TLP **16** (5–6): 688–702, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000223

*Logic Programming with Graph Automorphism: Integrating nauty with Prolog (Tool Description)**

MICHAEL FRANK and MICHAEL CODISH

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
(*e-mail:* `frankm,mcodish}@cs.bgu.ac.il`)

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

This paper presents the `pl-nauty` library, a Prolog interface to the `nauty` graph-automorphism tool. Adding the capabilities of `nauty` to Prolog combines the strength of the "generate and prune" approach that is commonly used in logic programming and constraint solving, with the ability to reduce symmetries while reasoning over graph objects. Moreover, it enables the integration of `nauty` in existing tool-chains, such as SAT-solvers or finite domain constraints compilers which exist for Prolog. The implementation consists of two components: `pl-nauty`, an interface connecting `nauty`'s C library with Prolog, and `pl-gtools`, a Prolog framework integrating the software component of `nauty`, called `gtools`, with Prolog. The complete tool is available as a SWI-Prolog module. We provide a series of usage examples including two that apply to generate Ramsey graphs.

1 Introduction

Many problems, particularly in combinatorics, reduce to asking whether some graph with a given property exists, or alternatively, asking how many such non-isomorphic graphs exist. Such graph search and graph enumeration problems are notoriously difficult, in no small part due to the extremely large number of symmetries in graphs. In practical problem solving, it is often advantageous to eliminate these symmetries which arise naturally due to graph isomorphism: typically, if a graph G is a solution then so is any other graph G' that is isomorphic to G.

General approaches to graph search problems typically involve either: *generate and test*, explicitly enumerating all (non-isomorphic) graphs and checking each for the given property, or *constrain and generate*, encoding the problem for some general-purpose discrete satisfiability solver (i.e. SAT, integer programming, constraint

* Supported by the Israel Science Foundation, grant 182/13.

programming), which does the enumeration implicitly. In the explicit approach, one typically iterates, repeatedly applying an extend and reduce approach: First *extend* the set of all non-isomorphic graphs with n vertices, in all possible ways, to graphs with $n + 1$ vertices; and then *reduce* the extensions to their non-isomorphic (canonical) representatives. In the constraint based approach, one typically first encodes the problem and then applies a constraint solver in order to produce solutions. The (unknown) graph is represented in terms of Boolean variables describing it as an adjacency matrix A. The encoding is a conjunction of constraints that constitute a model, φ_A, such that any satisfying assignment to φ_A is a solution to the graph search problem. Typically, symmetry breaking constraints (Crawford *et al.* 1996; Codish *et al.* 2013) are added to the model to reduce the number of isomorphic solutions, while maintaining the correctness of the model.

It remains unknown whether a polynomial time algorithm exists to decide the graph isomorphism problem. Nevertheless, finding good graph isomorphism algorithms is critical when exploring graph search and enumeration problems. Recently an algorithm was published by Babai (2015) which runs in time $O(\exp(log^c(n)))$, for some constant $c > 1$, and solves the graph isomorphism problem. Nevertheless, top of the line graph isomorphism tools use different methods, which are, in practice, faster.

McKay (1981) introduces an algorithm for graph canonization, and its implementation, called `nauty` (which stands for *no automorphisms, yes?*), is described in (McKay 1990). In contrast to earlier works, where the canonical representation of a graph was typically defined to be the smallest graph isomorphic to it (in the lexicographic order), `nauty` introduced a notion which takes structural properties of the graph into account. The `nauty` tool has been applied in a wide range of applications which can be posed in terms of graph isomorphism, including graph drawing (Abelson *et al.* 2002), identifying symmetries for SAT-solving (Aloul *et al.* 2003), organic chemistry (Kouri *et al.* 2015), and many others. It has been reported to apply to huge input graphs, including the report in (Royle 2015) on how it applies to a 76 million vertex graph. For details on how `nauty` defines canonicity and for the inner workings of the `nauty` algorithm see (McKay 1981; McKay 1990; Hartke and Radcliffe 2009; McKay and Piperno 2014). In recent years `nauty` has gained a great deal of popularity and success. Other, similar tools, are bliss (Junttila and Kaski 2007) and saucy (Darga *et al.* 2004).

The `nauty` graph automorphism tool consists of two main components. (1) a C library, `nauty`, which may be linked to at runtime, that contains functions applicable to find the canonical labeling of a graph, and (2) a collection of applications, `gtools`, that implement an assortment of common tasks that `nauty` is typically applied to. When downloading the tool both components are included. During compilation static library files are created for the C library. These files may be linked to at runtime, and header files are provided which may be included in foreign C code. During compilation, the applications of `gtools` are compiled into a set of command line applications.

This paper presents a lightweight Prolog interface to both components of `nauty` which we term `pl-nauty` and `pl-gtools`. The implementation of `pl-nauty` is by direct use of Prolog's foreign language interface. The implementation of `pl-gtools` is slightly more complex. Each `gtools` application is run as a child process with the input and output controlled via Unix pipes. The `pl-gtools` framework provides a set of general predicates to support this type of application integration.

The integration of `nauty` into Prolog facilitates programming with the strengths of the two paradigms: logic programming for solving graph search problems on the one hand, and efficient pruning of (intermediate) solutions modulo graph isomorphism, on the other. It enables Prolog programs which address graph search problems to apply `nauty` natively, through Prolog, in the process of graph search and enumeration. Graphs may be generated non-deterministically and may be canonized deterministically. The same approach naturally for other search problems where the object of the search can be represented as a graph. For example, in (Codish *et al.* 2016), the authors prove that any data-oblivious sorting algorithm requires at least 25 comparisons for 9 inputs (an open problem since 1966). Symmetry breaks which can be expressed in terms of subgraph isomorphism are applied to reduce the search space for networks of 24 comparisons from 10^{37} networks to 10^{21} networks. This is a key step in the construction of a proof that no 24 comparison network exists.

The `pl-nauty`tool also facilitates the interaction with various graph representations: those used in `nauty`, and those more natural for use with Prolog. The interface for `nauty` from within Prolog combines well also with other tools and techniques typically applied when addressing graph search problems, such as constraint and SAT based programming. For example, recent work (Codish *et al.* 2016), presents a computer-based proof that the Ramsey number $R(4,3,3) = 30$, thus closing a long open problem concerning the value of $R(4,3,3)$. That paper made extensive use of SAT solvers, symmetry breaking techniques, and the `nauty` library. It was this experience that led us to implement `pl-nauty`. In this paper we adopt the search for Ramsey graphs as a running example. This problem has a very simple formal specification and we can provide concise Prolog + `nauty` code to demonstrate the way our tool can be applied.

The remaining sections of this paper are organized in the following manner: Section 2 introduces the definitions used throughout the paper, as well as the running example of Ramsey graphs. Section 3 introduces the core of the `pl-nauty` library by examples. Section 4 details the `pl-gtools` framework, and details the template used to integrate `gtools` applications with Prolog. Section 5 closes some technical loose ends, including details of supported platforms, package availability, and additional references to source code. Finally, Section 6 concludes.

2 Preliminaries

A graph $G = (V, E)$ consists of a set of vertices $V = [n] = \{\ 1, \ldots, n\ \}$ and a set of edges $E \subseteq V \times V$. In the examples presented in this paper graphs are always simple. Meaning that they are undirected, there are no self loops, and no multiple edges. The tools we present allow also directed graphs and support vertex-coloring.

Two graphs $G = ([n], E)$ and $G' = ([n], E')$ are said to be isomorphic if the vertices of one graph may be permuted to obtain the other. Namely, if there exists a permutation $\pi\colon [n] \to [n]$ such that $(u,v) \in E \iff (\pi(u),\pi(v)) \in E'$. Graph isomorphism is an equivalence relation. As such, it induces equivalence classes on any set of graphs, wherein graphs G, G' are in the same equivalence class if G and G' are isomorphic. The canonical representation of a graph G is some fixed value $can(G)$ such that for every graph G' isomorphic to G we have $can(G) = can(G')$.

The running example we use throughout this paper concerns the generation of Ramsey graphs: A $R(s,t;n)$ Ramsey graph, where $s,t,n \in \mathbb{N}$, is a graph G with n vertices such that G contains no clique of size s nor an independent set of size t. We denote by $\mathbb{R}(s,t;n)$ the set of all non-isomorphic Ramsey $R(s,t;n)$ graphs. The Ramsey number $R(s,t)$ is the smallest natural number n for which no $R(s,t;n)$ graph exist.

3 Interfacing Prolog with `nauty`'s C library

The `pl-nauty` interface is implemented using the foreign language interface of SWI-Prolog (Wielemaker *et al.* 2012). The `nauty` C library is linked with corresponding C code written for Prolog, which involves four low-level Prolog predicates: (1) `densenauty/8`, (2) `canonic_graph/6`, (3) `isomorphic_graphs/6`, and (4) `graph_convert/5`. The experienced `nauty` user will find `densenauty/8` to be a direct interface to the corresponding C function in `nauty`. The `canonic_graph/6` predicate performs graph canonization only. The `isomorphic_graphs/6` predicate tests two graphs for isomorphism, and `graph_convert/5` converts between the supported graph formats such as between the `graph6` (McKay) format often used in `nauty` and the Boolean adjacency matrices natural in logic programming. The `graph6` format is a format used by `nauty` for storing undirected graphs. Files in this format contain only printable ASCII characters, and have one line per graph. The format is not human readable, therefore in most cases it is only used to store graphs. The full details of the `graph6` format may be found in (McKay).

We present several examples of the `pl-nauty` library in Prolog. The first two examples revolve around enumerating Ramsey graphs modulo isomorphism. The rest are simple demonstrations of the core `pl-nauty` predicates in various cases. In the first example we apply a straightforward iterative approach to enumerate all solutions modulo isomorphism. The second example illustrates how `nauty` integrates into an existing tool-chain, all specified as part of the Prolog process. Here we first construct a constraint model, infused with a partial symmetry breaking predicate. Then, apply the finite domain constraint compiler BEE (Metodi and Codish 2012; Metodi *et al.* 2013), which stands for **B**en-Gurion University **E**qui-propagation **E**ncoder (written in Prolog), to obtain a CNF model, apply a SAT solver (through its Prolog interface), and then generate all solutions of constraint model. At the end of each iteration we apply predicates from the `pl-nauty` library to remove isomorphic solutions. The core of the code, with an emphasis on using the `pl-nauty` library is presented below. The complete code is available for download as part of the `pl-nauty` library, in the `examples` directory.

3.1 The First Example: Generate and Test

In the code below, the predicate genRamseyGT(S, T, N, Graphs) iterates starting from the empty graph to generate in Graphs, the set of all canonical Ramsey $(S, T; N)$ colorings. We represent graphs as Boolean adjacency matrices: a list of N length-N lists. At iteration I it takes Acc, the canonical set of Ramsey $(S, T; I)$ colorings computed thus far, and calls the predicate extendRamsey(S, T, I, Acc, NewAcc) to obtain, NewAcc, the canonical set of Ramsey $(S, T; I+1)$ colorings.

```
genRamseyGT(S, T, N, Graphs) :-
    genRamsey(0, S, T, N, [[]], Graphs).
genRamsey(I, S, T, N, Acc, Graphs) :-
    I < N, !, I1 is I+1, extendRamsey(S, T, I, Acc, NewAcc),
    genRamsey(I1, S, T, N, NewAcc, Graphs).
genRamsey(N, _, _, N, Graphs, Graphs).
```

The predicate extendRamsey(S, T, N, Graphs, NewGraphs) takes a list, Graphs of (canonical) Ramsey $(S, T; N)$ graphs. Then, a new vertex is added in all possible ways to each graph in Graphs and those new graphs that are Ramsey $(S, T; N+1)$ colorings are canonized. Finally, the resulting graphs are sorted to remove duplicates, resulting in NewGraphs. It is the call to canonic_graph/3 that interfaces to our Prolog integration of the nauty tool.

```
extendRamsey(S, T, N, Graphs, NewGraphs) :-
    N1 is N+1,
    findall(Canonic,
        (member(Graph, Graphs), addVertex(Graph, NewGraph),
         isRamsey(S,T,N1,NewGraph),              /* #1 (test)*/
         canonic_graph(N1, NewGraph, Canonic)),  /* #2 (reduce)*/
        GraphsTmp),
    sort(GraphsTmp, NewGraphs).
```

The predicate addVertex(Matrix,ExtendedMatrix) extends non-deterministically an adjacency Matrix with a new vertex by adding a new first row and equal first column.

```
addVertex(Matrix,[NewRow|NewRows]) :-
    NewRow = [0|Xs],
    addFirstCol(Matrix,Xs,NewRows).

addFirstCol([],[],[]).
addFirstCol([Row|Rows],[X|Xs],[[X|Row]|NewRows]) :-
    member(X,[0,1]),
    addFirstCol(Rows,Xs,NewRows).
```

To complete the example, we illustrate the test predicate isRamsey(S,T,N,Graph) which succeeds if the given Graph is a Ramsey $(S, T; N)$ coloring. This is so if it is not possible to choose S vertices from the graph, the edges between which are all "colored" 0, or T vertices from the graph, the edges between which are all "colored" 1.

```
isRamsey(S,T,N,Graph) :-
    forall( choose(N, S, Vs), mono(0, Vs, Graph) ),
    forall( choose(N, T, Vs), mono(1, Vs, Graph) ).

mono(Color, Vs, Graph) :-
    cliqueEdges(Vs,Graph,Es), maplist(==(Color), Es).

cliqueEdges([],_,[]).
cliqueEdges([I|Is],Graph,Es) :-
    cliqueEdges(I, Is, Graph, Es0),
    cliqueEdges(Is, Graph, Es).

cliqueEdges(_,[],_,[]).
cliqueEdges(I,[J|Js], Graph, [E|Es]) :-
     nth1(I, Graph, Gi),
     nth1(J, Gi, E),
     cliqueEdges(I,Js,Graph,Es).
```

```
choose(N,K,C) :-
    numlist(1,N,Ns),
    length(C,K),
    choose(C,Ns).

choose([],[]).
choose([I|Is],[I|Js]) :-
   choose(Is,Js).
choose(Is,[_|Js]) :-
    choose(Is,Js).
```

We first demonstrate the application of the genRamseyGT to the so called, "party problem". What is the smallest number of people that must be invited to a party so that at least three know each other, or at least three do not know each other. This is the smallest N for which there is no $(3, 3; N)$ coloring. The following two calls illustrate that there is a single canonical coloring when $N = 5$ and none when $N = 6$. So, the answer to the party problem (as well-known) is 6.

```
?- genRamseyGT(3,3,5,Gs).
Gs = [ [[0,1,1,0,0],
        [1,0,0,1,0],
        [1,0,0,0,1],
        [0,1,0,0,1],
        [0,0,1,1,0]]].
```

```
?- genRamseyGT(3,3,6,Gs).
Gs = [].
```

We make three observations regarding the generation of graphs in this example. Consider the predicate extendRamsey/5.

1. If the call canonic_graph(N1, NewGraph, Canonic), at the line marked /* #2 */, is replaced by the line Canonic = NewGraph, then all solutions are found, not just the canonical ones. For example, when $N = 5$ there are 12 solutions, all of them isomorphic.

   ```
   ?- genRamseyGT(3,3,5,Gs), length(Gs,M).
   M = 12.
   ```

2. If the call to isRamsey(S,T,N1,NewGraph), at the line marked /* #1 */, is removed then we generate all non-isomorphic graphs on N vertices.

Table 1. *Enumerating $R(3,5;n)$ graphs: Generate, Test & Reduce (time out: 1 hour).*

n	4	5	6	7	8	9	10	11	12	13	14
$\lvert\mathbb{R}(3,5;n)\rvert$	7	13	32	71	179	290	313	105	12	1	0
time (sec)	0.00	0.00	0.03	0.20	0.90	4.66	16.61	39.24	52.72	55.75	56.20
nauty (sec)	0.00	0.00	0.00	0.01	0.02	0.06	0.09	0.11	0.11	0.11	0.11
all Solutions	41	387	5617	113949	-	-	-	-	-	-	-
Prolog (sec)	0.00	0.05	1.50	76.09	-	-	-	-	-	-	-

For example,

```
?- genRamseyGT(3,3,5,Gs), length(Gs,M).
M = 34.
```

3. If both changes are made, then we generate all graphs on `N` vertices.

```
?- genRamseyGT(3,3,5,Gs), length(Gs,M).
M = 1024.
```

We now demonstrate the application of the `genRamseyGT` to generate incrementally all non-isomorphic $(3,5;N)$ Ramsey colorings. It is known (Radziszowski 1994) that $R(3,5) = 14$. Table (1) summarizes the enumeration of all non-isomorphic $(3,5;N)$ colorings graphs (for $4 \leqslant N \leqslant 14$). The first row indicates the number of (non-isomorphic) colorings generated. The next rows detail the time (in seconds) to compute these colorings and the time spent in the calls to `canonic_graph`. It is notable that the time spent to reduce solutions modulo isomorphism using `nauty` is negligible. The next row indicates the number of solutions found using Prolog without `nauty`. These are all solutions, not just the canonical ones. Here in the code for the predicate `extendRamsey/5` we omit the call to `canonic_graph(N1, NewGraph, Canonic)`, at the line marked `/* #2 */`, replacing it by the line `Canonic = NewGraph`. The last line in the table indicates the time required to compute the solutions in the previous line. It is clear that the number of isomorphic graphs generated quickly becomes overwhelming. Not to mention that if the goal is to compute the set of solutions modulo isomorphism, then this will have to be dealt with at the end anyway.

To summarize this section, we stress that this is a toy application with the intention to illustrate an application of the integration of Prolog with the `nauty` package. A more elaborate solution of this problem would, for example, combine the calls

```
addVertex(Graph, NewGraph), isRamsey(S,T,N1,NewGraph)
```

in `extendRamsey` to add edges connecting the new vertex to the rest of the graph incrementally so as not to violate the `isRamsey` condition. This combination could also perform various propagation based optimizations.

3.2 The Second Example: Constrain and Generate

In the code below, the predicate `genRamseyCG(S, T, N, Graphs)` encodes an instance `ramsey(S,T,N)` to a finite domain constraint model. We adopt BEE

(Metodi and Codish 2012; Metodi *et al.* 2013) for this purpose. The call to encode/3 generates a constraint model, Constraints and the $N \times N$ Matrix of Boolean (Prolog) variables. The Matrix structure serves as a mapping between the instance variables, which talks about the search for Ramsey colorings, and the Constraints variables. It specifies the connection between variables in the constraint model and edges in the unknown graph we are searching for. The call to bCompile/2 compiles the constraints to a corresponding CNF. The call to solveAll/3 iterates with the underlying SAT solver to provide all satisfying Assignments of the CNF (modulo the variables of interest in the list Booleans). Satisfying assignments are then decoded back to the world of graphs in the call to decode/3, and finally it is here that we call on the predicate canonic_graph/3 from the pl-nauty interface to restrict solutions to their canonical forms and remove isomorphic solutions by sorting these.

```
genRamseyCG(S, T, N, Graphs) :-
    encode(ramsey(S,T,N), Matrix, Constraints),
    bCompile(Constraints,CNF),
    projectVariables(Matrix, Booleans),
    solveAll(CNF,Booleans,Assignments),
    decode(Assignments,Matrix,Graphs0),
    maplist(canonic_graph(N), Graphs0, Graphs1),
    sort(Graphs1, Graphs).
```

The predicate encode/3 is presented below. It first creates an $N \times N$ adjacency Matrix with Boolean variables representing the object of the search for a Ramsey(S,T;N) graph. It then imposes three sets of constraints: (1) the call to lex_star/2 constrains the rows of Matrix to be pairwise lexicographically ordered. This implements the symmetry break described in (Codish *et al.* 2013); (2) the first call to no_clique/4 constrains the graph represented by Matrix to contain no independent set of size S, and (3) the second call to no_clique/4 constrains the graph represented by Matrix to contain no clique of size T. The full details of the example are available for download as part of the pl-nauty library, in the examples directory.

```
encode(ramsey(S,T,N), map(Matrix), Constraints) :-
        adj_matrix_create(N, Matrix),
        lex_star(Matrix, Cs1-Cs2),          /* #1 */
        no_clique(0, S, Matrix, Cs2-Cs3),   /* #2 */
        no_clique(1, T, Matrix, Cs3-Cs4),   /* #3 */
        Cs4 = [], Constraints = Cs1.
```

The following illustrates the BEE constraint model, with the associated adjacency matrix, produced by a call to the encode/3 predicate for a Ramsey $R(3,3;5)$ instance. Note that the elements on the diagonal of the matrix are -1 which is how *false* is represented in BEE. The constraint model consists of three types of constraints corresponding to the three annotated calls in encode/3.

Table 2. *Enumerating $R(3,5;n)$ graphs: Constrain, Generate & Reduce.*

n	4	5	6	7	8	9	10	11	12	13	14
$\lvert\mathbb{R}(3,5;n)\rvert$	7	13	32	71	179	290	313	105	12	1	0
$\#SAT$	7	18	63	255	1100	3912	7319	3806	272	2	0
time (sec)	0.00	0.00	0.00	0.02	0.02	0.12	0.74	1.97	1.16	1.15	0.07
nauty (sec)	0.00	0.00	0.00	0.00	0.01	0.05	0.16	0.05	0.00	0.00	0.00

```
Matrix = [[-1,A,B,C,D],
          [A,-1,E,F,G],
          [B,E,-1,H,I],
          [C,F,H,-1,J],
          [D,G,I,J,-1]]
```

```
% #1 pairwise lexicographical order
bool_arrays_lex([B,C,D],[E,F,G]),
bool_arrays_lex([A,B,D],[F,H,J]),
bool_arrays_lex([A,F,G],[B,H,I]),
bool_arrays_lex([A,E,F],[D,I,J]),
bool_arrays_lex([B,E,I],[C,F,J]),
bool_arrays_lex([C,F,H],[D,G,I]),
```

```
% #2 no independent set
bool_array_or([A,B,E]),
bool_array_or([A,C,F]),
bool_array_or([A,D,G]),
bool_array_or([B,C,H]),
bool_array_or([B,D,I]),
bool_array_or([C,D,J]),
bool_array_or([E,F,H]),
bool_array_or([E,G,I]),
bool_array_or([F,G,J]),
bool_array_or([H,I,J]),
```

```
% #3 no clique
bool_array_or([-A,-B,-E]),
bool_array_or([-A,-C,-F]),
bool_array_or([-A,-D,-G]),
bool_array_or([-B,-C,-H]),
bool_array_or([-B,-D,-I]),
bool_array_or([-C,-D,-J]),
bool_array_or([-E,-F,-H]),
bool_array_or([-E,-G,-I]),
bool_array_or([-F,-G,-J]),
bool_array_or([-H,-I,-J])
```

Table (2) summarizes the enumeration of all non-isomorphic $(3,5;N)$ colorings graphs using the constrain and generate approach (for $4 \leqslant N \leqslant 14$). The first row indicates the number of (non-isomorphic) colorings generated. The second row indicates the number of colorings found when solving the constraint model (with the partial symmetry break). The next rows detail the time (in seconds) to compute these colorings and the time spent in the calls to `canonic_graph`. It is notable that the time spent to reduce solutions modulo isomorphism using `nauty` is negligible.

3.3 The *graph_convert/5* predicate

The `graph_convert/5` predicate performs conversions between the different graph formats that are supported by `pl-nauty`. Supported formats include: adjacency matrices, adjacency lists, edge lists, and the `graph6` format. As an example, to convert a graph from the `graph6` format, to Prolog's adjacency matrix format:

```
?- Graph = 'DqK', graph_convert(5,graph6_atom,adj_matrix,Graph,Matrix).
Matrix = [[0,1,1,0,0],[1,0,0,1,0],[1,0,0,0,1],[0,1,0,0,1],[0,0,1,1,0]]
```

3.4 The *canonic_graph/6* predicate

The `canonic_graph/6` predicate performs graph canonization and it takes the form `canonic_graph(N, InputFmt, OutputFmt, Graph, Perm, Canonic)` where `InputFmt` is the format of the `N` vertex input graph (`Graph`), `OutputFmt` is the format of the canonical graph (`Canonic`), and `Perm` is the permutation whose application to the input graph renders the canonical representative. For example:

```
?- G = [[0,1,0,0,0], [1,0,1,0,1], [0,1,0,1,0], [0,0,1,0,1], [0,1,0,1,0]],
N = 5, canonic_graph(N,adj_matrix,adj_matrix,G,Perm,Canonic).

Canonic = [[0,0,0,0,1],[0,0,0,1,1],[0,0,0,1,1],[0,1,1,0,0],[1,1,1,0,0]],
Perm    = [1, 5, 2, 4, 3]
```

A compact version of canonic_graph/6 is also included in pl-nauty in the form of the predicate canonic_graph/3. The predicate canonic_graph/3 takes the form canonic_graph(NVert, Graph, Canonic) and it is equivalent to canonic_graph (NVert, adj_matrix, adj_matrix, Graph, _, Canonic). For example:

```
?- G = [[0,1,0,0,0], [1,0,1,0,1], [0,1,0,1,0], [0,0,1,0,1], [0,1,0,1,0]],
N = 5, canonic_graph(N,G,Canonic).

Canonic = [[0,0,0,0,1],[0,0,0,1,1],[0,0,0,1,1],[0,1,1,0,0],[1,1,1,0,0]]
```

3.5 The isomorphic_graphs/6 predicate

The isomorphic_graphs/6 predicate tests for graph isomorphism. It takes the form: isomorphic_graphs(N, Graph1, Graph2, Perm, Canonic, Opts) and tests if the two N vertex input graphs, Graph1 and Graph2 are isomorphic via a permutation Perm. If they are then Canonic is the canonical form they share. The predicate takes a list Opts of options to customize the behavior of this predicate. Options include any of the following: fmt1(Fmt1) the format of Graph1, fmt2(Fmt2) the format of Graph2, cgfmt(CgFmt) the format of Canonic. In the case where Graph1 and Graph2 are not isomorphic the predicate will fail silently. For example:

```
?- N = 5,
Graph1 = [[0,1,0,1,1],[1,0,1,0,0],[0,1,0,1,0],[1,0,1,0,1],[1,0,0,1,0]],
Graph2 = [[0,1,0,1,1],[1,0,1,0,0],[0,1,0,0,1],[1,0,0,0,1],[1,0,1,1,0]],
isomorphic_graphs(N, Graph1, Graph2, Perm, Canonic, []).

Perm    = [1,2,3,5,4],
Canonic = [[0,1,0,1,0],[1,0,0,0,1],[0,0,0,1,1],[1,0,1,0,1],[0,1,1,1,0]]

?- N = 5,
Graph1 = [[0,1,1,0,1],[1,0,0,0,1],[1,0,0,0,0],[0,0,0,0,0],[1,1,0,0,0]],
Graph2 = [[0,1,0,0,1],[1,0,1,1,0],[0,1,0,0,1],[0,1,0,0,1],[1,0,1,1,0]],
isomorphic_graphs(N, Graph1, Graph2, Perm, Canonic, []).

false.
```

3.6 The densenauty/8 predicate

Most of the core predicates of pl-nauty and many of the examples described above are based on the densenauty/8 predicate. The densenauty/8 predicate is a direct interface to the nauty C library function of the same name. The predicate is called in a similar fashion to its counterpart in the nauty C library. A complete documentation of densenauty/8 may be found in the source code provided with pl-nauty, and in the nauty user guide (McKay 2016).

Briefly, the predicate densenauty/8 takes the following form:

```
densenauty(NVert, Graph, Labeling, Partition,
           Permutation, Orbits, Canonic, Opts)
```

where NVert is the number of vertices in the input graph, Graph is the input graph, Labeling, Partition and Orbits are the labeling, partition and orbits of the input graph, as described in the nauty user guide (McKay 2016), Canonic is the canonical form of the input graph, and Permutation is the permutation of the nodes of the input graph which may be applied to obtain the Canonic representative. The last argument, Opts is used to modify the behavior of densenauty. For example, it may be used to control the format of the input graph, and Canonic representative.

4 Interfacing Prolog and gtools

The nauty graph automorphism tool comes with a collection of applications called gtools, that implement an assortment of common tasks that nauty is typically applied to. During installation (of nauty) these are compiled into a set of command line applications. These applications cannot simply be loaded using the foreign language interface. Each application is like a black box. We do not wish to access its source code. One straightforward approach to integrate gtools with Prolog is to run each such application from within Prolog, write its output to a temporary file, and then to read the file, and continue with the task that the Prolog program is addressing.

A more elegant solution makes use of Unix pipes to skip that intermediate step of writing and reading from files. The output is directly read/written via Prolog. The voodoo is using pipes (which are like in-memory files). We have implemented a Prolog library called pl-gtools, which provides a framework for calling the applications in gtools using Unix pipes. The pl-gtools framework supports two types of gtools applications which take any number of command line arguments and write their output to standard output. The first type does not require any input, and the second requires some form of input (from standard input). We present a general template to support the two "sides" of the pipe: a child predicate (which typically executes a gtools command), and a parent predicate (which typically reads the output of the child).

The framework includes predicates: gtools_exec/6 and gtools_fetch/2, and two additional predicates for applications which respectively require uni- and bi-directional communication: gtools_fork_exec/2 and gtools_fork_exec_bidi/2. For uni-directional communication, a call to gtools_fork_exec(Parent, Child) will fork and execute the Parent goal as the parent process and the Child goal as the child process. It assumes that both Parent and Child take an additional argument which is unified with the corresponding input/output streams (to support communication from child to parent). For bi-directional communication, a call to gtools_fork_exec_bidi(Parent, Child) is exactly the same, except that the Parent and Child take two additional arguments to support two way communication.

The predicate `gtools_fetch/2` reads the next line from the output stream of the child and converts it to an atom. When the end of the stream is reached, then the predicate fails. A call to `gtools_exec/6` takes the form `gtools_exec(NautyDir, Cmd, Args, InputStream, OutputStream, ErrorStream)` where: `NautyDir` is the directory in the file system which contains the `gtools` applications, `Cmd` is the name of the `gtools` command that we which to execute, and `Args` is its argument list. The final three arguments specify the standard input, output and error streams. The call to `gtools_exec/6` invokes the `exec/1` predicate of SWI-Prolog, replacing the current process image with `Cmd` and its `Args`.

We present two example uses of `pl-gtools`. The first, calls `geng` from `gtools`, which iterates over all non-isomorphic graphs with a given number of vertices. The second, calls `shortg` from `gtools`, which reduces a set of graphs to non-isomorphic members.

4.1 Example 1: `geng`

This example illustrates how the framework is applied for an application which reads no input. The `gtools` application `geng` receives an argument `n` and outputs one line for each non-isomorphic graph with n vertices. Its Prolog implementation consists of three predicates: `geng/2`, `parent_geng/2` and `child_geng/2`. The predicate `geng/2` is the main predicate which backtracks over all results of the `gtools` application. The predicates `parent_geng/2` and `child_geng/2` implement respectively the parent and child sides of the pipe.

```
geng(N, Graph) :-
    gtools_fork_exec(geng:parent_geng(Graph), geng:child_geng(N)).

parent_geng(Graph,Read) :-
    gtools_fetch(Read, Graph).

child_geng(N,Stream) :-
    gtools_exec('nauty26r3', geng, ['-q', N], _, Stream, _).
```

4.2 Example 2: `shortg`

This example illustrates how the framework is applied for an application which reads from standard input. The `shortg` application reads a list of graphs in the `graph6` format (McKay) from standard input, and removes all isomorphic duplicates, writing to standard output. It can be applied as follows:

After integrating `shortg` with `pl-gtools` it could be called from Prolog like so:

```
?- InputGraphs = ['DRo','Dbg','DdW','DLo','D[S','DpS','DYc','DqK','DMg',
                  'DkK','Dhc','DUW'], % a list of graphs in graph6 format
   shortg(InputGraphs, OutputGraphs). % the call to shortg

   OutputGraphs = ['DqK'].
```

The implementation of shortg in Prolog consists of three predicates and is very similar to that for geng except that communication between the child and parent processes is bi-directional.

```
shortg(In, Out) :-
     gtools_fork_exec_bidi(shortg:parent_shortg(In, Out),
                           shortg:child_shortg).

parent_shortg(In, Out, PRead, PWrite) :-
      maplist(writeln(PWrite), In),
      flush_output(PWrite),
      close(PWrite),
      findall(O, gtools_fetch(PRead, O), Out),
      close(PRead).

child_shortg(CRead, CWrite) :-
      gtools_exec('nauty26r3', shortg, ['-q'], CRead, CWrite, _).
```

In this example, shortg/2 takes two arguments: In a list of input graphs in the graph6 format, to be reduced modulo isomorphism, and Out will be unified with the set of reduced graphs. The predicate calls the gtools_fork_exec_bidi/2 predicate. Pipes are opened to set up two way communication between the parent and child.

Two additional predicates are implemented: one for the parent process and one for the child process. Each predicate takes, as its last two arguments the read and write ends of the pipes, so communication may be established. In our case, the parent writes the set of input graphs to the write end of the pipe, and then reads the results from the read end of the child's pipe. The child calls gtools_exec/6, and executes shortg/2.

5 Technical Details

A short overview of some technical details regarding pl-nauty and pl-gtools follows.

The package containing pl-nauty and pl-gtools is available for download from the pl-nauty homepage at: http://www.cs.bgu.ac.il/ frankm/plnauty. The package contains a README file, which contains usage and installation instructions, as well as an examples directory containing the examples discussed in this paper. The C code for pl-nauty may be found in the src directory. Also in the src directory are the two module files for pl-nauty and pl-gtools.

Both pl-nauty and pl-gtools were compiled and tested on Debian Linux and Ubuntu Linux using the 7.x.x branch of SWI-Prolog. It is important to mention that both pl-nauty and pl-gtools contain Linux specific features, and are oriented towards SWI-Prolog. It should also be noted that pl-nauty is not thread-safe, for reasons of performance. If you require a thread-safe version of pl-nauty you should synchronize calls to the predicates of the pl-nauty module.

It is important to point out that the purpose of this paper is to demonstrate the utility of the tool. We believe that portability will become a community effort, and should present a small challenge, once the paradigm of programming with Prolog + `nauty` catches on.

6 Conclusion

We have presented, and made available, a Prolog interface to the core components of the `nauty` graph-automorphism tool (McKay 1981) which is often cited as "The world's fastest isomorphism testing program" (see for example `http://www3.cs.stonybrook.edu/ algorith/implement/nauty/implement.shtml`). The contribution of the paper is in the utility of the tool which we expect to be widely used. The tool facilitates programming with the strengths of two paradigms: logic programming for solving graph search problems on the one hand, and efficient pruning of (intermediate) solutions modulo graph isomorphism, on the other. It enables Prolog programs which address graph search problems to apply `nauty` natively, through Prolog, in the process of graph search and enumeration. Graphs may be generated non-deterministically and may be canonized deterministically.

References

Abelson, D., Hong, S.-H. and Taylor, D. E. 2002. *A Group-Theoretic Method for Drawing Graphs Symmetrically*, Springer Berlin Heidelberg, Berlin, Heidelberg, 86–97.

Aloul, F. A., Ramani, A., Markov, I. L. and Sakallah, K. A. 2003. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Trans. on CAD of Integrated Circuits and Systems 22,* 9, 1117–1137.

Babai, L. 2015. Graph isomorphism in quasipolynomial time. *CoRR abs/1512.03547.*

Codish, M., Cruz-Filipe, L., Frank, M. and Schneider-Kamp, P. 2016. Sorting nine inputs requires twenty-five comparisons. *J. Comput. Syst. Sci. 82,* 3 (May), 551–563.

Codish, M., Frank, M., Itzhakov, A. and Miller, A. 2016. Computing the ramsey number r(4,3,3) using abstraction and symmetry breaking. *Constraints*, 1–19. Also In *Proceedings of the Thirteenth International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming.*

Codish, M., Miller, A., Prosser, P. and Stuckey, P. J. 2013. Breaking symmetries in graph representation. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China*, F. Rossi, Ed. IJCAI/AAAI.

Crawford, J., Ginsberg, M., Luks, E. and Roy, A. 1996. Symmetry-breaking predicates for search problems. Morgan Kaufmann, 148–159.

Darga, P. T., Liffiton, M. H., Sakallah, K. A. and Markov, I. L. 2004. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41st Design Automation Conference.*

Hartke, S. G. and Radcliffe, A. J. 2009. McKay's canonical graph labeling algorithm. *Contemporary Mathematics 479*, 99–111.

Junttila, T. and Kaski, P. 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*, SIAM, 135–149.

Kouri, T. M., Pascua, D. and Mehta, D. P. 2015. Random models and analyses for chemical graphs. *Int. J. Found. Comput. Sci. 26,* 2, 269.

McKay, Brendan D. and Piperno, A. 2016. *nauty and Traces Users Guide (Version 2.6)*.

McKay, B. *The graph6 format*. (web pages describing the format).

McKay, B. 1990. *nauty* user's guide (version 1.5). Tech. Rep. TR-CS-90-02, Australian National University, Computer Science Department.

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

McKay, B. D. and Piperno, A. 2014. Practical graph isomorphism, II. *Journal of Symbolic Computation 60*, 94–112.

Metodi, A. and Codish, M. 2012. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming 12,* 4-5, 465–483.

Metodi, A., Codish, M., and Stuckey, P. J. 2013. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR) 46*, 303–341.

Radziszowski, S. P. 1994. Small Ramsey numbers. *Electronic Journal of Combinatorics*. Revision #14: January, 2014.

Royle, G. 2015. Spectacular graph automorphisms.
`https://symomega.wordpress.com/2015/06/26/spectacular-graph-automorphisms/`. Blog. Viewed July 2016.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.