Abstract

The program package orbiter can be used to classify discrete structures like graphs, codes, designs, and objects from finite geometry. We describe how to use this package and how the package works.

1 Credo

Many Computer Algebra systems are currently available. Some systems are very good for integration and differentiation problems and many can do symbolic computations such as the algebraic manipulations of expressions involving polynomials and transcendental functions. Some systems are very good at group theory. This leaves the question why anyone would care developing yet another system.

Let us have a look at Combinatorics. This is an area of Mathematics whose boundaries are not clearly defined. In fact, even the question what Combinatorics is is probably not easy to answer. Many combinatorialists are concerned with discrete structures (graphs, trees, designs, codes, and many other objects). In Analytic Combinatorics, one wishes to find out properties of these structures by algebraic or probabilistic means. An algebraic means would be for instance a generating function. A probabilistic method would look at properties of random graphs, for instance (of course, this is a vastly simplified point of view). However, there are combinatorial structures that defy any attack from these fronts. Typically, these structures are defined by conditions that cannot be formalized in the world of tools that we just mentioned.

Here is an example: While it is possible to count the number of linear codes of length $n$ and dimension $k$ over the field $\mathbb{F}_q$, no such count is known for the linear codes of length $n$ and dimension $k$ over the field $\mathbb{F}_q$ with minimum distance $d$. So, while counting vector subspaces of a given dimension $k$ under the action of the symmetry group of the Hamming space is possible, the additional condition on the minimum distance seems to evade any attempts of being formalized using these techniques. At present, the only way to approach these structures is by generating them exhaustively. Compared to the methods of Analytic Combinatorics, this is a vastly unelegant procedure. In particular, it touches upon the
question of P versus NP of Theoretical Computer Science, since we have to decide the isomorphism question for these structures. Namely, while making a list of representatives of structures, we have to decide if a potentially new object is already contained in the list of representatives that we have established before. This is among the hardest known problems in Computer Science, and we have to decide this problem not just a few times but rather all the time.

What can be done? First of all, the problem outlined above can be restated as the problem of finding the orbits of a group on a set. For instance, in the case of codes with minimum distance \( d \) we act with the symmetry group of the Hamming space on the subspaces of dimension \( k \) of an \( n \)-dimensional vector space over \( \mathbb{F}_q \) such that the subspace has minimum distance \( d \). This is potentially a gigantic problem. There are a great number of subspaces, and computing the minimum distance of a subspace is also not easy. It is conceivable that it is even impossible to store in memory the set of such subspaces, so it is even more contrived to perform orbit calculations on such large sets.

Before we go on, let us quote the combinatorialist Peter Cameron:

\[
\text{Just because a problem is hard does not mean we should not solve it.}
\]

So, if we nonetheless decide to classify these structures (which often is of great interest for applications), we are looking for real algorithms and real implementations that work efficiently. We certainly cannot expect any kind of polynomial time algorithm is the putput size is potentially exponential in the input parameters (if we take the trivial group acting on the set of subsets, for instance). However, the point is that we strive to find algorithms that work reasonably well for small cases. So, what is it that we want?

Certainly, we would want to beat the naive approach of enumeration by stepping through the whole set of objects and performing some kind of orbit algorithm as outlined above. First of all, it is prohibitive to have all objects in memory. Worse, it is even prohibitive to run through all the objects once. We need to find a set of representatives of the orbits without even looking at the whole orbits.

What we wish to do is to “classify” the orbits without enumerating the whole set. Classify here means that we choose representatives out of each orbit. In the wider sense we also want the associated stabilizer subgroups, and we want to be able to map an arbitrary object to one of the representatives that we have chosen. All these algorithms are available in \texttt{orbiter}. In fact, they are available in a sort of template form, and they can be applied to all kinds of objects once the object can be encoded using either sets or subspaces.

The algorithms in \texttt{orbiter} are mostly drawn from group theory and combinatorics, and it would certainly be possible to write these algorithms in existing computer algebra packages. The drawback is that existing packages are either not efficient enough or not freely available. For this reason, we decided to implement these algorithms from scratch anew, knowing that
this would require us to start again from a very low level and duplicate algorithms that are available in some form elsewhere. Nonetheless, the implementation in orbiter leads to some of the fastest known algorithms to classify discrete structures, and so it could be said that in this case the end justifies the means.

By putting these algorithms in a library of C++ functions, we have hope that they can be reused by other people (as we did reuse other people’s work in developing orbiter, as can be seen from the acknowledgements in Section 8). In order for other people to reuse these algorithms, we do the following things:

1. Release the package under the free software licence GNU-GPL. You can use the software for your own (non-commercial) projects.
2. Make available documentation. This is what the present document is about.

A few words about the history of this project. The earliest roots of this project date back to the beginning 90’s. Some of the algorithms have been re-coded multiple times. In the late 90’s, the project DISCRETA was established. This was redesigned in the early 2000’s, and orbiter is the final product in this chain of developments. The biggest change between DISCRETA and orbiter is the way groups are represented. In DISCRETA, group elements were stored as lists, namely the list of images of the points in the permutation action. In orbiter, this was replaced by the concept of a group action. We use polymorphism to allow for different types of groups to be represented via a uniform action interface. A group action is an interface to a group with a specified action. The action is independent from the representation of the group elements. The representation is decided elsewhere. For instance, when dealing with matrix groups over finite fields, the group elements are represented as matrices. The action provides the functionality to evaluate images by performing the necessary matrix-vector multiplication.

As pointed out before, the project duplicates some of the work done elsewhere. For instance, the data type for longintegers in orbiter is probably not extremely efficient. Much better packages are available. Note that when this project was started, not many of these other packages were readily available. Now that we have a longinteger class that works reasonably well, we decide to stick with it. In our experience, relying on a great number of external non-standard libraries makes a package difficult to install and use (in particular for the unexperienced, and many of the mathematicians that might want to work with orbiter might fall into this category). For this reason, we prefer to leave things the way they are. It is certainly possible to write code using orbiter and using a different library for longinteger arithmetic on top of this.
2 Introduction and Installation

Orbiter is a program package written in C++ to classify discrete structures. It is available from the web-site

http://www.math.colostate.edu/~betten/orbiter.html

The package is distributed in a file called orbiter_yymmdd.tar.gz (where yy stands for the year, mm for the month and dd for the day of the release). After unpacking, the following directory structure is established:

    ORBITER/
    ORBITER/DATA
    ORBITER/DATA/BLT
    ORBITER/DATA/BLT/23
    ORBITER/DATA/GRAPH_AND_TOURNAMENT
    ORBITER/DATA/HYPEROVAL
    ORBITER/SRC
    ORBITER/SRC/BLT
    ORBITER/SRC/GRAPH
    ORBITER/SRC/HYPEROVAL
    ORBITER/SRC/LIB
    ORBITER/SRC/LIB/ACTION
    ORBITER/SRC/LIB/DISCRETA
    ORBITER/SRC/LIB/GALOIS
    ORBITER/SRC/LIB/INCIDENCE
    ORBITER/SRC/LIB/SNAKES_AND_LADDERS
    ORBITER/SRC/LIB/TOP_LEVEL
    ORBITER/SRC/LIB/WINDOWS
    ORBITER/SRC/LIB/WINDOWS/SAVE

To compile, execute

    cd ORBITER
    cd SRC
    make
    cd ../..

The core of the system is a library of C++ classes that is contained in ORBITER/SRC/LIB. Several application programs are part of orbiter. They are stored in subdirectories of ORBITER/SRC such as

    ORBITER/SRC/BLT
    ORBITER/SRC/GRAPH
The library itself comes in six parts that from a hierarchy of layers of smaller libraries of classes. The layers in increasing order are

```
ORBITER/SRC/LIB/GALOIS
ORBITER/SRC/LIB/ACTION
ORBITER/SRC/LIB/SNAKES_AND_LADDERS
ORBITER/SRC/LIB/INCIDENCE
ORBITER/SRC/LIB/DISCRETA
ORBITER/SRC/LIB/TOP_LEVEL
```

Each library comes with its own declaration file (.h file). The declaration files are

```
ORBITER/SRC/LIB/GALOIS/galois.h
ORBITER/SRC/LIB/ACTION/action.h
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/snakesandladders.h
ORBITER/SRC/LIB/INCIDENCE/incidence.h
ORBITER/SRC/LIB/DISCRETA/discreta.h
ORBITER/SRC/LIB/TOP_LEVEL/top_level.h
```

Instead of including all six files, an application can just include the file

```
ORBITER/SRC/LIB/orbiter.h
```

This file will include all six include files in turn. By using makefiles to set compiler options that list the search path, the include command in an application would simply be

```
#include "orbiter.h"
```

The idea behind orbiter is that code that is of general purpose should be part of the library. This has the slight complication that classes in the library do not know about classes that are defined in the application itself. So, for instance when orbiter is supposed to compute orbits of groups acting on certain classes of objects, the classes responsible for computing the orbits do not know what kind of objects they deal with. In order to make this work, we require that an object is defined as a subset of a certain basic set. For instance, in many cases the object is defined as a subset itself. In other cases, the object may be a vector space that is spanned by a basis, and the basis is a subset of the basic set, which in turn is the set of vectors of a finite vector space. In the latter case, the problem arises that a vector space has several bases. In this case, orbiter's library knows about equivalent bases, and chooses the lexicographically least basis to represent the subspace.
When working with symmetry groups, orbiter provides an interface to handle group operations like multiplication or inversion of group elements. It also provides access to the group action. So, for instance, given a point and a group element, one can compute the image of this point under the group element. When working through this interface, one never needs to know the kind of group that is considered. This way, an algorithm for computing orbits can be implemented that does not depend on a specific group. This is the basic idea behind the libraries ACTION and SNAKES_AND_LADDERS. An interface for permutation group algorithms is provided in ACTION, and an algorithm to compute orbits working with permutation groups through this interface is provided in SNAKES_AND_LADDERS. This way, applications that classify graphs, codes, designs, or geometries can be written that all use the same algorithm to classify orbits. The applications differ only in the group that they set up and possibly in the conditions on the subsets that define the combinatorial object. We will see a few examples for this model in the subsequent parts of this manual.

Let us come back once more to the issue that classes in the library do not know about classes in the application program. One way that this can be handled is through function pointer. The library classes call a function through a function pointer that has been stored previously. This function is known as a callback function. In order to access a member function from the class in the application (which cannot be accessed from the library, since the library does not know about the application class), one can use the following trick: The callback function receives an additional pointer of type void * which is the object of the class defined in the application. The library treats this pointer as a void pointer. The callback function casts this pointer to the type that it should be and then calls the member function that is supposed to perform the real work. We will see examples of this technique in several places below.

Next, we will describe various applications of orbiter.

3 Classifying Graphs and Tournaments

Let us look at how we can classify graphs and tournaments using orbiter. The program that we use is called graph.out and it resides in

   ORBITER/SRC/GRAPH

The testing will take place in

   ORBITER/DATA/GRAPH_AND_TOURNAMENT

In this directory, the makefile allows us to classify graphs and tournaments on $n$ vertices, for small values of $n$. For instance, typing
make g4

will produce the classification of graphs on 4 vertices. The actual command is

\texttt{ORBITER/SRC/GRAPH/graph.out -n 4 -v 1}

The output produced by this command contains the following lines:

\begin{verbatim}
Found 1 orbits at depth 6
  0 : 1 orbits
  1 : 1 orbits
  2 : 2 orbits
  3 : 3 orbits
  4 : 2 orbits
  5 : 1 orbits
  6 : 1 orbits
total: 11
\end{verbatim}

This means that there are 11 graphs on 4 vertices in total, and that the number of graphs on 4 vertices with \( k \) edges is

\begin{tabular}{|c|c|c|c|c|c|c|}
  \hline
  \( k \) & 0 & 1 & 2 & 3 & 4 & 5 \\
  \# graphs & 1 & 1 & 2 & 3 & 2 & 1 \\
  \hline
\end{tabular}

Of course, we could have noticed that the number of graphs with \( k \) edges is the same as the number of graphs with \( \binom{n}{2} - k \) edges, which would have saved us a little bit. Next, we could run successively the commands

\texttt{make g4}
\texttt{make g5}
\texttt{make g6}
\texttt{make g7}
\texttt{make g8}
\texttt{make g9}

to find out that the number of graphs with \( n \) vertices is

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
  \hline
  \( n \) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \# graphs & 1 & 2 & 4 & 11 & 34 & 156 & 1044 & 12346 & 274668 \\
  \hline
\end{tabular}

which is nothing else than Sloan’s sequence A000088 (we have taken the freedom to fill in the values for \( n = 1, 2, 3 \) by hand; these numbers are not difficult to come by). While it
is well-known that we can use techniques from enumerative combinatorics to obtain these numbers, the point that is interesting to note is this: By issuing the commands above, we have actually *constructed* each of the graphs in the list. If we were inclined to do so, we could look at these graphs and investigate them further. Of course, this would require us to gain a better understanding of how *orbiter* maintains lists of orbits.

Let us have a closer look at the classification program. The relevant files are:

```
ORBITER/SRC/GRAPH/graph.C
ORBITER/SRC/GRAPH/graph.h
ORBITER/SRC/GRAPH/graph_generator.C
```

Of these, *graph.C* is the main program, *graph.h* contains declarations, and *graph_generator.C* defines a class with the same name (we generally rely on the convention that a class is defined in a file with the same name). Let us have a look at the class `graph_generator`, which contains the following declarations (amongst others):

```cpp
class graph_generator {
    public:
        action *A_base; // symmetric group on n vertices
        action *A_on_edges; // action on pairs
};
```

These are pointers to objects of type `action`. An action is the way that permutation groups are defined in *orbiter*. The two actions `A_base` and `A_on_edges` correspond to the action on points and on edges, respectively. The first action is called `A_base` because we consider this the basic action from which the other action is derived. Looking into `graph_generator.C`, we see the following commands (inside the function `graph_generator::init`):

```cpp
A_base = new action;
A_base->init_symmetric_group(n, verbose_level - 3);
```

The new command allocates memory for the action object. The command `init_symmetric_group` makes `A_base` become a symmetric group $\text{Sym}_n$. In *orbiter*, this means that the group is acting on the numbers $0, 1, \ldots, n - 1$. Regarding the second action, we find the commands:

```cpp
A_on_edges = new action;
A_on_edges->induced_action_on_pairs(*A_base, A_base->Sims, verbose_level - 3);
```

The first command allocates memory for the object. The second command initialize the induced action of $\text{Sym}_n$ on the set of unordered pairs. This means that `A_on_edges` acts
on the numbers $0, 1, \ldots, \binom{n}{2} - 1$. The identification with unordered pairs is according to the lexicographic ordering of subsets of size 2:

$$01, 02, 03, \ldots, n - 1, n.$$ 

There are two reasons for working with two actions. First, it is more efficient to work with the group $\text{Sym}_n$ acting on the vertices. This is what the action $A_{\text{base}}$ is for. The action $A_{\text{on_edges}}$ is the action of this group on unordered pairs, and graphs can be identified with orbits on subsets of pairs in this action. Thus, by classifying the orbits on subsets in this action, orbiter will classify graphs on $n$ vertices. The size of the subset corresponds to the number of edges $k$ in the graph. The second reason for having the action $A_{\text{base}}$ around is the small degree of this action. As it turns out, permutation group algorithms are more efficient if small degree actions are used to represent the group. For this reason, all arithmetic in $\text{Sym}_n$ is done in the action on $n$ points. orbiter uses the action $A_{\text{on_edges}}$ whenever it is classifying subsets, but maintains the group and group elements in the action $A_{\text{base}}$.

In case that we are classifying tournaments (directions of the complete graph $K_n$), we work with the action on ordered pairs, created using

$$A_{\text{on_edges}}\rightarrow \text{induced_action_on_ordered_pairs}(*A_{\text{base}},$$

$$A_{\text{base}}\rightarrow \text{Sims}, \text{verbose_level} - 3);$$

In this case, the pairs are ordered in such a way that each ordered pair appears together with the pair obtained by reversing the order. These pairs appear consecutively, and with the pair that lists the smaller element first preceding the other pair. The lexicographic ordering of subsets is used to arrange the pairs. So, the ordering of the $n(n - 1)$ ordered pairs is

$$01, 10, 02, 20, 03, 30, \ldots, n - 1, n, n, n - 1.$$ 

Let us look at a specific example. Say we want to classify the graphs on 4 vertices with 3 edges. We run the command

$$\text{ORBITER/SRC/GRAPH/graph.out -n 4 -v 1 -W}$$

The option $-W$ means that the data that is computed is written to file. After completion, we find many files in the current directory. Let’s look into the file graph_4_lvl_3 which contains representatives for the orbits on graphs with 4 vertices and 3 edges (slightly edited to better fit the page):

```plaintext
# 3
3 0 1 2 6 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabaaaaaaaaacaaaaaaaa
   aaaaaaaaaaaaaaaaaacaaaaabadacaaacabadadaadacab
```
What is this telling us? First of all, the file is somewhat hard to read for humans. It is a compromise between efficiency and machine readability. It is also a relict of a file format from an earlier program system, so there is a legacy issue here. The first row indicates that we have representatives of size 3. The representatives are listed in the following three rows. Again, for some reason, each row starts by listing the size of the subset. Then, the subset is listed, followed by the order of the stabilizer. The remaining characters are an encoding of the generators for the stabilizer themselves, and are intended for machine processing. The row that starts with -1 lists (among other things that are not relevant at the moment) the number of representatives, and the distribution of group orders in parentheses. It also lists the average of all group orders. So, the graphs on 4 vertices with 3 edges are represented by

<table>
<thead>
<tr>
<th>Representative</th>
<th>Edges</th>
<th>Stabilizer Order</th>
<th>Description</th>
<th>Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 1, 2}</td>
<td>{01, 02, 03}</td>
<td>6</td>
<td>3-Claw</td>
<td><img src="image" alt="3-Claw" /></td>
</tr>
<tr>
<td>{0, 1, 3}</td>
<td>{01, 02, 04}</td>
<td>6</td>
<td>Triangle</td>
<td><img src="image" alt="Triangle" /></td>
</tr>
<tr>
<td>{0, 1, 4}</td>
<td>{01, 02, 12}</td>
<td>2</td>
<td>Path</td>
<td><img src="image" alt="Path" /></td>
</tr>
</tbody>
</table>

In the drawings, the vertex numbered 0 sits at the very top, and the ordering of vertices in counterclockwise. We will follow this convention throughout the manual. The average stabilizer order is \(\frac{6+6+2}{3} = \frac{14}{3}\).

Let us move on to the class of regular graphs. The command

\begin{verbatim}
make g8r3
\end{verbatim}

which is an abbreviation for

\begin{verbatim}
ORBITER/SRC/GRAPH/graph.out -n 8 -depth 12 -regular 3 -v 2 -W
\end{verbatim}

classifies cubic graphs on 8 vertices. We find 6 graphs. Reading the file
we find that these six graphs are represented by

<table>
<thead>
<tr>
<th>Representative</th>
<th>Stabilizer Order</th>
<th>Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 1, 2, 7, 8, 13, 22, 23, 24, 25, 26, 27}</td>
<td>1152</td>
<td><img src="image1" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 1, 2, 7, 8, 14, 19, 23, 24, 25, 26, 27}</td>
<td>16</td>
<td><img src="image2" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 1, 2, 7, 9, 15, 18, 20, 24, 25, 26, 27}</td>
<td>4</td>
<td><img src="image3" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 1, 2, 7, 9, 15, 20, 21, 23, 24, 25, 26}</td>
<td>12</td>
<td><img src="image4" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 1, 2, 9, 10, 14, 16, 19, 20, 24, 26, 27}</td>
<td>48</td>
<td><img src="image5" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 1, 2, 9, 10, 14, 16, 19, 21, 24, 25, 27}</td>
<td>16</td>
<td><img src="image6" alt="Drawing" /></td>
</tr>
</tbody>
</table>

Of course, the drawings are not particularly nice. Thinking about the issue of how to draw a graph “nicely” leads us to contemplate what representatives are chosen for the graphs that we compute. The answer is that orbiter always chooses the lexicographically least set in each orbit as representative. In addition, the representatives are listed in lexicographically increasing order.

Next, let us have a look at tournaments. A tournament is a directed graph such that any two vertices are connected by exactly one directed edge (thus, a tournament is a direction of a complete graph). Let us classify tournaments with orbiter. The command

```
make t4
```

which is an abbreviation for

```
ORBITER/SRC/GRAPH/graph.out -n 4 -v 2 -tournament -W
```
classifies tournaments on 4 vertices. The file `tournament_4_lvl_6` contains the 4 tournaments. They are:

<table>
<thead>
<tr>
<th>Representative</th>
<th>Stabilizer Order</th>
<th>Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 2, 4, 6, 8, 10}</td>
<td>1</td>
<td><img src="1" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 2, 4, 6, 9, 10}</td>
<td>3</td>
<td><img src="2" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 2, 5, 6, 8, 10}</td>
<td>1</td>
<td><img src="3" alt="Drawing" /></td>
</tr>
<tr>
<td>{0, 2, 5, 6, 8, 11}</td>
<td>3</td>
<td><img src="4" alt="Drawing" /></td>
</tr>
</tbody>
</table>

No piece in graph theory is complete without the unique graph on 10 vertices that is 3-regular of girth 5, better known as the Petersen graph. In `orbiter`, we issue the command

```
ORBITER/SRC/GRAPH/graph.out -n 10 -regular 3 -depth 15
-girth 5 -v 2 -W
```

The file `graph_10_r3_g5_lvl_15` is created (amongst many others), and contains the graph

\{0, 1, 2, 11, 12, 20, 21, 28, 29, 31, 33, 36, 38, 41, 42\}

with an automorphism group of order 120. When drawn, this graph looks like this

![Petersen graph](5)
Of course, because of the lexicographic condition on the orbit representatives chosen, this is not the best drawing of this graph.

This may be a good point to look at what orbiter is really doing when we ask for the classification of a certain set of orbits. In a nutshell, the classification proceeds by establishing a tree. The nodes of the tree are the orbits on “partial sets.” The definition of partial sets depends on the problem at hand. In the example of classifying graphs, we can think of the partial objects as graphs with fewer edges satisfying the natural induced conditions. A descendant $B$ of a node $A$ is a subset $B$ that contains $A$. An immediate descendant is a descendant whose size is exactly one larger than the previous set. Thus, $B$ is an immediate descendant of $A$ if $A \subseteq B$ and $|B \setminus A| = 1$. We can draw the tree in such a way that the root node (corresponding to the empty set) is at the very top, and so that the descendants are below the node from which they originate, connected to that node by an edge. In the case of the Petersen graph, this tree has 460 nodes and looks like this:

The Petersen graph itself is represented by the node that is at the very bottom and at the very left. The graphs represented in this tree have the property that they are defined on 10 vertices, each vertex has degree at most 3, and they contain no cycles of length less than 5. For each orbit of this kind of graph, the lexicographically least representation is chosen.

Let us look at the structure of the class graph_generator. In it, we find the declaration

```
generator *gen;
```

The orbiter class generator is responsible for classifying orbits of a group. To initialize, we find in the function graph_generator::init the call

```
gen->initialize(A_base, A_on_edges,
A_base->strong_generators, A_base->tl,
target_depth,
```
prefix, verbose_level - 1);

As pointed out above, the first two arguments \( A_{\text{base}} \) and \( A_{\text{on} \text{edges}} \) represent the permutation group whose orbits on subsets we plan to classify. The next two arguments \( A_{\text{base}}->\text{strong} \text{generators} \) and \( A_{\text{base}}->\text{tl} \) represent the actual generating system for the group. The argument \( \text{target} \text{depth} \) is a variable that has been computed previously to represent the size of subsets that we wish to classify. It depends on the type of graph or tournament that we specified using the command line arguments. The argument \( \text{prefix} \) is the prefix for the name of the output files. The last argument \( \text{verbose} \text{level} - 1 \) specifies how verbose the function \( \text{initialize} \) be during its processing.

In order to get started, we also need to specify the conditions that we want to impose on the subsets. These conditions depend on the type of graph or tournament that we wish to classify, and the specifics have been determined by command line arguments. Instead of going into all the details, let us just distinguish the case of graphs and the case of tournaments. In both cases, we inform the class \( \text{generator} \) of a test function. This is done by means of a function pointer. So, we find the command

\[
\text{gen->init_check_func}(::\text{check} \text{conditions},
(\text{void} *)\text{this}/*\text{candidate}\_\text{check}\_\text{data}*/);
\]

which informs \( \text{generator} \) of the presence of the function \( ::\text{check} \text{conditions} \). This test function is a global function because function pointers are a feature of C, and they can only be pointers to global functions, not member functions of a class. This leaves the issue of telling the global test function the instance of the class \( \text{graph}\_\text{generator} \). This is done by means of a void pointer. This is a pointer that is declared to be of type void, and that is passed to the class \( \text{generator} \). Whenever the instance of \( \text{generator} \) wishes to call the test function, it passes the data pointer as an extra argument. The test function then takes this argument and type casts it to the appropriate type (here \( \text{my}\_\text{generator} \)), and issues the call of the appropriate member function. Here is the definition of the global test function:

\[
\text{INT check_conditions(ostream &ost, INT len, INT *S,}
\text{ void *data, INT verbose_level)}
\]

\[
\{ 
\text{graph_generator *Gen = (graph_generator *) data;}
\]

\[
\text{if (Gen->f_tournament) }\{
\text{return Gen->check_conditions_tournament(len, S, verbose_level);} 
\}
\text{else }\{
\text{return Gen->check_conditions(len, S, verbose_level);} 
\}
\]
As we can see, the void pointer `data` is the fourth argument. The second and third argument are the set that we wish to test. It is simply a vector $S$ of integers of length $len$. The first argument can be ignored. The last argument specifies the veroseness of the test function. The line

```c
    graph_generator *Gen = (graph_generator *) data;
```

is the cast of the void pointer `data` to a pointer to an object of type `graph_generator`. This is so that the test function can then call the appropriate member functions of the class `graph_generator` to perform the actual testing.

4 Classifying Arcs, Ovals and Hyperovals

A set of points in the Desarguesian projective plane $\text{PG}(2, q)$ is called an `arc` if no three points of the set are collinear (i.e., lie on a line). Thus, an arc is a set of points that intersects any line in at most 2 points. It is known that that an arc has size at most $q + 2$. If $q$ is odd, this bound can be improved to $q + 1$. Arcs of size $q + 1$ are called `ovals` and arcs of size $q + 2$ are called `hyperovals`. Examples for ovals are the `conics`, i.e., the zero-sets of nondegenerate homogeneous quadratic polynomials in three variables. By a theorem of Segre, ovals in $\text{PG}(2, q)$ with $q$ odd are conics. For $q$ even, ovals are known that are not conics. Hyperovals arise from ovals using the following procedure. Consider the set of tangent lines to the oval (a tangent line is a line intersecting the oval in exactly one point; there is exactly one tangent line at each point of the oval). It is well-known that the set of $q + 1$ tangent lines to an oval intersect in a unique point, called the `nucleus` of the oval. The oval together with the nucleus gives rise to a hyperoval. If the oval is a conic, the resulting hyperoval is called a `hyperconic`. However, not every hyperoval must be a hyperconic.

A full classification of ovals and hyperovals seems to be beyond reach at the moment. For this reason, there is great interest in using the computer to classify ovals and hyperovals in small order projective planes. The computer can give us examples of hyperovals that any classification would have to include. Hyperovals have been classified for $q \leq 32$. We can use `orbiter` to reproduce these classifications on a laptop machine. However, the case of hyperovals in $\text{PG}(2, 64)$ seems to be beyond reach.

The program that we will use is called `hyperoval.out` and resides in

```
    ORBITER/SRC/HYPEROVAL
```

There is a `makefile` in

```
    ORBITER/DATA/HYPEROVAL
```
that we can use. The main body of the program is the class `arc_generator` in

```
ORBITER/SRC/HYPEROVAL/arc_generator.C
```

The class `arc_generator` maintains the projective plane \(\text{PG}(2,q)\) together with its group \(\text{PGL}(3,q)\). The incidence structure of the plane is encoded in an object

```
projective_space *P2;
```

that is initialized using the commands

```
P2 = new projective_space;
P2->init(2, q, FALSE /* f_init_group */, 
    f_semilinear, NULL /*const BYTE *override_poly*/, FALSE /* f_basis */, 
    0 /*verbose_level - 2*/);
```

This command sets up a 2-dimensional projective space defined over the field \(\mathbb{F}_q\). The group is maintained in an `action` object named \(A\), initialized via

```
A->init_matrix_group(TRUE /* f_projective */, 3, q, 
    /*override_poly*/, f_semilinear, f_basis, 0 /*verbose_level*/);
```

This sets up a 3-dimensional matrix group acting on projective space. The flag `f_semilinear` has previously been set to `TRUE` if \(q\) is not a prime. This flag determines whether we create the semilinear group. The flag `f_basis` is `TRUE` and implied that a stabilizer chain for the group is constructed. Two objects are implicitly set up. The first is an object of type `matrix_group`, the second is an object of type `finite_field`. The objects are initialized using the following commands

```
matrix_group *M;
finite_field *F;
M = A->G.matrix_grp;
F = M->GFq;
```

Since we will need the action on the set of lines also, we have a separate `action` object \(A_{on\_lines}\). This is initialized by first setting up an object of type `action_on_grassmannian`. The whole sequence of commands is

```
A_on_lines = new action;
AG = new action_on_grassmannian;
AG->init(*A, 3 /*n*/, 2 /*k*/, q, F, verbose_level - 2); 
A_on_lines->induced_action_on_grassmannian(A, AG, 
    FALSE /*f_induce_action*/, NULL /*sims *old_G*/, 
    MINIMUM(verbose_level - 2, 2));
```

16
Observe that the object `finite_field *F` is needed to set up the `action_on_grassmannian` object `AG`, which in turn is needed to set up the `action` object `A_on_lines`.

Inside `ORBITER/DATA/HYPEROVAL`, we could issue any of the commands

```
make 8
make 16
```

to classify the hyperovals in PG(2, 8) and PG(2, 16), respectively. This algorithm classifies the orbits of PTL(2, q) on arcs in PG(2, q). The results will be in the files

```
ORBITER/DATA/HYPEROVAL/CLASSIFY_8/arc_8_lvl_10
ORBITER/DATA/HYPEROVAL/CLASSIFY_16/arc_16_lvl_18
```

In PG(2, 8), the only hyperoval is the hyperconic. In PG(2, 16), there are two hyperovals. One is the hyperconic, and the other is the Lunelli-Sce hyperoval. They both can be found in the file `CLASSIFY_16/arc_16_lvl_18` (which – as pointed out before – is only semi-readable for humans). The search tree for the hyperovals in PG(2, 16) has 4214 nodes and looks like this:

![Search Tree Diagram](image)

With this tree, we can already spot the problem with this classification method. The tree is quite “bushy.” A great number of intermediate nodes run dead quickly, preventing this method from being effective for larger problem sizes. In order to proceed to larger instances of the problem of classifying hyperovals, we need to employ a different search method.

### 5 Classifying Hyperovals By Breaking The Symmetry

Let us look at the case of hyperovals in PG(2, 32). It is known that there are exactly six hyperovals. To classify the hyperovals in PG(2, 32), we cannot proceed as before. The reason
is that there are too many arcs of size 9 or more that need to be considered. This makes the previous algorithm very slow and most likely fail because of memory issues. Instead, we use a more powerful algorithm.

The algorithm that we use to classify hyperovals in harder cases is known as “breaking the symmetry”. This algorithm proceeds in three stages.

In stage one, we classify arcs of size $s$ first, for some relatively small value of $s$. This is done using the algorithm that we used before to classify orbits of arcs.

Once we have the classification of arcs of size $s$, we consider each arc in turn and extend that arc to a hyperoval in all possible ways. This is stage two, and it is called the lifting. Since hyperovals have size $q + 2$ (in this case), the lifting of a particular $s$-set amounts to choosing $q + 2 - s$ additional points satisfying all the requirements. This process can be formulated as an exact cover problem. This is a problem that has been studied by Computer Scientists for some time. Fast algorithms for solving instances of this problem are available, for instance Knuth’s algorithm dancing links.

Let us look in more detail at how this exact cover problem is formulated. First of all, the points that have been chosen already determine $\binom{s}{2}$ secants and $s(q + 1 - (s - 1))$ tangents. None of the points lying on a secant can be chosen, since we do not allow three points on a line. We say that a point is valid if it is not on any of the secants though any two of the $s$ chosen points (if $s = 1$, we simply require that the point be different from the chosen point). In order to turn the tangents into secants, exactly one valid point has to be chosen on each tangent. This is the lifting. Observe that some of the previously external lines may end up containing points from the lifting. We will take care of this later. To perform the lifting, we consider the set of tangent lines and the set of points that are valid. We form the incidence matrix whose rows are the tangent lines and whose columns are the valid points. A matrix entry in a certain row and column is one if the valid point associated to that column lies on the tangent line associated to that row. All other entries are zero. The exact cover problem is equivalent to finding $q + 2 - s$ columns of this incidence matrix so that the column vectors add up (over the integers) to the all-one vector. This kind of problem can be solved using the above-mentioned algorithm “dancing links”. The only issue with this approach is that we do not control the external lines. Some of them may end up with more than two points on it. These fake liftings do not correspond to hyperovals and need to be eliminated. As it turns out, the number of fake liftings is not that large, so we can eliminate them after the exact cover algorithm has completed.

Once we have all hyperovals that arise by extending arcs of size $s$, we perform another round of isomorphism classification. This is step three of the algorithm. Let us not go into the details of this step right now.

The classification of hyperovals in $PG(2, 32)$ is a case that is well-suited to breaking the symmetry. The command
make 32

which is equivalent to

```
ORBITER/SRC/HYPEROVAL/hyperoval.out -v 4 -q 32 -target_size 34
   -compute_starter 7 CLASSIFY_32/ -W
```

classifies the arcs of size 7 in PG(2, 32). Once this is done, we issue the command

```
make lift
```

which is defined as

```
ORBITER/SRC/HYPEROVAL/hyperoval.out -v 6 -q 32 -target_size 34
   -lift 7 CLASSIFY_32/ -lex -solve
```

to compute all liftings, i.e. all hyperovals containing one of the representatives of an isomorphism class of arc of size 7. Then we issue four more commands:

```
make 32_read_solutions
make 32_compute_orbits
make 32_classify
make 32_report
```

The first command reads the liftings. The second command processes the liftings and computes the orbits under the stabilizer of the arc from which the lifting originated. The third command performs the classification and the fourth command produces a report that is a tex file. These commands may take a little while. For instance, the command `make 32_classify` takes a little over an hour on a laptop. The total sequence should be done in less than 2 hours. The last command produces the file `hyperovals_32.tex` inside `ORBITER/DATA/HYPEROVAL`. This file can be processed using `pdflatex hyperovals_32.tex`. The resulting file `hyperovals_32.pdf` contains the classification of hyperovals in PG(2, 32) in a human-readable form. The file is available from the `orbiter` web-site listed above.

Let us look at the specifics of how the lifting is done. Inside `arc_generator`, there is a function

```
void arc_generator::compute_lifts(const BYTE *prefix, INT starter_size,
   INT f_lex, INT f_split, INT split_r, INT split_m,
   INT f_solve, INT f_read_instead,
   INT verbose_level)
```
This function performs the lifting. The arguments are as follows: `prefix` is a string that is used to locate the files from Step 1. `starter_size` is the size of the starters that should be used. The classification of starters of that size must have been done in Step 1 (but it would be OK if we had classified larger size sets in Step 1, except that we would not use this knowledge). The option `f_lex` determines if we apply the lexicographic ordering to reduce the set of liftings. The options `f_split, split_r` and `split_m` have to do with parallel computing. They allow that only a slice of the lifting problem is done. Namely, if `f_split` is `TRUE` then only starter whose number `i` is congruent to `split_r` modulo `split_m` are listed. The option `f_solve` determines whether the lifts should be computed. If `FALSE`, the system of equations is prepared but not solved. The option `f_read_instead` is useful if the systems have been handed over to an external solver, and the solutions from the external solver should be read from file. The final option `verbose_level` determines how much text output should be produced.

Let us look inside the function `arc_generator::compute_lifts`. We find the data structure

```c
exact_cover *E;
```

that manages the exact cover problem. The object of type `exact_cover` is initialized using the commands

```c
E->init(this /* user_data */,
    A, A,
    nb_points_total, target_size, block_size, starter_size,
    prefix, base_fname,
    f_lex, f_split, split_r, split_m,
    arc_generator_lifting_prepare_function,
    arc_generator_lifting_cleanup_function,
    arc_generator_early_test_function,
    this,
    verbose_level - 1);
E->add_solution_test_function(
    callback_arc_test,
    (void *) this,
    verbose_level - 1);
```

The `init` function sets up the `exact_cover` object. The `add_solution_test_function` function is optional, and can be used if an additional test should be performed on the solutions of the exact cover problem. We will talk about this in a little while. Let us look at the `init` function first. The arguments `nb_points_total, target_size, block_size` and `starter_size` specify the parameters of the exact cover problem: `nb_points_total` is set to \( q^2 + q + 1 \) (in `arc_generator::init`) and gives the number of points (or lines) in the
projective plane, target_size specifies the size of the object that we wish to create (here, a hyperoval has $q + 2$ points), block_size is the number of lines through a point and also the number of points on a line. Finally, starter_size is the size of the starter sets. The exact_cover object cannot initialize itself, since it does not know what problem we are going to solve (exact_cover is a class in the orbiter class library and as such does not know what exact cover problem it is about to be used for). For this reason, we pass function pointers

```
arc_generator_lifting_prepare_function,
arc_generator_lifting_cleanup_function
arc_generator_early_test_function, and
```

as well as a data pointer this, representing the arc_generator object itself. These function pointers are used to maintain the instance of the exact cover problem:

```
arc_generator_lifting_prepare_function
```

sets up the instance of the exact cover problem that is associated to one particular starter set. It does so by computing several arrays and matrices that encode the exact cover problem. This function is invoked once the starter set has been read from file, which is at the beginning of the exact cover solver.

```
arc_generator_lifting_cleanup_function
```

frees up any data pointers that have been allocated by the previous function, and is invoked once the exact cover problem has been solved, which is at the end of the exact cover solver. The function

```
arc_generator_early_test_function
```

is needed to that we can test if a certain point can possibly be added to a set of points previously chosen. It is the same test function that is used by blt.out when it classifies arcs. It simply tests if a given point is compatible with a previously chosen set.

The job of the exact_cover object is to run over all starter, load a particular starter from file, load a set of points called the live points that go along with it, and handle the exact cover problem though the callback functions described above. Once the exact cover problem is defined, it will proceed in one of several ways. If the flag f.solve is TRUE, an object of type exact_cover_solver is created and the DLX algorithm (dancing links) is invoked to solve the problem. If the flag f.read_instead is TRUE then the solutions are read from a file. This is useful if an external solver is used.

The final command add_solution_test_function in the sequence of commands that set up the exact_cover object E (see above) installs yet another function pointer. The purpose if this function is to do an extra test on the solutions of the exact cover problem. As we
pointed out previously, the exact cover problem does not completely capture the condition that no three points are collinear. There may be lines that are external to the starter set that end up with more than two points from the exact cover solver. These fake solutions need to be removed. This is exactly the purpose of the function callback_arc_test.

6 Classifying BLT-Sets

We will describe one application that falls in the area of finite geometry. A $Q(4, q)$ space is a four-dimensional projective space over the field $\mathbb{F}_q$ equipped with a non-degenerate quadratic form. In such a space, sets of points that are known as BLT-sets are of interest. A BLT-set is a set of $q+1$ points on the $Q(4, q)$ quadric satisfying a triple condition. Namely, whenever $P, Q, R$ are any three points of the BLT-set then $(P + Q + R)\perp$ is an anisotropic line. This means that the three points form a triad (a triangle without any sides) such that no point of $Q(4, q)$ is collinear to all three of $P, Q, R$. BLT-sets are known to exist whenever $q$ is an odd prime power. It is an important problem to classify BLT-sets. By this, we mean the following: Two BLT-sets of $Q(4, q)$ are isomorphic if there is a collineation of $Q(4, q)$ that takes one set to the other. The classification problem for BLT-sets asks for the isomorphism classes of BLT-sets. This is computing the orbits of the group of collineations on the set of BLT-sets of $Q(4, q)$.

To classify BLT-sets of $Q(4, q)$ using orbiter, we again use the principle of Breaking the Symmetry. In this case, this means we will classify partial BLT-sets of size $s$ first, where $s$ is a relatively small number (compared to $q$). The representative of these orbits are the starter. Once this is done, all BLT-sets containing any of the starter will be computed. This is done using rainbow cliques in colored graphs. Finally, another round of isomorph classification will take place. Once all this is done, we can create a pdf file containing the classification in human-readable format.

The main program that we use is called blt.out. It resides in

```
ORBITER/SRC/BLT
```

and facilitates the search and classification for BLT-sets and partial BLT-sets. Here, a partial BLT-set is a set of points on the $Q(4, q)$ quadric such that $(P + Q + R)\perp$ is an anisotropic line for any three points $P, Q, R$ in the set. Thus, a BLT-set is a partial BLT-set of size $q+1$. For small values of $q$, one can classify the BLT-sets using blt.out. The search tree for the
BLT sets in $Q(4,11)$ has 189 nodes and looks like this:

The four isomorphism classes of BLT-sets of $Q(4,11)$ correspond to the leaves at depth 12 in this tree.

The program blt.out relies on the class blt_generator in

```
ORBITER/SRC/BLT/blt_generator.C
```

The class blt_generator maintains the quadric $Q(4, q)$ together with its group $P\Gamma O(5, q)$. The incidence structure of the quadric is encoded in an object

```
orthogonal *O;
```

The object $O$ is set up automatically by the function that sets up the group, which is

```
A->init_orthogonal_group(epsilon, n, q, override_poly,
    f_semilinear, f_basis, 0/*verbose_level - 2*/);
```

Here, $A$ is an action object that is part of the class arc_generator, epsilon is 0, $n$ is 4, and $q$ is the field order. The argument override_poly is a number specifying a polynomial over $F_p$ that is used to create the field $F_q$ (this is only relevant if $q = p^h$ for some prime $p$ and some integer $h > 1$). The flag f_semilinear specifies if we want the semilinear group and the flag f_basis specifies if we want to set up a stabilizer chain for the group (it should be TRUE). Once the orthogonal group is set up, we can initialize two object pointers:

```
matrix_group *M;
orthogonal *O;
M = A->subaction->G.matrix_grp;
O = M->O;
```
Classifying BLT-Sets By Breaking the Symmetry

If we want to proceed to higher orders of $q$, we find that the classification method in `blt.out` is not optimal. For this reason, we resort to the method of Breaking the Symmetry. This means the following. In Step 1, we classify the partial BLT-sets of some size. Let us call the representatives of the isomorphism types starter. In Step 2, we lift – in all possible ways – all starter sets to BLT-sets. This can be done by formulating the problem of lifting BLT-sets in the language of graph theory. The lifts of a BLT-starter $S$ are in one-to-one correspondence to the rainbow cliques of size $q + 1 - |S|$ in a certain colored graph $\Gamma_{S,\ell}$. Here, $\ell$ is a line intersecting $S$ in a single point. We will have `blt.out` create the colored graphs, and invoke the program `all_cliques.out` to do the clique finding for us. Computationally, the clique finding step is the dominant part of the algorithm. Once all the cliques of $\Gamma_{S,\ell}$ for all starter sets $S$ have been found, we move on to Step 3, where we do a final isomorph classification. This is done back in `blt.out`.

Let us look at an example. Suppose we wish to classify the BLT-sets of $Q(4,23)$. The example `makefile` for this classification problem is in

```
cd ORBITER/DATA/BLT/23
```

In this directory, we issue the command `make`. This will compute the classification of BLT-sets of order $q = 23$. After a few minutes, the file

```
```

containing a report of the 9 BLT-sets of $Q(4,23)$ is created. This file is human-readable. To change the parameter $q$ to any odd prime power $q$ you wish to investigate, we can change the line

```
Q=23
```

in `makefile`. Larger values of $q$ require longer computing times. The largest value of $q$ that has been run to this day was $q = 67$. The computation was performed in parallel and took 16 years total to complete. This is definitely not a computation that should be tried at home.

Contributors

The following list acknowledges people who have contributed code to `orbiter`, most of them unknowingly.
<table>
<thead>
<tr>
<th>Who</th>
<th>What</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brendan McKay</td>
<td>Nauty</td>
<td>Program to compute a canonical form of graphs</td>
</tr>
<tr>
<td>Brendan McKay</td>
<td>Possolve</td>
<td>Program to solve diophantine systems (author)</td>
</tr>
<tr>
<td>Alfred Wassermann</td>
<td>Possolve</td>
<td>Initial adaptation</td>
</tr>
<tr>
<td>Volker Widor</td>
<td>Possolve</td>
<td>C++ translation</td>
</tr>
<tr>
<td>Paul Hsieh</td>
<td>Super fast hash</td>
<td>Computing hash values</td>
</tr>
<tr>
<td>Xi Chen</td>
<td>DLX</td>
<td>Implementation of Don Knuth’s dancing links algorithm</td>
</tr>
<tr>
<td>Wolfgang Boessenecker</td>
<td>Code/Decode</td>
<td>Simple compression algorithm</td>
</tr>
</tbody>
</table>