other by permuting the rows, permuting the columns, and permuting the symbols. This equivalence relation is important in the study of Latin squares, quasigroups, and other subjects.

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

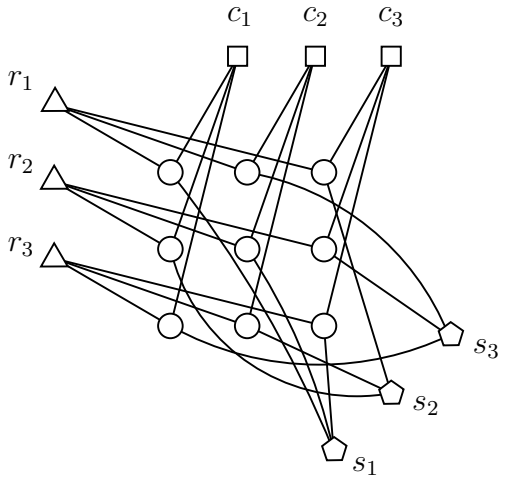


Figure 7: Isotopy as graph isomorphism

Figure 7 shows how to translate isotopy into isomorphism. There are four types of vertex, with four corresponding colours: one vertex for each row, one vertex for each column, one vertex for each symbol, and one vertex for each matrix position. The edges indicate in an obvious fashion what the row, column, and symbol is for each matrix

entry. Other related equivalences, such as paratopy (main class isotopy) can be handled in similar fashion [5].

15 Utilities

The **nauty** package includes a suite of programs called **gtools** that provide efficient processing of files of graphs stored in `graph6` or `sparse6` format. These formats are defined in Section 19.

Most of the **gtools** programs will run on any system with a modern C compiler, but a few need Unix-like facilities. For example, the program `shortg` requires a program compatible with the Unix `sort` program, as well as pipes.

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; see Section 23 for more details.

`geng` : generate small graphs

`genbg` : generate small bicoloured graphs

`gentourng` : generate small tournaments

`directg` : generate small digraphs with given underlying graph

`watercluster2` : a faster alternative to `direct` written by Gunnar Brinkmann

`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 subset 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

`linegraphg` : make the linegraphs of a file of graphs

`subdivideg` : make the subdivision graphs of a file of graphs

`ranlabg` : randomly relabel graphs

`planarg` : test graphs for planarity and find embeddings or obstructions. The planarity code uses the method of Boyer and Myrvold [1] and was programmed by Paulette Lieby for the Magma project.

Further programs will be added. Requests are welcome.

16 Installing nauty and Traces

nauty is prepared using the `autoconf` configuration system. You need a C compiler and library at least recent enough to support the basic commands of ANSI C.

If you have a Unix-like operating system, which includes Linux, MacOSX or a Windows environment such as `cygwin`, first run the configuration script like this:

```
./configure
```

That will examine the facilities available on your computer and generate a custom `makefile` and custom definition files `nauty.h`, `gtools.h` and `naututil.h`. Then run

```
make
```

to compile the basic files of **nauty** and **Traces**.

This will generate the executable file `dreadnaut` and a lot of object files and libraries. For compiling your own programs, the most convenient way to link with **nauty** or **Traces** is to use the static library `nauty.a`. For example, with the `gcc` compiler, a simple compilation might be

```
gcc -o myprog myprog.c nauty.a
```

Execution of `make` without arguments will also generate the library `nauty1.a`, which is the same except that MAXN is defined to be equal to WORDSIZE. This is more efficient if you only want to process small graphs.

For both `nauty.a` and `nauty1.a`, WORDSIZE is determined automatically as specified in Section 3. However, there may be times when you want to use a particular word size, so you can also make the following variants by, for example, `make nautyL1.a` :

`nautyW.a`: WORDSIZE = 32, unlimited MAXN

`nautyW1.a`: WORDSIZE = MAXN = 32

`nautyL.a`: WORDSIZE = 64, unlimited MAXN

`nautyL1.a`: WORDSIZE = MAXN=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.

To use these versions, you need to compile your program in the same way. For example:

```
gcc -o myprog -DWORDSIZE=64 -DMAXN=WORDSIZE myprog.c nautyL1.a
```

Using the function `nauty_check()` in your program is a way to check that you compiled it correctly; see Section 11.

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

```
make checks
```

followed by

```
./runalltests
```

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

If you are on a system where `configure` doesn't work or `make` is not available, 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.basic` as a guide.

Some compilation options are provided at the `configure` stage. Run `./configure --help` to see the syntax and more information.

(a) The choice of compiler, compiler options, linker options, libraries and include files can be influenced by the environment variables CC, CFLAGS, LDFLAGS, LIBS and CPPFLAGS. For example, to compile a 32-bit program on a 64-bit Intel system, this might work (assuming a Bourne-type shell such as `bash`):

```
CFLAGS=-m32 LDFLAGS=-m32 ./configure
```

(Note that the hardware 32/64-bit mode is entirely independent of the value of WORDSIZE.)

(b) Section 3 describes how WORDSIZE is chosen when no explicit definition of it appears. To override this default rule, you can use

```
./configure --enable-wordsize=NN
```

where `NN` is 16, 32 or 64. However, an explicit definition of WORDSIZE at compile time takes precedence even over this.

(c) Generally **nauty** and **Traces** are not thread-safe. However, if your compiler supports thread-local storage, configuring with

```
./configure --enable-tls
```

will mark static and external variables as thread-local. (The most common syntax is to add the attribute `__thread`.) This means that **nauty** or **Traces** can be invoked at the same time by different threads. This may slow it down slightly if you aren't using threads. There are sample programs `nauthread1.c` and `nauthread2.c` in the distribution.

(d) Some C compilers and processors support commands like `__builtin_clz()` that locate the first 1-bit in a word faster than **nauty**'s standard macros can do it. These are detected automatically and influence the definition of the FIRSTBIT and TAKEBIT macros. However, if this causes trouble for some reason, you can turn off this feature using

```
./configure --disable-clz
```

(e) The output written by **Traces** and some of the output written by **dreadnaut** will look prettier on terminals that support ANSI control sequences if you use

```
./configure --enable-ansicontrols
```

Don't use this if you plan to read **dreadnaut** output with a program.

17 Recent changes

See the file `changes24-25.txt` for a list of changes made since version 2.4.

18 More on automorphism groups

nauty and **Traces** use the Random Schreier Method to process the automorphism group as it is found. For **nauty**, this is optional: see the field `schreier` of the options. Difficult graphs with substantial automorphism groups will benefit the most from this addition.

The Random Schreier Method is a probabilistic algorithm that determines information about the group only with some probability. However, this nondeterminism does not effect the result of **nauty** or **Traces**. Occasionally a different set of generators will be found, but the group generated will be the same and the canonical labelling will be unaffected.

There is a parameter for tuning the method, which can be changed using the function `schreier_fails()`. See the file `schreier.txt` for documentation of this and the other group functions. The default value is 10, but smaller values may be better if the group is very large. In **dreadnaut**, the `G` command sets this parameter.

18.1 Listing the full automorphism group

The automorphism group of a graph can be exceedingly large, so trying to list it all might be a bad idea. However, it can be done using the code in the files `naugroup.h` and `naugroup.c`, as illustrated by the following program.

This works with sparse **nauty** as well, but not with **Traces**.

18.2 nautyex3.c : Listing the whole automorphism group

```
/* This program prints the entire automorphism group of an n-vertex
   polygon, where n is a number supplied by the user.
*/

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

/*****************************************************************************/

void
writeautom(int *p, int n)
/* Called by allgroup.  Just writes the permutation p. */
{
    int i;

    for (i = 0; i < n; ++i) printf(" %2d",p[i]); printf("\n");
}

/*****************************************************************************/

int
main(int argc, char *argv[])
{
    DYNALLSTAT(graph,g,g_sz);
    DYNALLSTAT(int,lab,lab_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    static DEFAULTOPTIONS_GRAPH(options);
    statsblk stats;

    int n,m,v;
    grouprec *group;

 /* The following cause nauty to call two procedures which
        store the group information as nauty runs. */

    options.userautomproc = groupautomproc;
    options.userlevelproc = grouplevelproc;

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
            m = SETWORDSNEEDED(n);
```

```
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

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

            EMPTYGRAPH(g,m,n);
            for (v = 0; v < n; ++v) ADDONEEDGE(g,v,(v+1)%n,m);

            printf("Automorphisms of C[%d]:\n",n);
            densenauty(g,lab,ptn,orbits,&options,&stats,m,n,NULL);

         /* Get a pointer to the structure in which the group information
            has been stored.  If you use TRUE as an argument, the
            structure will be "cut loose" so that it won't be used
            again the next time nauty() is called.  Otherwise, as
            here, the same structure is used repeatedly. */

            group = groupptr(FALSE);

         /* Expand the group structure to include a full set of coset
            representatives at every level.  This step is necessary
            if allgroup() is to be called. */

            makecosetreps(group);

         /* Call the procedure writeautom() for every element of the group.
            The first call is always for the identity. */

            allgroup(group,writeautom);
        }
        else
            break;
    }
    exit(0);
}
```

18.3 Giving known generators to Traces

Traces provides the possibility of giving it known automorphisms. This is only likely to be useful for very regular graphs that have automorphisms which are difficult to discover.

The method is illustrated in the following sample program.

18.4 nautyex9.c : Giving known generators to Traces

```
/* This program demonstrates how known automorphisms can be given
   to Traces.  We compute the automorphism group of the circulant
   graph of order n with i is adjacent to j iff j-i is a square
   mod n.  We need that -1 is a square so that the graph is
   undirected, which means that the prime factors of n must be
   congruent to 1 mod 4.  (This is the Paley graph in the event
   that p is a prime.)
*/

#include "traces.h"

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,p,p_sz);
    DYNALLSTAT(boolean,issquare,issquare_sz);
    DYNALLSTAT(int,lab,lab_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    static DEFAULTOPTIONS_TRACES(options);
    TracesStats stats;
 /* Declare and initialize sparse graph structures */
    SG_DECL(sg);

    int deg,n,m,i,j;
    size_t k;
    permnode *gens;

 /* Select option for passing generators to Traces */

    options.generators = &gens;

 /* Read a number of vertices and process it */

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 2)
        {
            m = SETWORDSNEEDED(n);
            nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

            DYNALLOC1(int,lab,lab_sz,n,"malloc");
            DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
            DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
```

```
   DYNALLOC1(int,p,p_sz,n,"malloc");
   DYNALLOC1(boolean,issquare,issquare_sz,n,"malloc");

/* Initialise list of automorphisms */

   gens = NULL;

/* Find the squares and the degree */

   for (i = 0; i < n; ++i) issquare[i] = FALSE;
   for (i = 0; i < n; ++i) issquare[(i*i)%n] = TRUE;
   if (!issquare[n-1])
   {
       printf("-1 must be a square mod n; try again\n");
       continue;
   }

   deg = 0;
   for (i = 1; i < n; ++i) if (issquare[i]) ++deg;

/* Now make the graph */

   SG_ALLOC(sg,n,n*deg,"malloc");
   sg.nv = n;               /* Number of vertices */
   sg.nde = n*deg;            /* Number of directed edges */

   for (i = 0; i < n; ++i)
   {
       sg.v[i] = i*deg;     /* Position of vertex i in v array */
       sg.d[i] = deg;       /* Degree of vertex i */
   }

   for (i = 0; i < n; ++i)   /* Edges */
   {
       k = sg.v[i];
       for (j = 1; j < n; ++j)
           if (issquare[j]) sg.e[k++] = (i + j) % n;
   }

/* Add known automorphisms */

   /* We wouldn't need freeschreier() if we were only
      processing one graph, but it doesn't hurt.  This
      is how to properly dispose of previous generators. */

   freeschreier(NULL,&gens);

   /* Cyclic rotation */
```

```
            for (i = 0; i < n; ++i) p[i] = (i + 1) % n;
            addpermutation(&gens,p,n);

            /* Reflection about 0 */
            for (i = 0; i < n; ++i) p[i] = (n - i) % n;
            addpermutation(&gens,p,n);

         /* Call Traces */

            Traces(&sg,lab,ptn,orbits,&options,&stats,NULL);

            printf("Automorphism group size = ");
            writegroupsize(stdout,stats.grpsize1,stats.grpsize2);
            printf("\n");

        /* Traces left the automorphims we gave it, augmented by
           any extra automorphims it found, in a circular list
           pointed to by gens.  See schreier.txt for documentation. */
        }
        else
            break;
    }

    exit(0);
}
```

18.5 nautyex10.c : Two-step canonical labelling with Traces

Traces uses a different tree-traversal method when it only wants to compute the automorphism group only, without canonical labelling. For some types of difficult graphs, this can be much faster than canonical labelling.

Since **Traces** can also make use of known automorphisms, it is sometimes faster to compute the group first and then use the group for computing the canonical labelling. In **dreadnaut** this is achieved with the sequence `P -c x c x`. The following illustrates how to achieve the same in a program.

An extension (not shown) would be to use the orbit sizes from the first computation to partition the graph before the second computation.

```
/* This program demonstrates how an isomorphism is found between
   two graphs, using the Moebius graph as an example.
   This version uses Traces and demonstrates how to compute the
   automorphism group separately before computing the canonical
   labelling.  Although this is slower for easy graphs like
   those here, it can be faster for some very difficult graphs.
*/

#include "traces.h"

int
main(int argc, char *argv[])
{
    DYNALLSTAT(int,lab1,lab1_sz);
    DYNALLSTAT(int,lab2,lab2_sz);
    DYNALLSTAT(int,ptn,ptn_sz);
    DYNALLSTAT(int,orbits,orbits_sz);
    DYNALLSTAT(int,map,map_sz);
    static DEFAULTOPTIONS_TRACES(options);
    TracesStats stats;
    permnode *generators;
 /* Declare and initialize sparse graph structures */
    SG_DECL(sg1); SG_DECL(sg2);
    SG_DECL(cg1); SG_DECL(cg2);

    int n,m,i;

 /* Read a number of vertices and process */

    while (1)
    {
        printf("\nenter n : ");
        if (scanf("%d",&n) == 1 && n > 0)
        {
```

```
    if (n%2 != 0)
    {
        fprintf(stderr,"Sorry, n must be even\n");
        continue;
    }

    m = SETWORDSNEEDED(n);
    nauty_check(WORDSIZE,m,n,NAUTYVERSIONID);

    DYNALLOC1(int,lab1,lab1_sz,n,"malloc");
    DYNALLOC1(int,lab2,lab2_sz,n,"malloc");
    DYNALLOC1(int,ptn,ptn_sz,n,"malloc");
    DYNALLOC1(int,orbits,orbits_sz,n,"malloc");
    DYNALLOC1(int,map,map_sz,n,"malloc");

/* Now make the first graph */

    SG_ALLOC(sg1,n,3*n,"malloc");
    sg1.nv = n;              /* Number of vertices */
    sg1.nde = 3*n;           /* Number of directed edges */

    for (i = 0; i < n; ++i)
    {
        sg1.v[i] = 3*i;     /* Position of vertex i in v array */
        sg1.d[i] = 3;       /* Degree of vertex i */
    }

    for (i = 0; i < n; i += 2)   /* Spokes */
    {
        sg1.e[sg1.v[i]] = i+1;
        sg1.e[sg1.v[i+1]] = i;
    }

    for (i = 0; i < n-2; ++i)  /* Clockwise edges */
        sg1.e[sg1.v[i]+1] = i+2;
    sg1.e[sg1.v[n-2]+1] = 1;
    sg1.e[sg1.v[n-1]+1] = 0;

    for (i = 2; i < n; ++i)  /* Anticlockwise edges */
        sg1.e[sg1.v[i]+2] = i-2;
    sg1.e[sg1.v[1]+2] = n-2;
    sg1.e[sg1.v[0]+2] = n-1;

/* Now make the second graph */

    SG_ALLOC(sg2,n,3*n,"malloc");
    sg2.nv = n;              /* Number of vertices */
    sg2.nde = 3*n;           /* Number of directed edges */
```

```
    for (i = 0; i < n; ++i)
    {
        sg2.v[i] = 3*i;
        sg2.d[i] = 3;
    }

    for (i = 0; i < n; ++i)
    {
        sg2.v[i] = 3*i;
        sg2.d[i] = 3;
        sg2.e[sg2.v[i]] = (i+1) % n;      /* Clockwise */
        sg2.e[sg2.v[i]+1] = (i+n-1) % n;  /* Anti-clockwise */
        sg2.e[sg2.v[i]+2] = (i+n/2) % n;  /* Diagonals */
    }

/* Now we make the canonically labelled graphs by a two-step
   process.  The first call to Traces computes the
   automorphism group.  The second call computes the
   canonical labelling, using the automorphism group from
   the first call.

   We have declared a variable "generators" that will be
   used to hold the group generators between the two calls.
   It has to be initialised to NULL and its address has to
   be given to Traces using options.generators.  After the
   second call, we need to discard the generators with a
   call to freeschreier(), which also initializes it again. */

    generators = NULL;
    options.generators = &generators;

    options.getcanon = FALSE;
    Traces(&sg1,lab1,ptn,orbits,&options,&stats,NULL);
    options.getcanon = TRUE;
    Traces(&sg1,lab1,ptn,orbits,&options,&stats,&cg1);
    freeschreier(NULL,&generators);

    options.getcanon = FALSE;
    Traces(&sg2,lab1,ptn,orbits,&options,&stats,NULL);
    options.getcanon = TRUE;
    Traces(&sg2,lab1,ptn,orbits,&options,&stats,&cg2);
    freeschreier(NULL,&generators);

/* Compare canonically labelled graphs */

    if (aresame_sg(&cg1,&cg2))
    {
```

```
                printf("Isomorphic.\n");
                if (n <= 1000)
                {
                 /* Write the isomorphism.  For each i, vertex lab1[i]
                    of sg1 maps onto vertex lab2[i] of sg2.  We compute
                    the map in order of labelling because it looks better. */

                    for (i = 0; i < n; ++i) map[lab1[i]] = lab2[i];
                    for (i = 0; i < n; ++i) printf(" %d-%d",i,map[i]);
                    printf("\n");
                }
            }
            else
                printf("Not isomorphic.\n");
        }
        else
            break;
    }

    exit(0);
}
```

19 Graph formats used by the utilities

This is the file `formats.txt`.

```
Description of graph6 and sparse6 encodings
-------------------------------------------
Brendan McKay, bdm@cs.anu.edu.au
Updated May 2005.

General principles:

  All numbers in this description are in decimal unless obviously
  in binary.

  Apart from the header, there is one object per line.  Apart from the
  header and the end-of-line characters, all bytes have a value in the
  range 63-126 (which are all printable ASCII characters).  A file of
  objects is a text file, so whatever end-of-line convention is locally
  used is fine).

Bit vectors:

  A bit vector x of length k can be represented as follows.
      Example:  1000101100011100

  (1) Pad on the right with 0 to make the length a multiple of 6.
      Example:  100010110001110000

  (2) Split into groups of 6 bits each.
      Example:  100010 110001 110000

  (3) Add 63 to each group, considering them as bigendian binary numbers.
      Example:  97 112 111

  These values are then stored one per byte.
  So, the number of bytes is ceiling(k/6).

  Let R(x) denote this representation of x as a string of bytes.

Small nonnegative integers:

  Let n be an integer in the range 0-68719476735 (2^36-1).

  If 0 <= n <= 62, define N(n) to be the single byte n+63.
  If 63 <= n <= 258047, define N(n) to be the four bytes
      126 R(x), where x is the bigendian 18-bit binary form of n.
```

```
  If 258048 <= n <= 68719476735, define N(n) to be the eight bytes
      126 126 R(x), where x is the bigendian 36-bit binary form of n.

  Examples:  N(30) = 93
             N(12345) = N(000011 000000 111001) = 126 66 63 120
             N(460175067) = N(000000 011011 011011 011011 011011 011011)
                          = 126 126 63 90 90 90 90 90


Description of graph6 format.
-----------------------------

Data type:
   simple undirected graphs of order 0 to 68719476735.

Optional Header:
   >>graph6<<     (without end of line!)

File name extension:
   .g6

One graph:
   Suppose G has n vertices.  Write the upper triangle of the adjacency
   matrix of G as a bit vector x of length n(n-1)/2, using the ordering
   (0,1),(0,2),(1,2),(0,3),(1,3),(2,3),...,(n-1,n).

   Then the graph is represented as  N(n) R(x).

Example:
   Suppose n=5 and G has edges 0-2, 0-4, 1-3 and 3-4.

   x = 0 10 010 1001

   Then N(n) = 68 and R(x) = R(010010 100100) = 81 99.
   So, the graph is  68 81 99.


Description of sparse6 format.
------------------------------

Data type:
   Undirected graphs of order 0 to 68719476735.
   Loops and multiple edges are permitted.

Optional Header:
   >>sparse6<<     (without end of line!)

File name extension:
```

```
   .s6

General structure:

  Each graph occupies one text line.  Except for end-of-line characters,
  each byte has the form 63+x, where 0 <= x <= 63.  The byte encodes
  the six bits of x.

  The encoded graph consists of:
(1) The character ':'.   (This is present to distinguish
                                  the code from graph6 format.)
        (2) The number of vertices.
        (3) A list of edges.
        (4) end-of-line

  Loops and multiple edges are supported, but not directed edges.

Number of vertices n:

  1, 4, or 8 bytes N(n) as above.
  This is the same as graph6 format.

List of edges:

  Let k be the number of bits needed to represent n-1 in binary.

  The remaining bytes encode a sequence

      b[0] x[0] b[1] x[1] b[2] x[2] ... b[m] x[m]

  Each b[i] occupies 1 bit, and each x[i] occupies k bits.
  Pack them together in bigendian order, and pad up to a
  multiple of 6 as follows:
  1. If (n,k) = (2,1), (4,2), (8,4) or (16,5), and vertex
     n-2 has an edge but n-1 doesn't have an edge, and
     there are k+1 or more bits to pad, then pad with one
     0-bit and enough 1-bits to complete the multiple of 6.
  2. Otherwise, pad with enough 1-bits to complete the
     multiple of 6.
  These rules are to match the gtools procedures, and to avoid
  the padding from looking like an extra loop in unusual cases.

  Then represent this bit-stream 6 bits per byte as indicated above.

  The vertices of the graph are 0..n-1.
  The edges encoded by this sequence are determined thus:

     v = 0
```

```
    for i from 0 to m do
       if b[i] = 1 then v = v+1 endif;
       if x[i] > v then v = x[i] else output {x[i],v} endif
    endfor

  In decoding, an incomplete (b,x) pair at the end is discarded.

Example:

  :Fa@x^

  ':' indicates sparse6 format.
  Subtract 63 from the other bytes and write them in binary,
  six bits each.

   000111 100010 000001 111001 011111

  The first byte is not 63, so it is n.  n=7
  n-1 needs 3 bits (k=3).  Write the other bits in groups
  of 1 and k:

    1 000  1 000  0 001  1 110   0 101  1 111

    This is the b/x sequence  1,0 1,0 0,1 1,6 0,5 1,7.
    The 1,7 at the end is just padding.
    The remaining parts give the edges 0-1 0-2 1-2 5-6.
```

For a description of the `planarcode` and `edgecode` formats, see the `plantri` documentation at http://cs.anu.edu.au/~bdm/plantri.

20 Other ways to use nauty

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.maths.qmul.ac.uk/~leonard/grape/
(c) LINK: http://dimacs.rutgers.edu/~berryj/LINK.html
(d) Sage-combinat: http://mupad-combinat.sourceforge.net/
(e) Macaulay2: http://mupad-combinat.sourceforge.net/

21 Licence details

nauty and **Traces** are copyright to Brendan McKay and Adolfo Piperno, except as below. They can be used freely for most non-profit purposes; please see `nauty.h` for more details.

The file `planarity.c` contains code written by Paulette Lieby for the Magma project. The file `watercluster2.c` is copyright to Gunnar Brinkmann.

22 Acknowledgements

So many people have made contributions to **nauty** that listing them all would be futile. Heart-felt thanks to all of you.

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

23 Help texts for the utilities

```
===== addedgeg ====================================================

Usage: addedgeg [-lq] [-D#] [-btfF] [-z#] [infile [outfile]]

 For each edge nonedge e, output G+e if it satisfies certain conditions

    The output file has a header if and only if the input file does.

    -l  Canonically label outputs
    -D# Specify an upper bound on the maximum degree of the output
    -b  Output has no new cycles of odd length
    -t  Output has no new 3-cycle if input doesn't
    -f  Output has no new 4-cycle if input doesn't
    -F  Output has no new 5-cycle if input doesn't
    -z# Output has no new cycles of length less than #
        -btfFz can be used in arbitrary combinations
    -q  Suppress auxiliary information

===== amtog =======================================================

Usage: amtog [-n#sghq] [infile [outfile]]

 Read graphs in matrix format.

    -n#   Set the initial graph order to # (no default).
          This can be overridden in the input.
    -g    Write the output in graph6 format (default).
    -s    Write the output in sparse6 format.
    -h    Write a header (according to -g or -s).
    -q    Suppress auxiliary information.

   Input consists of a sequence of commands restricted to:

    n=#   set number of vertices (no default)
          The = is optional.
```

```
    m     Matrix to follow (01 any spacing or no spacing)
          An 'm' is also assumed if 0 or 1 is encountered.
    M     Complement of matrix to follow (as m)
    t     Upper triangle of matrix to follow, row by row
           excluding the diagonal. (01 in any or no spacing)
    T     Complement of upper trangle to follow (as t)
    q     exit (optional)

===== biplabg ======================================================

Usage: biplabg [-q] [infile [outfile]]

 Label bipartite graphs so that the colour classes are contiguous.
  The first vertex of each component is assigned the first colour.
  Vertices in each colour class have the same relative order as before.
  Non-bipartite graphs are rejected.

    The output file has a header if and only if the input file does.

    -q  Suppress auxiliary information.

===== catg =========================================================

Usage: catg [-xv] [infile]...

  Copy files to stdout with all but the first header removed.

     -x  Don't write a header.
         In the absence of -x, a header is written if
         there is one in the first input file.

     -v  Summarize to stderr.

===== complg =======================================================

Usage: complg [-lrq] [infile [outfile]]

 Take the complements of a file of graphs.

    The output file has a header if and only if the input file does.

    -r  Only complement if the complement has fewer edges.
    -l  Canonically label outputs.
    -q  Suppress auxiliary information.

===== copyg ========================================================

Usage: copyg [-gsfp#:#qhx] [infile [outfile]]
```

```
  Copy a file of graphs with possible format conversion.

     -g  Use graph6 format for output
     -s  Use sparse6 format for output
         In the absence of -g and -s, the format depends on
         the header or, if none, the first input line.

     -p# -p#:#
         Specify range of input lines (first is 1)
     -f  With -p, assume input lines of fixed length
          (ignored if header or first line has sparse6 format).

     -h  Write a header.
     -x  Don't write a header.
         In the absence of -h and -x, a header is written if
         there is one in the input.

     -q  Suppress auxiliary output.

===== countg ====================================================

Usage: [pickg|countg] [-fp#:#q -V] [--keys] [-constraints -v] [ifile [ofile]]

 countg : Count graphs according to their properties.
  pickg : Select graphs according to their properties.

  ifile, ofile : Input and output files.
        '-' and missing names imply stdin and stdout.

  Miscellaneous switches:
     -p# -p#:#   Specify range of input lines (first is 1)
     -f          With -p, assume input lines of fixed length
                        (only used with a file in graph6 format)
     -v          Negate all constraints
     -V          List properties of every input matching constraints.
     -q          Suppress informative output.

  Constraints:
     Numerical constraints (shown here with following #) can take
     a single integer value, or a range like #:#, #:, or :#.  Each
     can also be preceded by '~', which negates it.   (For example,
     -~D2:4 will match any maximum degree which is _not_ 2, 3, or 4.)
     Constraints are applied to all input graphs, and only those
     which match all constraints are counted or selected.

     -n#  number of vertices     -e#  number of edges
     -d#  minimum degree         -D#  maximum degree
```

```
     -m#  vertices of min degree -M#  vertices of max degree
     -r   regular                -b   bipartite
     -z#  radius                 -Z#  diameter
     -g#  girth (0=acyclic)      -Y#  total number of cycles
     -T#  number of triangles    -K#  number of maximal independent sets
     -H#  number of induced cycles
     -E   Eulerian (all degrees are even, connectivity not required)
     -a#  group size  -o# orbits  -F# fixed points  -t vertex-transitive
     -c#  connectivity (only implemented for 0,1,2).
     -i#  min common nbrs of adjacent vertices;     -I# maximum
     -j#  min common nbrs of non-adjacent vertices; -J# maximum

  Sort keys:
     Counts are made for all graphs passing the constraints.  Counts
     are given separately for each combination of values occuring for
     the properties listed as sort keys.  A sort key is introduced by
     '--' and uses one of the letters known as constraints.  These can
     be combined:  --n --e  --r  is the same as --ne --r and --ner.
     The order of sort keys is significant.

===== deledgeg ====================================================

Usage: deledgeg [-lq] [-d#] [infile [outfile]]

 For each edge e, output G-e

    The output file has a header if and only if the input file does.

    -l  Canonically label outputs
    -d# Specify a lower bound on the minimum degree of the output
    -q  Suppress auxiliary information

===== directg =====================================================

Usage: directg [-q] [-u|-T|-G] [-V] [-o] [-f#] [-e#|-e#:#] [infile [outfile]]

 Read undirected graphs and orient their edges in all possible ways.
  Edges can be oriented in either or both directions (3 possibilities).
  Isomorphic directed graphs derived from the same input are suppressed.
  If the input graphs are non-isomorphic then the output graphs are also.

   -e# | -e#:#  specify a value or range of the total number of arcs
    -o     orient each edge in only one direction, never both
   -f#  Use only the subgroup that fixes the first # vertices setwise

    -T  use a simple text output format (nv ne edges) instead of digraph6
    -G  like -T but includes group size as third item (if less than 10^10)
          The group size does not include exchange of isolated vertices.
```

```
    -V  only output graphs with nontrivial groups (including exchange of
          isolated vertices).  The -f option is respected.
    -u  no output, just count them
    -q  suppress auxiliary information

===== dretog ========================================================

Usage: dretog [-n#o#sghq] [infile [outfile]]

 Read graphs in dreadnaut format.

   -o#   Label vertices starting at # (default 0).
         This can be overridden in the input.
   -n#   Set the initial graph order to # (no default).
         This can be overridden in the input.
   -g    Use graph6 format (default).
   -s    Use sparse6 format.
   -h    Write a header (according to -g or -s).

  Input consists of a sequence of dreadnaut commands restricted to:

   n=#   set number of vertices (no default)
         The = is optional.
   $=#   set label of first vertex (default 0)
         The = is optional.
   $$    return origin to initial value (see -o#)
   ".." and !..\n   comments to ignore
   g     specify graph to follow (as dreadnaut format)
         Can be omitted if first character of graph is a digit or ';'.
   q     exit (optional)

===== genbg =========================================================

Usage: genbg [-c -ugs -vq -lzF] [-Z#] [-D#] [-A] [-d#|-d#:#] [-D#|-D#:#] n1 n2
                [mine[:maxe]] [res/mod] [file]

 Find all bicoloured graphs of a specified class.

  n1   : the number of vertices in the first class
  n2   : the number of vertices in the second class
 mine:maxe : a range for the number of edges
              #:0 means '# or more' except in the case 0:0
  res/mod : only generate subset res out of subsets 0..mod-1
   file : the name of the output file (defailt stdout)
  -c    : only write connected graphs
  -z    : all the vertices in the second class must have
          different neighbourhoods
  -F    : the vertices in the second class must have at least two
```

```
          neighbours of degree at least 2
  -L    : there is no vertex in the first class whose removal leaves
          the vertices in the second class unreachable from each other
  -Z#   : two vertices in the second class may have at most # common nbrs
  -A    : no vertex in the second class has a neighbourhood with is a
          subset of another vertex in the second class
  -D#   : specify an upper bound for the maximum degree
          Example: -D6.  You can also give separate maxima for the
          two parts, for example: -D5:6
  -d#   : specify a lower bound for the minimum degree.
          Again, you can specify it separately for the two parts: -d1:2
  -g    : use graph6 format for output (default)
  -s    : use sparse6 format for output
  -a    : use Greechie diagram format for output
  -u    : do not output any graphs, just generate and count them
  -v    : display counts by number of edges to stderr
  -l    : canonically label output graphs (using the 2-part colouring)

  -q    : suppress auxiliary output

  See program text for much more information.

===== geng ========================================================

Usage: geng [-cCmtfbd#D#] [-uygsnh] [-lvq]
             [-x#X#] n [mine[:maxe]] [res/mod] [file]

 Generate all graphs of a specified class.

      n    : the number of vertices
 mine:maxe : a range for the number of edges
             #:0 means '# or more' except in the case 0:0
   res/mod : only generate subset res out of subsets 0..mod-1

     -c    : only write connected graphs
     -C    : only write biconnected graphs
     -t    : only generate triangle-free graphs
     -f    : only generate 4-cycle-free graphs
     -b    : only generate bipartite graphs
                (-t, -f and -b can be used in any combination)
     -m    : save memory at the expense of time (only makes a
                difference in the absence of -b, -t, -f and n <= 28).
     -d#   : a lower bound for the minimum degree
     -D#   : a upper bound for the maximum degree
     -v    : display counts by number of edges
     -l    : canonically label output graphs

     -u    : do not output any graphs, just generate and count them
```

```
     -g    : use graph6 output (default)
     -s    : use sparse6 output
     -y    : use the obsolete y-format instead of graph6 format
     -h    : for graph6 or sparse6 format, write a header too

     -q    : suppress auxiliary output (except from -v)

  See program text for much more information.

===== genrang ======================================================

Usage: genrang [-P#|-P#/#|-e#|-r#|-R#] [-l#] [-m#] [-t] [-a]
         [-s|-g] [-S#] [-q] n num [outfile]

 Generate random graphs.
     n  : number of vertices
    num : number of graphs

    -s  : Write in sparse6 format (default)
    -g  : Write in graph6 format
    -P#/# : Give edge probability; -P# means -P1/#.
    -e# : Give the number of edges
    -r# : Make regular of specified degree
    -R# : Make regular of specified degree but output
          as vertex count, edgecount, then list of edges
    -l# : Maximum loop multiplicity (default 0)
    -m# : Maximum multiplicity of non-loop edge (default and minimum 1)
         -l and -m are only permitted with -R and -r without -g
    -t  : Make a random tree
    -a  : Make invariant under a random permutation
    -S# : Specify random generator seed (default nondeterministic)
    -q  : suppress auxiliary output

    Incompatible: -P,-e,-r,-R,-t; -s,-g,-R; -R,-a; -s,-g & -l,-m.

===== gentourng ====================================================

Usage: gentourng [-cd#D#] [-ugs] [-lq] n [res/mod] [file]

 Generate all tournaments of a specified class.

      n    : the number of vertices
   res/mod : only generate subset res out of subsets 0..mod-1

     -c    : only write strongly-connected tournaments
     -d#   : a lower bound for the minimum out-degree
     -D#   : a upper bound for the maximum out-degree
     -l    : canonically label output graphs
```

```
     -u    : do not output any graphs, just generate and count them
     -g    : use graph6 output (lower triangle)
     -s    : use sparse6 output (lower triangle)
     -h    : write a header (only with -g or -s)
  Default output is upper triangle row-by-row in ascii

     -q    : suppress auxiliary output

  See program text for much more information.

===== labelg ========================================================

Usage: labelg [-qsg] [-fxxx] [-S|-t] [-i# -I#:# -K#] [infile [outfile]]

 Canonically label a file of graphs.

    -s  force output to sparse6 format
    -g  force output to graph6 format
        If neither -s or -g are given, the output format is
        determined by the header or, if there is none, by the
        format of the first input graph. Also see -S.
    -S  Use sparse representation internally.
         Note that this changes the canonical labelling.
         Multiple edges are not supported.  One loop per vertex is ok.
    -t  Use Traces.
         Note that this changes the canonical labelling.
         Multiple edges and loops are not supported, nor invariants.

    The output file will have a header if and only if the input file does.

    -fxxx  Specify a partition of the point set.  xxx is any
        string of ASCII characters except nul.  This string is
        considered extended to infinity on the right with the
        character 'z'.  One character is associated with each point,
        in the order given.  The labelling used obeys these rules:
         (1) the new order of the points is such that the associated
        characters are in ASCII ascending order
         (2) if two graphs are labelled using the same string xxx,
        the output graphs are identical iff there is an
        associated-character-preserving isomorphism between them.
        No option can be concatenated to the right of -f.

    -i#  select an invariant (1 = twopaths, 2 = adjtriang(K), 3 = triples,
        4 = quadruples, 5 = celltrips, 6 = cellquads, 7 = cellquins,
        8 = distances(K), 9 = indsets(K), 10 = cliques(K), 11 = cellcliq(K),
       12 = cellind(K), 13 = adjacencies, 14 = cellfano, 15 = cellfano2,
       16 = refinvar(K))
```

```
    -I#:#  select mininvarlevel and maxinvarlevel (default 1:1)
    -K#   select invararg (default 3)

    -q  suppress auxiliary information

===== linegraphg ========================================================

Usage: linegraphg [-q] [infile [outfile]]

 Take the linegraphs of a file of graphs.
  Input graphs with no edges produce only a warning message.

    The output file has a header if and only if the input file does.

    -q  Suppress auxiliary information.

===== listg =============================================================

Usage: listg [-fp#:#l#o#Ftq] [-a|-A|-c|-d|-e|-H|-M|-s|-b|-G|-y|-Yxxx] [infile [outfile]]

 Write graphs in human-readable format.

    -f  : assume inputs have same size (only used from a file
          and only if -p is given)
    -p#, -p#:#, -p#-# : only display one graph or a sequence of
          graphs.  The first graph is number 1.  A second number
          which is empty or zero means infinity.
    -a  : write as adjacency matrix, not as list of adjacencies
    -A  : same as -a with a space between entries
    -l# : specify screen width limit (default 78, 0 means no limit)
          This is not currently implemented with -a or -A.
    -o# : specify number of first vertex (default is 0).
    -d  : write output to satisfy dreadnaut
    -c  : write ascii form with minimal line-breaks
    -e  : write a list of edges, preceded by the order and the
          number of edges
    -H  : write in HCP operations research format
    -M  : write in Magma format
    -W  : write matrix in Maple format
    -b  : write in Bliss format
    -G  : write in GRAPE format
    -y  : write in dot file format
    -Yxxx : extra dotty commands for dot files (arg continues to end of param)
    -t  : write upper triangle only (affects -a, -A, -d and default)
    -s  : write only the numbers of vertices and edges
    -F  : write a form-feed after each graph except the last
    -q  : suppress auxiliary output
```

```
    -a, -A, -c, -d, -M, -W, -H and -e are incompatible.

===== multig ========================================================

Usage: multig [-q] [-u|-T|-G|-A|-B] [-e#|-e#:#]
       [-m#] [-f#] [-D#|-r#|-l#] [infile [outfile]]

 Read undirected loop-free graphs and replace their edges with multiple
  edges in all possible ways (multiplicity at least 1).
  Isomorphic multigraphs derived from the same input are suppressed.
  If the input graphs are non-isomorphic then the output graphs are also.

    -e# | -e#:#  specify a value or range of the total number of edges
                 counting multiplicities
    -m# maximum edge multiplicity (minimum is 1)
    -D# upper bound on maximum degree
    -r# make regular of specified degree (incompatible with -l, -D, -e)
    -l# make regular multigraphs with multiloops, degree #
                 (incompatible with -r, -D, -e)
    Either -l, -r, -D, -e or -m with a finite maximum must be given
    -f# Use the group that fixes the first # vertices setwise
    -T  use a simple text output format (nv ne {v1 v2 mult})
    -G  like -T but includes group size as third item (if less than 10^10)
          The group size does not include exchange of isolated vertices.
    -A  write as the upper triangle of an adjacency matrix, row by row,
        including the diagonal, and preceded by the number of vertices
    -B  write as an integer matrix preceded by the number of rows and
        number of columns, where -f determines the number of rows
    -u  no output, just count them
    -q  suppress auxiliary information

===== newedgeg ======================================================

Usage: newedgeg [-lq] [infile [outfile]]

 For each pair of non-adjacent edges, output the graph obtained
  by subdividing the edges and joining the new vertices.

    The output file has a header if and only if the input file does.

    -l  Canonically label outputs
    -q  Suppress auxiliary information

===== NRswitchg =====================================================

Usage: NRswitchg [-lq] [infile [outfile]]

 For each v, complement the edges from N(v) to V(G)-N(v)-v.
```

```
    The output file has a header if and only if the input file does.

    -l  Canonically label outputs.
    -q  Suppress auxiliary information.

===== pickg ====================================================

Usage: [pickg|countg] [-fp#:#q -V] [--keys] [-constraints -v] [ifile [ofile]]

 countg : Count graphs according to their properties.
  pickg : Select graphs according to their properties.

  ifile, ofile : Input and output files.
        '-' and missing names imply stdin and stdout.

  Miscellaneous switches:
     -p# -p#:#   Specify range of input lines (first is 1)
     -f          With -p, assume input lines of fixed length
                        (only used with a file in graph6 format)
     -v          Negate all constraints
     -V          List properties of every input matching constraints.
     -q          Suppress informative output.

  Constraints:
     Numerical constraints (shown here with following #) can take
     a single integer value, or a range like #:#, #:, or :#.  Each
     can also be preceded by '~', which negates it.   (For example,
     -~D2:4 will match any maximum degree which is _not_ 2, 3, or 4.)
     Constraints are applied to all input graphs, and only those
     which match all constraints are counted or selected.

     -n#  number of vertices     -e#  number of edges
     -d#  minimum degree         -D#  maximum degree
     -m#  vertices of min degree -M#  vertices of max degree
     -r   regular                -b   bipartite
     -z#  radius                 -Z#  diameter
     -g#  girth (0=acyclic)      -Y#  total number of cycles
     -T#  number of triangles    -K#  number of maximal independent sets
     -H#  number of induced cycles
     -E   Eulerian (all degrees are even, connectivity not required)
     -a#  group size  -o# orbits  -F# fixed points  -t vertex-transitive
     -c#  connectivity (only implemented for 0,1,2).
     -i#  min common nbrs of adjacent vertices;     -I# maximum
     -j#  min common nbrs of non-adjacent vertices; -J# maximum

  Sort keys:
     Counts are made for all graphs passing the constraints.  Counts
```

```
     are given separately for each combination of values occuring for
     the properties listed as sort keys.  A sort key is introduced by
     '--' and uses one of the letters known as constraints.  These can
     be combined:  --n --e  --r  is the same as --ne --r and --ner.
     The order of sort keys is significant.

===== planarg ======================================================

Usage: planarg [-v] [-nVq] [-p|-u] [infile [outfile]]

 For each input, write to output if planar.

    The output file has a header if and only if the input file does.

    -v  Write non-planar graphs instead of planar graphs
    -V  Write report on every input
    -u  Don't write anything, just count
    -p  Write in planar_code if planar (without -p, same format as input)
    -k  Follow each non-planar output with an obstruction in sparse6
        format (implies -v, incompatible with -p)
    -n  Suppress checking of the result
    -q  Suppress auxiliary information

    This program permits multiple edges and loops

===== ranlabg ======================================================

Usage: ranlabg [-q] [-f#] [-S#] [infile [outfile]]

 Randomly relabel graphs.

    The output file has a header if and only if the input file does.

    -f# Don't relabel the first # vertices.
    -m# Output # randomly labelled copies of each input (default 1).
    -S# Set random number seed (taken from clock otherwise).

    -q  Suppress auxiliary information.

===== shortg =======================================================

Usage: shortg [-qvkdu] [-i# -I#:# -K#] [-fxxx] [-S|-t] [-Tdir] [infile [outfile]]

  Remove isomorphs from a file of graphs.

    If outfile is omitted, it is taken to be the same as infile
    If both infile and outfile are omitted, input will be taken
            from stdin and written to stdout
```

```
The output file has a header if and only if the input file does.

-s  force output to sparse6 format
-g  force output to graph6 format
    If neither -s or -g are given, the output format is
    determined by the header or, if there is none, by the
    format of the first input graph.
-S  Use sparse representation internally. Note that this changes the
    canonical labelling.
    Multiple edges are not supported.  One loop per vertex is ok.
-t  Use Traces.
    Note that this changes the canonical labelling.
    Multiple edges and loops are not supported, nor invariants.

-k  output graphs have the same labelling and format as the inputs.
    Without -k, output graphs have canonical labelling.
    -s and -g are ineffective if -k is given.

-v  write to stderr a list of which input graphs correspond to which
    output graphs. The input and output graphs are both numbered
    beginning at 1.  A line like
       23 : 30 154 78
    means that inputs 30, 154 and 78 were isomorphic, and produced
    output 23.

-d  include in the output only those inputs which are isomorphic
    to another input.  If -k is specified, all such inputs are
    included in their original labelling.  Without -k, only one
    member of each nontrivial isomorphism class is written,
    with canonical labelling.

-fxxx  Specify a partition of the point set.  xxx is any
    string of ASCII characters except nul.  This string is
    considered extended to infinity on the right with the
    character 'z'.  One character is associated with each point,
    in the order given.  The labelling used obeys these rules:
     (1) the new order of the points is such that the associated
    characters are in ASCII ascending order
     (2) if two graphs are labelled using the same string xxx,
    the output graphs are identical iff there is an
    associated-character-preserving isomorphism between them.
-i#  select an invariant (1 = twopaths, 2 = adjtriang(K), 3 = triples,
    4 = quadruples, 5 = celltrips, 6 = cellquads, 7 = cellquins,
    8 = distances(K), 9 = indsets(K), 10 = cliques(K), 11 = cellcliq(K),
   12 = cellind(K), 13 = adjacencies, 14 = cellfano, 15 = cellfano2,
   16 = refinvar(K))
-I#:#  select mininvarlevel and maxinvarlevel (default 1:1)
```

```
    -K#   select invararg (default 3)

    -u  Write no output, just report how many graphs it would have output.
        In this case, outfile is not permitted.
    -Tdir  Specify that directory "dir" will be used for temporary disk
        space by the sort subprocess.  The default is usually /tmp.
    -q  Suppress auxiliary output

===== showg ====================================================
Usage: showg [-p#:#l#o#Ftq] [-a|-A|-c|-d|-e] [infile [outfile]]

 Write graphs in human-readable format.

   infile is the input file in graph6 or sparse6 format
   outfile is the output file
   Defaults are standard input and standard output.

    -p#, -p#:#, -p#-# : only display one graph or a sequence of
          graphs.  The first graph is number 1.  A second number
          which is empty or zero means infinity.

    -a  : write the adjacency matrix
    -A  : same as -a with a space between entries
    -d  : write output to satisfy dreadnaut
    -c  : write compact dreadnaut form with minimal line-breaks
    -e  : write a list of edges, preceded by the order and the
          number of edges

    -o# : specify number of first vertex (default is 0)
    -t  : write upper triangle only (affects -a, -A, -d and default)
    -F  : write a form-feed after each graph except the last
    -l# : specify screen width limit (default 78, 0 means no limit)
          This is not currently implemented with -a or -A.
    -q  : suppress auxiliary output

    -a, -A, -c, -d and -e are incompatible.

===== subdivideg ================================================

Usage: subdivideg [-k#] [-q] [infile [outfile]]

 Make the subdivision graphs of a file of graphs.
    -k#  Subdivide each edge by # new vertices (default 1)

    The output file has a header if and only if the input file does.

    -q  Suppress auxiliary information.
```

```
===== watercluster2 ====================================================
usage: watercluster2 [ix] [oy] [m] [T] [C] [B] [S] .
The option ix restricts the maximum indegree to x.
The option oy restricts the maximum outdegree to y.
The default maximum in- and out-degrees are unlimited.
T means: Output directed graphs in T-code -- for details see header
B means: Output directed graphs in binary code -- for details see header
C means: Do really construct all the directed graphs in memory, but don't output them.
S means that for each edge only one direction must be chosen -- not both.
Default is that both are allowed
  -- so the edge a-b can become a->b AND b->a in the same output graph.
m means: read multicode instead of g6 code
```

References

[1] J. M. Boyer and W. J. Myrvold, On the cutting edge: simplified $O(n)$ planarity by edge addition, *J. Graph Alg. Appl.*, **8** (2004) 241–273.

[2] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, Exploiting Structure in Symmetry Generation for CNF, Proceedings of the 41st Design Automation Conference, 2004, 530–534. Source code at http://vlsicad.eecs.umich.edu/BK/SAUCY.

[3] B. D. McKay, Hadamard equivalence via graph isomorphism, *Discrete Math.*, **27** (1979) 213–214.

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

[5] B. D. McKay, A. Meynert and W. Myrvold, Small Latin squares, quasigroups and loops, *J. Combin. Designs*, to appear.

[6] T. Miyazaki, The complexity of McKay's canonical labelling algorithm, *in* Groups and Computation, II, *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, **28**, Amer. Math. Soc. (1997) 239–256.

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

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

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

[10] B. D. McKay and Adolfo Piperno, Practical graph isomorphism II, *submitted.* Preprint available at http://pallini.di.uniroma1.it.