# Graph Virtual Machine

**Nicholas Richardson**

During my dissertation, I worked with a program called **n**o **aut**omorphism **y**es(nauty). The program nauty uses an array of integers to store a graph as an adjacency matrix. Brendan McKay, who wrote nauty, also has a fast algorithm to create a canonical labeling of a graph for isomorphism testing. Using this, nauty can also generate all graphs up to isomorphism in a fast and efficient manner.

Many mathematicians understand the basics of programming, but do not think like programmers. For example, one way to define the series of Fibonacci numbers is F(1) = 0 and F(2) = 1 as a base case. Then use the general form F(N) = F(N-1) + F(N-2). This is very natural to a mathematician, but it is a terrible way to code. To use nauty, the user must have a good understanding of C/C++ and then be able to jump into a large project and understand how the project is setup. Depending on the project, this can be difficult to do even for experienced programmers. The proposed project attempts to create a layer between low level C/C++ programming and High Level mathematical proofs.

The Graph Virtual Machine (GVM) is an extension to nauty that provides an abstraction layer between programming and mathematical logic. The virtual machine will be able to control nauty functions while the user will write a simple program in the GVM language that will be similar to a mathematical proof. The GVM language will reduce the learning curve while still taking advantage of nauty's functions.

An attempt is made to keep all operators in the GVM as close to regular Graph Theory notation as possible. To make entering commands easier, the GVM should only use characters and symbols from a keyboard as input, so some operators must be changed. These changes will attempt to have some relation to the meaning of the original notation. For example, there is no less than or equal sign on a keyboard, so a new symbol must be used. The less than sign and the equal sign will be adjacent, <=, to form the less than or equal symbol. This symbol retains the original notation's meaning and is easy to type in with a keyboard.

In the future, the GVM will be able to automatically parallelize the graph theory code based on the computer or distributed network using MPI and CUDA. The two main types of parallel implementations will be specialized loops (there exists and for all) and nauty generators with residue classes. The parallelization option can be limitedly set from GVM language. For now, the GVM will concentrate on creating a robust language that can describe the functions that are commonly used in graph theory. The goal is to have one GVM program to run on LONI that will read an input file for its instructions. Small changes can be easily made in the GVM language without having to recompile nauty.

In nauty, graphs are stored as an array if integers. The number MAXN represents the size of an integer (in bits) and the maximum number of vertices that a graph can have to be stored in nauty. Currently nauty uses MAXN = 32, but this may be adjusted for 64-bit computers.

MAXN is a constant in the GVM and cannot be changed through the GVM language, the GVM program must be recompiled for any changes to take effect. For now, graphs will have 32 as the maximum number of vertices. If a 64-bit version can be reliably tested, the GVM will be upgraded for a maximum of 64 vertices. In the future, to enhance speed a method will be developed to extend the GVM to use functions in C/C++ that can be called from the GVM language. This method may include linking external libraries or involve having to recompile the GVM.

## Overview:

| | |
|---|---|
| **Part 1** | Graph Theory and Sets |
| Graph Theory Data Types | Specific data types used for graphs in the GVM language. |
| Sets | Set theory specific notation and operators. |
| Graph Theory Operators | Operators in the GVM for combining and modifying graphs. |
| Graph Theory Problems | Problems that must be figured out for the GVM |
| Specialized Loops and Conditions | For all and there exist statements specifically in the GVM |
| Automated Graph Drawing | An automated graph drawing program, |
| Example Programs | Sample program of the GVM language. |
| **Part 2** | Compiler Design **(SKIP THIS PART)** |
| Programming | Methods used to modify a graph in the GVM specific to making the language work. |
| List of All Operators | List of all operators that are needed in the language |
| List of Keywords | All keywords in the GVM. |
| **References** | |

## Graph Theory Data Types:

The base types in the GVM for graph theory are vertex, edge, graph, and class. A vertex contains an integer value for labelling. An edge consists of a pair of vertices, a head and tail. The order of vertices in an edge only matter in a directed graph. A graph consists of a vertex set and an edge set. A class is a set of graphs.

Several types of graphs can exist and have different properties. The most basic type of graph is a simple graph. A simple graph has no loops and no multiple edges. The next type is a directed graph. Edges in a directed graph go from one vertex to the next, so the order of vertices in an edge matter. The edge from u to v, or uv, and the edge from v to u, or vu, can be both stored in nauty's data structure as distinct edges. Both simple and directed graphs can be stored directly into the nauty data structure without any trouble. The GVM will concentrate on using simple and directed graphs for now.

In the future, the GVM can be extended to multigraphs and networks. A multigraph is a graph that can have multiple edges. The GVM will store a multigraph as a simple or directed graph, but will also track of the number of each possible edge in an array at runtime. An integer array of all possible edges can be stored to count how many edges exist between any two vertices in the graph. This requires a memory overhead and should only be used when needed. A network can have edge weights as an integer, and source/sink vertices. The extra information will also be tracked at runtime by the GVM.

Nauty has generators that can generate all graphs up to isomorphism. The goal of this project is to easily access the graphs from these generators. A specialized structure in GVM language will be able to set the same parameters to directly interface with nauty. The following is an example of geng, this generator will create all graphs up to isomorphism. Every graph generated will call the function OUTPROC in the GVM. The GVM will use this method to call a specified function in the GVM code with each graph generated. These functions should take a graph as an input and should return a Boolean expression as TRUE/FALSE. All graphs that are returned true will be added to a class that is returned when generation is completed. An option will be available to only count the graphs and return the total number instead of the class. This should be used when the number of graphs in the class will be very large. Other generators exist in nauty for specific graphs and will be included in the future.

nauty-geng will use this type to access the following

| GVM | geng command | description |
|---|---|---|
| | n | number of vertices 1..32, |
| | a:b | number of edges |
| | -d#D# | minimum and maximum degree |
| | -cC | 1 or 2 connected |
| | -tf | triangle/four cycle free |
| | -b | bipartite only |
| | res/mod | class(generate disjoint subsets of all graphs in parallel) |

## Sets:

| Operator | Name | Description |
|---|---|---|
| /\ | AND | Logical and, intersection |
| \/ | OR | Logical or, union |
| ! | NOT | Logical not |
| ^ | XOR | Exclusive or |
| {} | Set | Use { } to define elements in a set. Separate with a , |
| \|\| | Order | Size of the set |
| in | | An element of |
| notin | | Not an element of |
| TE, THERE_EXISTS | | There exists and element in a set |
| FA, FOR_ALL | | For all elements in a set |

A set is a collection of objects. In the GVM, three types of sets exist. The first is a class, which is defined as a set of graphs. The remaining two are the Vertex set and Edge set. These sets may be added to or removed from a graph with the Graph theory operators. The union and intersection of sets will be implemented. A class will have no complement, since it will have infinitely many graphs. The universe for the vertex set and edge set will be K_32 including loops for graphs that are not simple. The edge set will be a set of ordered pairs, but the order only matters in a directed graph.

## Graph Theory Operators:

Note that many of these operations are dependent on the current labeling of the graph. To do operations on graphs without labeling, the operations must be done on all possible graph labelings. To reduce the computational burden, nauty's canonical can be used, however this will not work for every case. The canonical labeling will order the vertices in such a manner that two isomorphic graphs will have the same labels. Finding a subgraph of a graph will have problems due to a graph labelling. The subgraph must search through all possible labelings. This will only work for small graphs, but another algorithm may be more efficient to use.

Unary Operators

| Operator | Right/Left Operand | Operand | Type Created (new) | Example | Operation |
|---|---|---|---|---|---|
| ‘ | Left | Graph | Graph | G’ | Edge Complement of G |
| # | Left | Graph | Graph | G# | Canonically label G (nauty) |
| & | Right | Vertex | Int | &G[v] | Degree of vertex |
| & | Right | Edge | Real | &e | Weight of edge |
| @ | Right | Graph | Graph | @G | Display G6 format (nauty) |
| ? | Right | Graph | Graph | ?G | Display Adjacency Matrix |
| $ | Right | Graph | Graph | $G | Display Edge list |

The operators @, ?, and $ are used for formatting the output of graphs at runtime. These should only be used when using the PRINT statement or WRITE statement for file I/O. The default output will be G6 format which uses printable characters to represent a graph. It is a common method to transfer a number of graphs that can then be read by another nauty program.

= Assignment Operator

| Left Operand | Right Operand | |
|---|---|---|
| (overwritten) | | |
| Class | Class | |
| Class | Graph | Class with 1 graph |
| Graph | Graph | |
| Graph | Vertex | Graph with 1 vertex |
| Graph | Edge | Graph with 1 edge |
| Edge | Edge | |
| Vertex | Vertex | |
| Vertex | Int | Vertex label |

The Assignment operator (=) is used to assign variables data to store.. The data type on the right side will be stored in the location on the left side. If casting is needed, the appropriate up-casting will occur.  The defined casting is also shown.  An integer can become a vertex.  A vertex or edge will become a graph.  A graph will become a class.  These casts are done when needed.

| | | | |
|---|---|---|---|
| Class | Graph | Vertex | Int |
| | | Edge | |

$\leftarrow \quad \leftarrow \quad \leftarrow$

These are binary operators.  These are based directly on the current labels of each vertex unless otherwise specified.  The letters C and D represent classes, G and H are graphs, e and f are edges, and u and v are vertices.  Remember that nauty uses labels and the new graph may have different labelings.  For certain operations, it will be possible to keep the labeling on part of the graph.   Using the disjoint union and join operators, the graph on the left side will have its labeling kept, while the right side will be relabeled. When contracting two edges, the lower vertex label will be kept.

**All operations are with current labelings**

| **Operator** | **LHS** | **RHS** | **Type Created** | **Example** | **Operation** |
|---|---|---|---|---|---|
| + | Class | Class | Class | C + D | Union of two classes |
| + | Class | Graph | Class | C + G | Union G with class C |
| + | Graph | Graph | Graph | G + H | Union of G and H |
| + | Graph | Vertex | Graph | G + v | Union v and G |
| + | Graph | Edge | Graph | G + e | Union G and e |
| + | Graph | VSET | Graph | G + {v} | Graph union with set of vertices |
| + | Graph | ESET | Graph | G + {e} | Graph union with set of edges |
| + | Vertex | Vertex | Graph | u + v | Union two vertices |
| + | Vertex | Edge | Graph | v + e | Union edge and vertex |
| + | Edge | Edge | Graph | e + f | Union two edges |
| <> | Graph | Graph | Graph | G <> H | Disjoint union of G and H |
| >< | Graph | Graph | Graph | G >< H | Join graphs G and H |
| >< | Graph | Vertex | Graph | G >< v | Join vertex v to G |
| >< | Edge | Vertex | Graph | e >< v | Join v to edge e, C3 |
| >< | Edge | Edge | Graph | e >< f | Join 2 edges, K4 |
| * | Vertex | Vertex | Edge | u * v | Create edge uv |
| * | Graph | Graph | Graph | G * H | Intersection of G and H |
| - | Class | Class | Class | C - D | Remove graphs in D from C |
| - | Class | Graph | Class | C - G | Remove G from C |
| - | Graph | Graph | Graph | G - H | Delete vertices of H from G |
| - | Graph | Vertex | Graph | G - v | Delete v from G |
| - | Graph | VSET | Graph | G - {v} | Delete set of vertices from G |
| \ | Graph | ESET | Graph | G \ {e} | Delete set of edges from G |
| \ | Graph | Graph | Graph | G \ H | Delete edges of H from G |
| \ | Graph | Edge | Graph | G \ e | Delete e from G |
| / | Graph | Edge | Graph | G / e | Contract edge e in G |

| | | | | | |
|---|---|---|---|---|---|
| [] | Graph | Graph | Graph | G[H] | The subgraph of G induced by H |
| [] | Class | Int | Graph | C[i] | The ith graph in class C |
| [] | Graph | Vertex | Vertex | G[v] | Access vertex v in G |
| [] | Graph | Edge | Edge | G[e] | Access edge e in G |
| [] | Graph | Int | Vertex | G[i] | Access vertex label i |

The program nauty uses a labeling system to keep track of vertices. In Graph Theory, many times vertex labels do not matter. This must be taken into account when creating the GVM. For example, take the union of two graphs. Using the + operator, G + H, the union will have vertex labels of both graphs. But using the <> operator, G <> H, will form a disjoint union. The new graph will have completely new labels than G and H. Joining two graphs, G >< H, will form a disjoint union and add all possible edges between every vertex in G to every vertex in H. The other operators may also be dependent on the current labeling of graphs.

Comparison Operators. These operators use a direct label comparison of two graphs and will return a TRUE or FALSE. It is possible for two graphs to be isomorphic, but not equal (==) since they may have different labels. The isomorphic comparison is equivalent to the equality when comparing two canonically labeled graphs (G =~ H is the same as #G == #H).

| **LHS** | **Operator** | **RHS** | **Description** |
|---|---|---|---|
| G | < | H | G is a proper subgraph of H |
| G | <= | H | G is a subgraph of H |
| G | > | H | H is a proper subgraph of G |
| G | >= | H | H is a subgraph of G |
| G | == | H | G is equal to H |
| G | != | H | G is not equal to H |
| e | in | G | e is in G |
| v | in | G | v is in G |
| e | notin | G | e is not in G |
| v | notin | G | v is not in G |
| G | =~ | H | G is isomorphic to H |
| G | ~ | H | G is isomorphic to H |
| | FORALL | | See Specialized loops and conditions |
| | THERE_EXISTS | | See Specialized loops and conditions |

The GVM may extend to subgraph comparisons without labels in the future. The comparisons will compare the subgraph over all possible labelings. These are resource intensive and should be avoided for anything other than small subgraphs.

| **LHS** | **Operator** | **RHS** | **Description** |
|---|---|---|---|
| G | << | H | G is a proper subgraph of H |
| G | <<= | H | G is a subgraph of H |
| G | >> | H | H is a proper subgraph of G |
| G | >>= | H | H is a subgraph of G |

Order of Operations for Graph Operators. The precedence of operators are still being determined, but this is currently proposed. The table list the precedence high to low

| | **Operator** | **LR or RL** | |
|---|---|---|---|
| HIGH | [] ( ) { } | Left to Right | Paired |
| | * /\ | Left to Right | Intersection, and |
| | \ / | Left to Right | Edge removal from graph |
| | <> >< + - ^ \/ | Left to Right | Union, or, xor, vertex removal |
| | ! & # ‘ | Left to Right | Graph properties, not |
| | @ ? $ \|\| | Left to Right | Print format, order |
| | < <= > >= == != =~ ~ | Left to Right | Comparison |
| LOW | = in notin | Right to Left | Assignment, element, not element |

## Specialized Loops and Conditions:

Two mathematical statements used commonly with sets and graphs are There Exists and For All. These will be in the GVM language as a specialized loop. By stating there exists an edge/vertex in G, parallel loops will run over the graph G for every edge (or vertex). Inside the statements, a copy of G and e (or v) to use in the algorithm. A true or false is expected depending on the condition asked. For There Exists specialized loop, the first true found will kill all processes and return true for the condition. If no edge/vertex is found, false is returned. After the conclusion of there exists, the graph G will be the original graph. Similarly, the for all loop will stop with the first false returned value.

This is an example of a for all and there exists specialized loops. This may also be done searching for an edge in G or searching for a graph in a particular class. For now, only one variable can be used. To find For all u,v in G, nested loops should be created.

```
FORALL( Vertex v in G )
{
        // test every vertex in graph G
        // return a true or false based on algorithm
        // assume TRUE if nothing is returned.
}
FA_TRUE
{
        // true is returned for every vertex in G
}
FA_FALSE
{
        // false was returned for at least one vertex in G
};
```

```
THERE_EXISTS( Vertex v in G )
{
        // test every vertex in graph G
        // return a true or false based on algorithm
        // assume FALSE if nothing is returned
}
TE_TRUE
{
        // true is returned for at least one vertex in G
}
TE_FALSE
{
        // false was returned for every vertex in G
};
```

## Automated Graph Drawing:

The program himmeli is a program that can take an edges list of a graph and create a image file. In the past, I used this program to take a list of graphs found through nauty to create a latex file that will display an image of each graph along with certain properties. The GVM may extend to creating an automated pdf list of graphs in the future. I found this a useful feature when dealing with hundreds of graphs. This may need a different program, but the GVM will be able to create a file in the required format to output any class or list of graphs.

## Example Programs:

See the attached .txt file for a sample program. The program extension for the GVM will be *.gvm. This is still being developed, so any input is welcome. Example is incomplete. Still trying to nail down exactly how to have the code be. That is the purpose of this document. Any input on how you would like the code to be will be appreciated.

- Keywords
- Operators
- Control Structures
- General coding style
- Etc.
- You can send me some example code if you want.

The program star.gvm creates a star graph. After the graph is created, the complement is taken. Since the graph only had 5 vertices, the complement is only on those 5 vertices, not all vertices for MAXN. The program kn_output.gvm shows functions and different output methods, including file I/O. These are samples and will be extended. The file xlm.gvm shows an algorithm for finding excluded minors for the plane. It includes functions in a separate file and shows how to use the generator and FOR_ALL statements.

**THIS DOCUMENT IS FAR FROM COMPLETE. IT WILL BE UPDATED AS THE PROJECT DEVELOPS.**

## Programming:

### This is compiler stuff, skip if you want.

The GVM language has the standard control structures for programming. Conditional statement can be made with IF THEN ELSE. There will be for, while and do/while loops. The addition of specialized loops and functions was mentioned in the previous section FORALL and THEREEXISTS. The program can have functions in a function block. These functions may be assigned to a generator option to be called for every graph that gets generated.

## List of All Operators:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | - | / | \ | * | = | ( ) | [ ] |
| != | @ | # | $ | % | ; | { } | ‘ |
| ~ | =~ | == | <= | < | >= | > | , |
| . | ? | >>= | <<= | <> | >< | \| | & |
| \/ | /\ | >> | << | ** | \|\| | : | ^ |
| ! | -> | <- | :: | := | | | |

## List of Keywords:

Programming Keywords

| **Keyword** | **Description** |
|---|---|
| START | |
| END | |
| IF | Conditional statement. Else is optional |
| THEN | IF bool_expr THEN statements ENDIF |
| ELSE (opt) | Or |
| ENDIF | IF bool_expr THEN statements ELSE statements ENDIF |
| FOR | FOR I = 1 TO 100 STEP 5 |
| TO | Statements |
| STEP (opt) | NEXT |
| NEXT | The step is optional |
| PRINT | Display output to screen |
| WHILE | |
| WEND | |
| DO | |
| WHILE (or UNTIL) | |
| LET | Declare variable |
| | |
| NEW | |
| OPEN | File I/O |
| CLOSE | |
| RECORD | |

| | |
|---|---|
| WRITE | |
| | |
| MOD | |
| INFINITY | |
| NULL | |
| | |
| RETURN | Return a data type |
| TRUE | Logical true |
| FALSE | Logical false |
| FUNCTION | For managing functions |
| INCLUDE | Include files from somewhere else |
| | |
| INT | Integer |
| REAL | Real number |
| STRING | String of characters |
| BOOL | Boolean value |

Graph Theory Keywords

| **Keyword** | **Description** |
|---|---|
| CLASS | Class of graphs |
| GRAPH | Graph |
| Simple- | |
| Directed- | |
| Multi- | |
| Network- | |
| VERTEX | Vertex |
| EDGE | edge |
| GENG | Geng generator properties |
| GENERATE | Use nauty generator |
| | |
| V(G) | Vertex set of G |
| E(G) | Edge set of G |
| NBD(G[v]) | Neighborhood subgraph of v in G |

Set Theory Keywords

| **Keyword** | **Description** |
|---|---|
| ESET | Set of edges |
| VSET | Set of vertices |
| AND | Logical and |
| OR | Logical or |
| XOR | Exclusive or |
| NOT | Logical not |

| | |
|---|---|
| FA | For all |
| TE | There exists |
| IN | An element of |
| NOTIN | Not an element of |
| NULL | Empty set |

## References

B. D. McKay and A. Piperno, Practical Graph Isomorphism, II, *J. Symbolic Computation* (2013) **60** 94-112. http://dx.doi.org/10.1016/j.jsc.2013.09.003.

Gross, Jonathan; Yellen, Jay, *Graph Theory And Its Applications*, CRC Press, 1999.

McKay, B. D. *nauty User Guide (Version 2.4),* Available at http://cs.anu.edu.au/~bdm/nauty/

Read, Ronald C., A Graph Theoretic Programming Language, *Graph Theory And Computing*, Academic Press, New York, 1972.

Richardson, Nicholas, *A Characterization Of Ramsey Graphs For R(3,4),* 2011.

West, D., *Introduction To Graph Theory 2nd Edition*, Prentice-Hall, New Jersey, 2001.