Orbiter – A System For Classifying Combinatorial Objects

Anton Betten

March 16, 2016

1 Introduction

Combinatorial objects are orbits under finite group actions. Combinatorial problems often require us to find the number of orbits either by counting arguments or by construction of orbit representatives. This is often very tedious and computers can be of great help. Orbiter is a software package devoted to solving these kinds of combinatorial problems. Orbiter is designed as a package of C++ classes which provide tools for studying many different kinds of combinatorial objects such as

1. Graphs,
2. Codes,
3. Designs,
4. Finite fields,
5. Vector spaces and subspaces over a finite field,
6. Finite geometries including spreads, translation planes, and polar spaces,
7. Permutations, partitions, subsets etc,
8. Partially ordered sets and lattices.

The basic design principle behind orbiter is to model combinatorial objects as equivalence classes of functions under group actions in the domain and in the range. We call these equivalence classes *symmetry classes of mappings.* In the most general case, the functions are between finite sets. We can also consider linear functions between finite dimensional vector spaces over finite fields. In Section 2, we will describe the theory by which combinatorial objects can be modeled as symmetry classes of mappings. After that, we look at some algorithms to classify combinatorial objects using these models. Orbiter provides efficient implementations of these algorithms. Classification programs for several types of combinatorial objects are provided with Orbiter. Additional classification programs can be implemented using the library. Orbiter is an open system, which means that additional C++ classes can be added.

2 Modeling Combinatorial Objects

A combinatorial object is an instance of a *type of combinatorial objects.* A combinatorial object lives in a space with symmetry and there is a notion of equivalence. Here, we wish to describe combinatorial objects using functions. The equivalence of combinatorial objects translates to a notion of equivalence of these functions. There are two basic models. One model consists of functions between sets. The second model has linear functions beween vector spaces. We call the two models the *set-model* and the *linear model.* We will first introduce the two models and then discuss some examples of combinatorial objects and how they can be modeled.

The set-model consists of three ingredients. There is

1. a group G acting on a finite set D,
2. a group H acting on a finite set R,
3. a set $\mathfrak{F}$ of functions from D to R.

Two functions $f_1, f_2 \in \mathfrak{F}$ are equivalent if there exist elements $g \in G$ and $h \in H$ such that

$$\bar{h} \circ f_1 \circ \bar{g}^{-1} = f_2$$

Here, the bar signifies that we consider the mappings which are induced by the respective element under the group action. This definition defines an action of $G \times H$ on $\mathfrak{F}$.

The *linear model* consists of

1. a finite field $\mathbb{F}_q$,
2. vector spaces D and R over $\mathbb{F}_q$,
3. a semilinear group G acting on a finite vector space D,
4. a semilinear group H acting on a finite vector space R,
5. a set $\mathfrak{F}$ of $\mathbb{F}_q$-linear mappings from D to R.

Two linear functions $f_1, f_2 \in \mathfrak{F}$ are equivalent if there exist elements $g \in G$ and $h \in H$ such that

$$\bar{h} \circ f_1 \circ \bar{g}^{-1} = f_2$$

The notion of equivalence introduces an equivalence relation on the set of combinatorial objects of a fixed type. It is also customary to call two objects *isomorphic* if they are equivalent. In Kerber [2], the orbits of the group $G \times H$ on $\mathfrak{F}$ under the action introduced above are called *symmetry classes of mappings*. The *isomorphism class* of a mapping $f \in \mathfrak{F}$ is the set of all $f' \in \mathfrak{F}$ which are isomorphic to f.

Combinatorial objects often come in families. A *family of combinatorial objects* is a sequence of combinatorial objects where the construction rule is fixed and only the basic set and the group action vary with a parameter n, say. There may be restrictions on the paramater n. For a family, we require that there are infinitely many possible values of n.

The task of determining the isomorphism classes of a certain combinatorial object is known as the *classification problem*. We distinguish different levels of solving the classification problem. There is a hierarchy of problems, listed here in the order of increasing difficulty:

1. The *existence problem*: We ask if there exists a combinatorial object of a given type.
2. The *orbit counting problem*: We determine the number of isomorphism types of combinatorial objects of a given type. At this level, we will be satisfied with the number of orbits. This problem is treated in enumerative combinatorics.
3. The *constructive enumeration / classification problem*: We ask to determine representatives for each isomorphism class.

A classification problem as described above aims at finding a system of representatives for the orbits of a group G on a specific set of objects. A system of representatives consists of exactly one object for each of the different G-orbits. Such a system is called a *transversal of the orbits.* Together with the representatives, we also want to find the stabilizer groups and the orbit lengths.

Beyond finding the transversal, we also want to be able to *recognize* objects and identify orbits. This means the following: Given any object, we want to determine the unique element in the transversal isomorphic to the given object. This element is called the *orbit representative* associated to the given object. The *recognition problem* requires us to recognize objects. A refined version of this is the *constructive recognition problem.* It requires us to recognize orbits and to identify a group element $g \in G$ which maps the given object to the representative.

Let us look at some combinatorial objects and how to represent them as symmetry classes of mappings.

1. **Graphs**: A graph on n vertices is a binary relation on a set V of n elements, called vertices. The pairs of vertices which are incident are called edges. The edge set E is the set of all edges. The graph is thus a pair (V, E). For simple graphs, we require that the binary relation is totally irreflexive and symmetric. Let the group $G = \mathrm{Sym}_n$ act on the set $V = \mathbb{Z}_n$. Let D be the set of unordered pairs of elements from V. Then G acts on D also. Let $R = \mathbb{Z}_2$ and consider the trivial group $H = 1$ acting on R. Let $\mathfrak{F}$ be the class of all functions from D to R. Then $\mathfrak{F}$ under the action of $G \times H = G \times 1$ corresponds to the simple graphs on n vertices.

2. **Sets of size k under equivalence**: Many combinatorial objects arise naturally as orbits of a group acting on k-subsets of a fixed set, for some fixed k. Let X be a set and let G be a group acting on X. The G-orbits on k-subsets of X can be described as the set $\mathfrak{F}$ of symmetry classes of mappings $f : \mathbb{Z}_k \to X$ under the action of $\mathrm{Sym}_k \times G$. The groups are Sym_k acting on $D = \mathbb{Z}_k$ and G acting on $R = X$.

3. **Subspaces of dimension k in $\mathbb{F}_q^n$ under equivalence**: Consider a group $A \leq \Gamma\mathrm{L}(n, q)$ acting on subspaces of dimension k for some $k \leq n$. The orbits of this action can be described as symmetry classes of mappings. Pick $D = \mathbb{F}_q^k$ and $R = \mathbb{F}_q^n$. Furthermore, pick $G = \mathrm{GL}(k, q)$ and $H = A$. The functions which we consider are the $\mathbb{F}_q$-linear functions $f : D \to R$ which are one-to-one. Thus

$$\mathfrak{F} = \{f : D \to R \mid f \text{ is linear and one-to-one }\}.$$

3 The classification algorithm

How does one classify combinatorial objects? From what we have learned in the previous section, the problem of classification requires us to compute orbits of certain groups acting on certain classes of functions as discussed in Section 2. The difficulty is that the set of functions may be very large, so that it may be infeasible to even look at every function. The existence of a group action on the set of functions may make this problem feasible in the first place. The set of functions may be very large but the number of orbits under the group may still be reasonably small. The question is if we can get hold of these orbits efficiently. To do so, we would like to avoid having to look at every function in the set $\mathfrak{F}$.

The main idea in the classification algorithm is to break the problem into smaller steps, each of which can be handled with reasonable effort. To enable small steps, we use an inductive scheme. In each step, we take a smaller classification problem and lift it to a larger one. To enable induction, we embed the combinatorial objects into a set which is equipped with an order relation. We use order structures like lattices and partially ordered sets (posets, for short). We start by embedding the combinatorial structures of the given type into a lattice. In order to enable induction, we require that the lattice has a rank function. What is a rank function? If $\mathfrak{L}$ is a lattice, a rank function on $\mathfrak{L}$ is the mapping $r : \mathfrak{L} \to \mathbb{Z}_{\geq 0}$ such that $r(0) = 0$ and if $x < y$

and there is no z such that $x < z < y$ then $r(y) = r(x) + 1$. The rank function as defined this way may or may not be well-defined. If it is, the we say that the lattice is ranked. For instance, the power set $\mathfrak{P}(X)$ of a fixed finite set X (i.e., the set of all subsets of X) is ranked. The rank of a set is the size of the set. Another example of a ranked lattice is the lattice $\mathfrak{L}(V)$ of all subspaces of the finite dimensional vector space V. The rank of a subspace is given by the dimension. The i-th layer in $\mathfrak{L}$ is

$$\mathfrak{L}_i = \{x \in \mathfrak{L} \mid r(x) = i\}.$$

The next step is to consider group actions on lattices and posets. A group G is said to act on the lattice $\mathfrak{L}$ if

$$x < y \Rightarrow x^g < y^g$$

for all $x, y \in \mathfrak{L}$ and all $g \in G$.

Let $\mathcal{C}$ be the combinatorial structures of a given type. Let $\mathcal{C}$ be embedded in a ranked lattice $\mathfrak{L}$. We require that there is a subset $\mathcal{P}$ of $\mathfrak{L}$ with the following properties:

1. $\mathcal{C} \subseteq \mathcal{P}$.
2. If $y \in \mathcal{P}$ and $x \in \mathfrak{L}$ with $x < y$ then $x \in \mathcal{P}$ also.
3. If $x \in \mathcal{P}$ and $g \in G$ then $x^g \in \mathcal{P}$ also.

The subset $\mathcal{P}$ of $\mathfrak{L}$ carries an order structure induced from $\mathfrak{L}$, and hence $\mathcal{P}$ is itself a poset. In fact, $\mathcal{P}$ is a semilattice: with x and y in $\mathcal{P}$, we also have $x \wedge y \in \mathcal{P}$ (but not necessarily $x \vee y \in \mathcal{P}$). We put $\mathcal{P}_i = \mathfrak{L}_i \cap \mathcal{P}$. Let $R = \{r(x) \mid x \in \mathcal{P}\}$. Then R is an interval of the form $[0, \ldots, t]$. The level t is called the target level. We introduce an indicator function for the poset $\mathcal{P}$ inside $\mathfrak{L}$. Let $\chi : \mathfrak{L} \to \{0, 1\}$ be one if x is in $\mathcal{P}$ and zero otherwise. The properties for $\mathcal{P}$ translate into properties for χ

1. $\chi(x^g) = \chi(x)$ for all $x \in \mathcal{L}$ and all $g \in G$, and
2. For $x < y$, we have $\chi(y) = 1$ implies $\chi(x) = 1$.

A function χ satisfying these two requirements is called a *test-function*.

With this setup, we can now devise an inductive scheme to classify the combinatorial objects $\mathcal{C}$. The classification process computes the orbits of G on $\mathcal{P}$ by induction on the layers. This computation starts at the bottom layer $\mathcal{P}_0$ and moves up from $\mathcal{P}_i$ to $\mathcal{P}_{i+1}$ for $i = 0, \ldots, t - 1$. In each step, the orbits of G on $\mathcal{P}_i$ are classified. Once $\mathcal{P}_t$ is classified, algorithm terminates. The G-orbits on $\mathcal{C}$ have been computed.

The set of G-orbits on the poset forms another partially ordered set. We say that $x^G < y^G$ if there are elements a, b with $a \in x^G$ and $b \in y^G$ and $a < b$ in the original poset. The poset that we obtain in this way is called the *poset of G-orbits*.

4 Examples

Let us illustrate this technique of classifying combintorial objects. We go back to the examples already introduced.

Let us first consider the problem of classifying graphs with n vertices. Let $V = \{0, \ldots, n - 1\}$ be the vertex set. Recall that two graphs are isomorphic if there is an element in Sym_V that takes the incidence

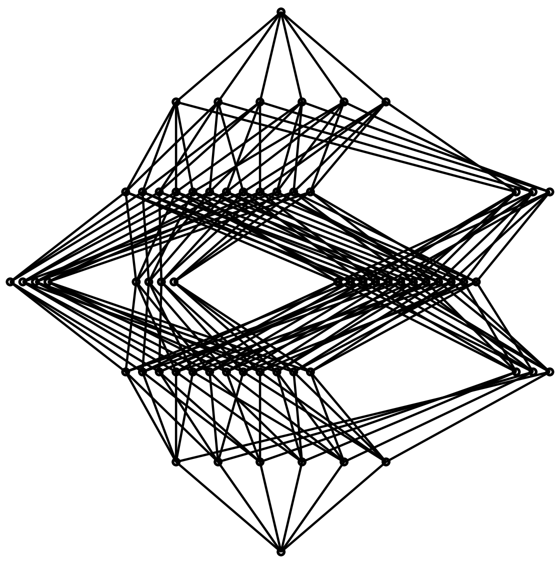


Figure 1: The lattice $\mathfrak{L} = \mathcal{P}$ of graphs on 4 vertices

relation of one graph to the incidence relation of the other. That is, for graphs (V, R_1) and (V, R_2) and for $g \in \mathrm{Sym}_n$,

$$(i^g, j^g) \in R_2 \iff (i, j) \in R_1.$$

We let $E = \binom{V}{2}$ be the set of all unordered pairs of elements of V and we identify a graph with the subset of E that corresponds to the incident pairs of vertices in the graph. The lattice $\mathfrak{L}$ and the partially ordered set $\mathcal{P}$ are both equal to $\mathfrak{P}(E)$, the power set of E. Thus, the test function χ is trivial: it yields the value one for every $x \in \mathfrak{L}$. The ordering of graphs is given by the inclusion of the associated subsets of E. Thus, a graph (V, R_1) is contained in (V, R_2) if $(i, j) \in R_1$ implies $(i, j) \in R_2$ for all $i, j \in V$. Figure 1 shows the lattice $\mathcal{P} = \mathfrak{L}$ of graphs on $n = 4$ vertices. The orbits of $G = \mathrm{Sym}_4$ are indicated by grouping the vertices together. Of course, the large number of nodes make this diagram somewhat hard to read. Often, all we need to know is a representative of each orbit. Hence it is more instructive to draw the poset of orbits as in Figure 2. The nodes represent orbits of graphs, and one representative graph is shown in each node. This representative is in fact the lexicographically least graph in its orbit.

Let us look at an example where the test function is more interesting. Suppose we want to classify cubic graphs on n vertices. A graph is cubic if every vertex is incident with exactly three edges. For cubic graphs to exist, the number of vertices has to be even. Again, we let $V = \{0, \ldots, n-1\}$ be the set of n vertices, and we have $E = \binom{V}{2}$ the set of unordered pairs of V. Again $\mathcal{L} = \mathfrak{P}(E)$ the power set of E. The test function χ and the poset $\mathcal{P}$ is defined as follows. If x is a subset of E, we set $\chi(x) = 1$ if each vertex in V is incident with *at most three edges* in x. It is easy to check that this function χ is a test function. We let $\mathcal{P} = \{x \in \mathfrak{L} \mid \chi(x) = 1\}$. The search poset $\mathcal{P}$ has elements of rank r for $0 \leq r \leq t$ where $t = \frac{3n}{2}$. Let us choose specific values. For $n = 6$, we find $t = 9$.

Suppose we want to classify cubic graphs on 6 vertices. The poset of orbits is shown in Figure 3.

This computation appears somewhat inefficient. It is better to observe that the complement of a cubic graph on 6 vertices is regular of degree 2. The 2-regular graphs on 6 vertices are much easier to compute since they have only 6 edges, and hence $t = 6$. For 2-regular graphs, we consider the search poset consisting of subsets of E which have the property that each vertex is incident with *at most two edges*. The resulting poset of orbits for this problem is shown in Figure 4.

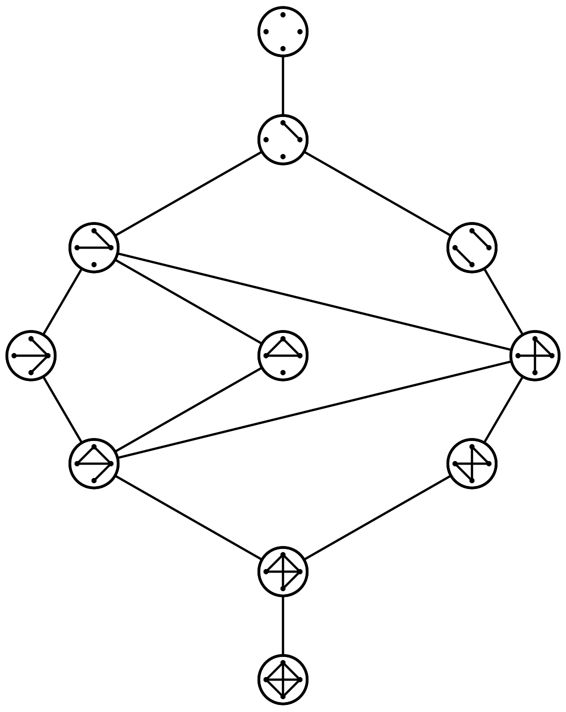


Figure 2: The poset of orbits

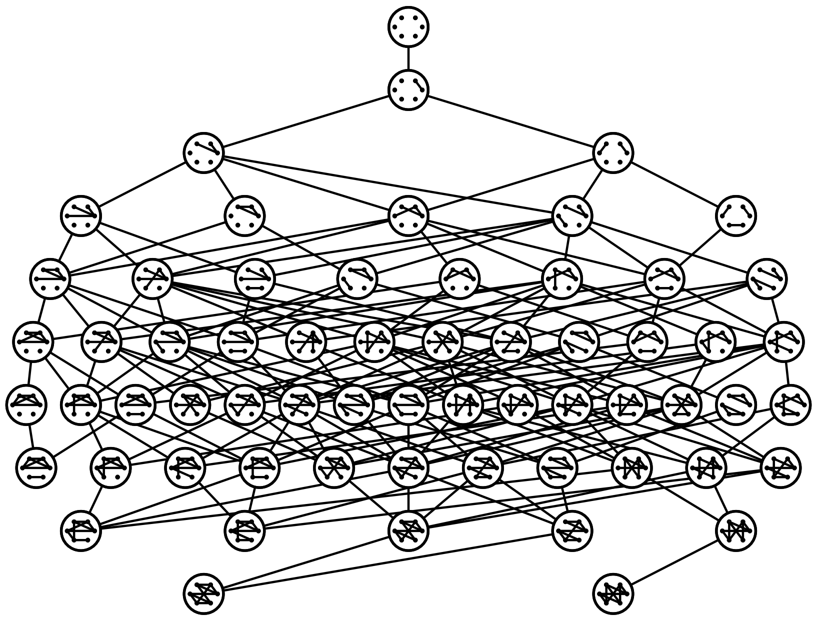


Figure 3: The poset of orbits for 3-regular graphs on 6 vertices

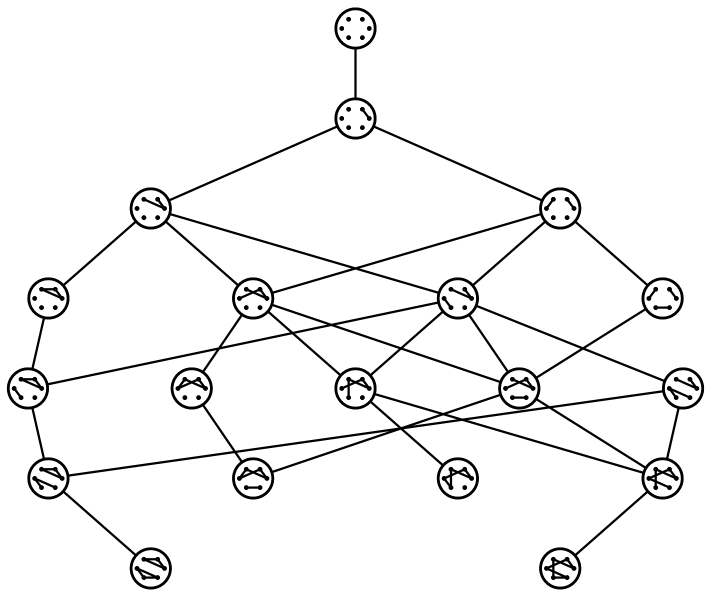


Figure 4: The poset of orbits for 2-regular graphs on 6 vertices

Let us now look at an example involving subspaces.

Suppose we want to classify optimal linear codes. An $[n, k, d]_q$ code is a linear code over the field $\mathbb{F}_q$ of length n, dimension k and with minimum distance d. What this means is that the code is a vector subspace of $\mathbb{F}_q^n$ of dimension k, such that the minimum distance between any two vectors in the space is at least d and there are two vectors at distance d. The distance between two vectors is the *Hamming distance,* which counts the number of places in which two given vectors differ. The most interesting codes have large d. So, for fixed n and k and q, we want to find the codes with largest d. These kinds of codes are called *distance optimal.* The group $G = \Gamma M(n, q)$ of semilinear monomial matrices over $\mathbb{F}_q$ acts on the set of codes and two codes are called equivalent if they lie in the same orbit under the group G. Suppose we want to find the best codes of length 8 and dimension 4 over the field $\mathbb{F}_2$ Suppose that we want a minimum distance of 4. We model this problem using the lattice $\mathfrak{L}$ of all subspaces of $\mathbb{F}_2^8$. The search poset $\mathcal{P}$ consists of all subspaces whose minimum distance is at least 4. The classification algorithm produces the poset of orbits indicated in Figure 5. Each level in this poset represents orbits of the monomial group on codes of length 8 with a certain dimension. The topmost level is dimension 0, the bottom level is dimension 4. Since we require distance 4, the choices for the first basis vector are limited to vectors with weight at least 4. The monomial group is transitive on vectors of a fixed weight, so the orbits at level one correspond to subspaces spanned by the vectors

$$(1, 1, 1, 1, 0, 0, 0, 0), (1, 1, 1, 1, 1, 0, 0, 0), (1, 1, 1, 1, 1, 1, 0, 0), (1, 1, 1, 1, 1, 1, 1, 0), (1, 1, 1, 1, 1, 1, 1, 1).$$

Let us not go through all the other orbits except point out that the unique orbit at level 4 (at the bottom) is the binary Hamming $[8, 4, 4]$ code with generator matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

and automorphism group of order 1344. This computation is proof that the $[8, 4, 4]$ binary Hamming code is unique (since there is only one node at the bottom of this diagram).

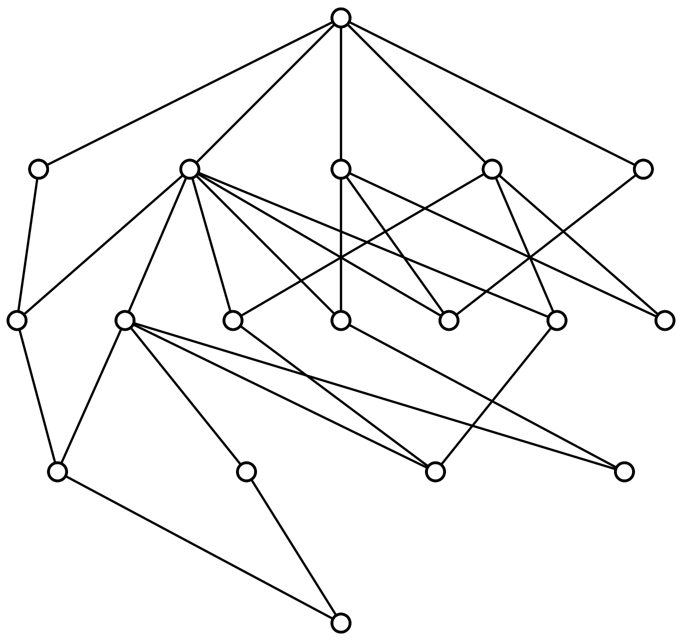


Figure 5: The poset of orbits for the classification of $[8, 4]$ codes with distance at least 4

The computation in this example is somewhat inefficient since there are a lot of orbits. An improvement can be achieved if the results described in [1] are used. The idea is to replace the linear model with a suitable model of functions between sets.

Here is the result from [1] that we need: Under the mild assumption that d is at least three, the combinatorial object can now be described using projective geometries. Namely, it can be shown that a code over $\mathbb{F}_q$ of length n, dimension k and distance at least d corresponds to a set of n points in $\mathrm{PG}(n-k-1, q)$ such that any $d-1$ are independent. Furthermore, equivalent codes under the action of G correspond to projectively equivalent sets under $\mathrm{P\Gamma L}(n-k, q)$ and conversely. For this reason, the problem of classifying optimal codes is equivalent to the problem of classifying up to projective equivalence the sets of size n in $\mathrm{PG}(n-k-1, q)$ with the property that any $d-1$ are independent.

The problem of classifying sets of points in projective space can be modeled using functions between sets. We consider subsets x of $\mathrm{PG}(n-k-1, q)$ of size n *or less* with the property that any $d-1$ are independent. If x has size less than $d-1$, we require that x itself be independent. Let us call such subsets *special*. Next, we consider the lattice $\mathfrak{L} = \mathfrak{P}(\mathrm{PG}(n-k-1, q))$ of all subsets of points of $\mathrm{PG}(n-k-1, q)$. We define a test function $\chi : \mathfrak{L} \to \{0, 1\}$ which maps x to 1 if x is special and to 0 otherwise. Then, as usual, $\mathcal{P} = \{x \in \mathfrak{L} \mid \chi(x) = 1\}$ is the search poset.

Let us look at the example of binary $[8, 4]$ codes with minimum distance 4 again. We compute the poset of orbits of special subsets in $\mathrm{PG}(3, 2)$ shown in Figure 6. The bottom node corresponds to the unique $[8, 4, 4]$ binary Hamming code.

The classification algorithm turns the poset of G-orbits into a tree. We call this tree the *tree of G-orbits*. The advantage of a tree is that each node and hence each G-orbit is visited exactly once. The way that we more from the poset of G-orbits to the tree of G-orbits is by removing edges. More precisely, we remove edges from the poset so that each node is incident with exactly one node below. This unique node below a given node is called the *canonical ancestor*.

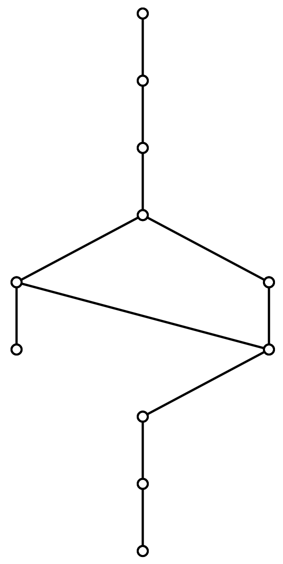


Figure 6: The poset of orbits for the Hamming code

The tree of orbits for the problem of 2-regular graphs from Figure 4 is shown in Figure 7.

5 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/SRC/
ORBITER/SRC/APPS/
ORBITER/SRC/APPS/BLT/
ORBITER/SRC/APPS/GRAPH/
ORBITER/SRC/APPS/HYPEROVAL/
ORBITER/SRC/APPS/TOOLS/
ORBITER/SRC/APPS/TRANSLATION_PLANE/
ORBITER/SRC/LIB/
ORBITER/SRC/LIB/ACTION/
ORBITER/SRC/LIB/DISCRETA/
ORBITER/SRC/LIB/GALOIS/
ORBITER/SRC/LIB/INCIDENCE/
ORBITER/SRC/LIB/orbiter.h
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/
ORBITER/SRC/LIB/TOP_LEVEL/
```

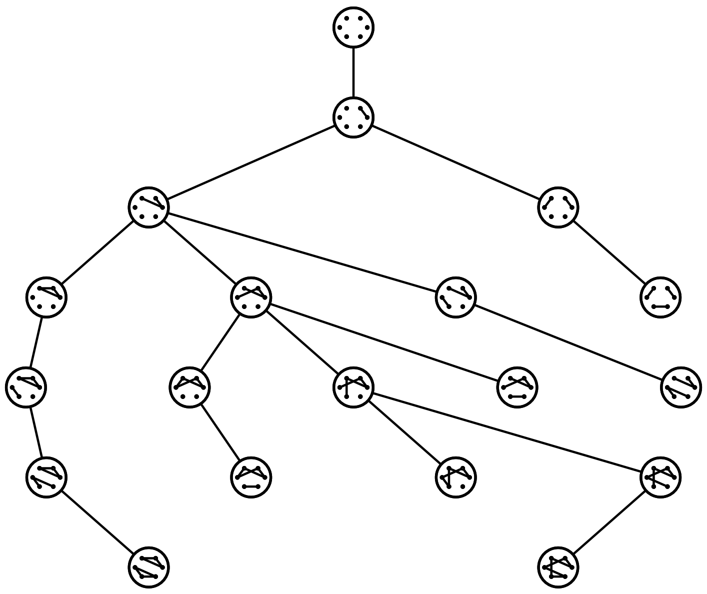


Figure 7: The tree of orbits for 2-regular graphs on 6 vertices

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/APPS` such as

```
ORBITER/SRC/APPS/BLT
ORBITER/SRC/APPS/GRAPH
ORBITER/SRC/APPS/HYPEROVAL
ORBITER/SRC/APPS/TOOLS
ORBITER/SRC/APPS/TRANSLATION_PLANE
```

The library itself comes in six pieces, forming a hierarchy of layers. The bottom layer is GALOIS. On top of this is ACTION. On top of this is SNAKES_AND_LADDERS. On top of this is INCIDENCE. On top of this is DISCRETA. Finally, the highest layer is TOP_LEVEL. Thus, the layers listed 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
```

It suffices to include the file

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

which will include all six include files previously mentioned in the appropriate order. By using makefiles to set compiler options that list the search path, the include command in an application would simply be

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

6 Contributors

The following list acknowledges people who have contributed code to `orbiter`.

Who	What	Description
Brendan McKay	Nauty	Program to compute a canonical form of graphs
Paul Hsieh	Super fast hash	Computing hash values
Xi Chen	DLX	Implementation of Don Knuth's dancing links algorithm

7 Getting Started

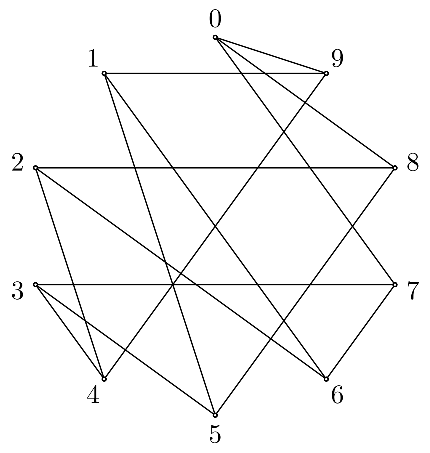
7.1 Johnson Graphs

The Johnson graph $J(n, k, s)$ is defined as follows. Vertices are the k-subsets of an n-element set. Two vertices are connected if the associated sets intersect in exactly s elements. Here is a program to create the Johnson graphs:

```
// johnson.C
//
// Anton Betten
// January 20, 2015

#include "orbiter.h"


int main(int argc, char **argv)
{
   INT verbose_level = 0;
   INT i, j, N, sz;
   INT *Adj;
   INT *set1;
   INT *set2;
   INT *set3;
   INT f_n = FALSE;
   INT n;
   INT f_k = FALSE;
   INT k;
   INT f_s = FALSE;
   INT s;

   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-n") == 0) {
         f_n = TRUE;
         n = atoi(argv[++i]);
         cout << "-n " << n << endl;
         }
      else if (strcmp(argv[i], "-k") == 0) {
         f_k = TRUE;
         k = atoi(argv[++i]);
         cout << "-k " << k << endl;
         }
      else if (strcmp(argv[i], "-s") == 0) {
         f_s = TRUE;
         s = atoi(argv[++i]);
         cout << "-s " << s << endl;
         }
      }

   if (!f_n) {
      cout << "Please use option -n <n>" << endl;
      exit(1);
      }
   if (!f_k) {
      cout << "Please use option -k <k>" << endl;
      exit(1);
      }
   if (!f_s) {
      cout << "Please use option -s <s>" << endl;
      exit(1);
      }

   N = INT_n_choose_k(n, k);


   Adj = NEW_INT(N * N);
   INT_vec_zero(Adj, N * N);

   set1 = NEW_INT(k);
   set2 = NEW_INT(k);
   set3 = NEW_INT(k);

   for (i = 0; i < N; i++) {
      unrank_k_subset(i, set1, n, k);
      for (j = i + 1; j < N; j++) {
```

```
            unrank_k_subset(j, set2, n, k);

            INT_vec_intersect_sorted_vectors(set1, k, set2, k, set3, sz);
            if (sz == s) {
               Adj[i * N + j] = 1;
               Adj[j * N + 1] = 1;
               }
            }
         }

   colored_graph *CG;
   BYTE fname[1000];

   CG = new colored_graph;
   CG->init_adjacency_no_colors(N, Adj, verbose_level);

   sprintf(fname, "Johnson_%ld_%ld_%ld.colored_graph", n, k, s);

   CG->save(fname, verbose_level);

   delete CG;
   FREE_INT(Adj);
   FREE_INT(set1);
   FREE_INT(set2);
   FREE_INT(set3);
}
```

Here is a drawing of the Johnson graph $J(5, 2, 0)$. This graph is better known as the Petersen graph.



The vertices of the graph are in correspondence to the subsets of a set. The subsets are listed in the lexicographical order. Here is a program to create the subsets of size k of $1, \ldots, n$ in lexicographic order.

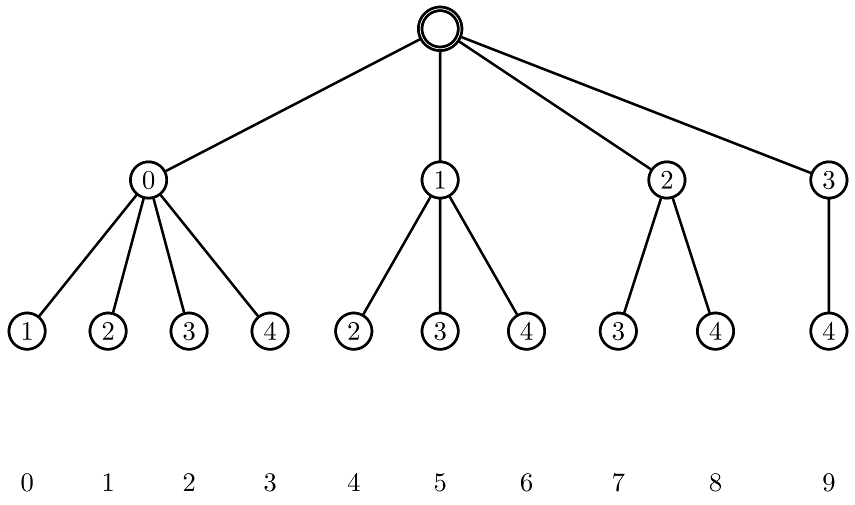
```
// all_k_subsets.C
//
// Anton Betten
// January 28, 2015

#include "orbiter.h"
```

```
int main(int argc, char **argv)
{
   INT verbose_level = 0;
   INT i, h;
   INT f_n = FALSE;
   INT n;
   INT f_k = FALSE;
   INT k;
   INT *set;
   INT N;
   BYTE fname[1000];

   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-n") == 0) {
         f_n = TRUE;
         n = atoi(argv[++i]);
         cout << "-n " << n << endl;
         }
      else if (strcmp(argv[i], "-k") == 0) {
         f_k = TRUE;
         k = atoi(argv[++i]);
         cout << "-k " << k << endl;
         }
      }

   if (!f_n) {
      cout << "Please use option -n <n>" << endl;
      exit(1);
      }
   if (!f_k) {
      cout << "Please use option -k <k>" << endl;
      exit(1);
      }

   sprintf(fname, "all_k_subsets_%ld_%ld.tree", n, k);
   set = NEW_INT(k);
   N = INT_n_choose_k(n, k);


   {
   ofstream fp(fname);

   for (h = 0; h < N; h++) {
      unrank_k_subset(h, set, n, k);
      fp << k;
      for (i = 0; i < k; i++) {
         fp << " " << set[i];
         }
      fp << endl;
      }
   fp << "-1" << endl;
   }
   FREE_INT(set);
}
```

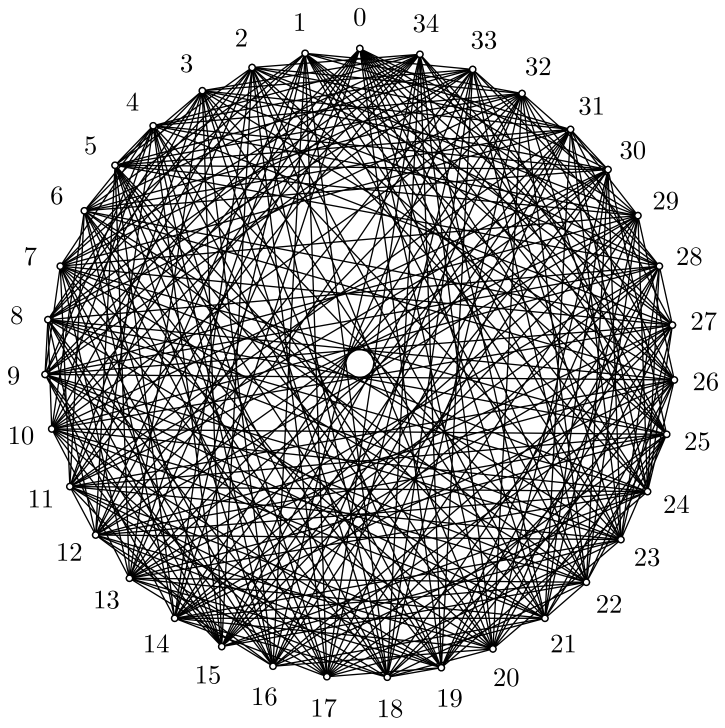
The subsets are written to a file:

```
2 0 1
2 0 2
2 0 3
2 0 4
2 1 2
2 1 3
2 1 4
2 2 3
2 2 4
2 3 4
-1
```

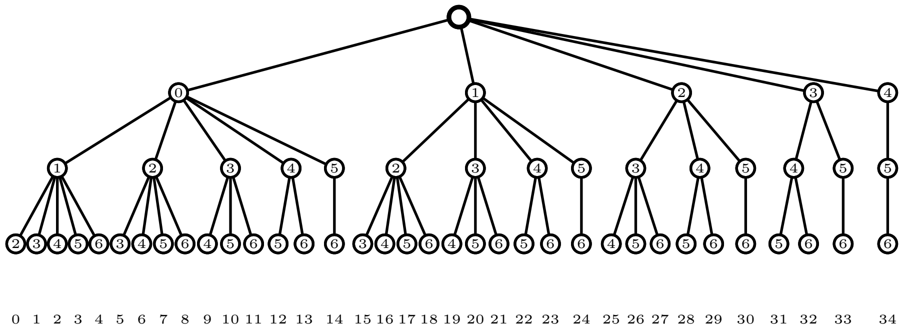
This is a drawing of the tree of subsets:



Here is $J(7,3,1)$



This is the tree of 3-subsets of $\{1,\ldots,7\}$:



7.2 Finite Fields

Suppose we want to have a finite field of order q. The `finite_field` class implements a lot of algorithms for finite fields. Let us postpone the discussion of how the field is implemented and look at an example first. Suppose we want the field with 4 elements.

```
// finite_field.C
//

#include "orbiter.h"
```

```
int main(int argc, char **argv)
{
   finite_field *F;
   INT verbose_level = 0;
   INT i, j, a;
   INT q = 4;

   F = new finite_field;
   F->init(q, verbose_level);

   cout << "addition table of GF(" << q << "):" << endl;
   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {

         a = F->add(i, j);

         cout << setw(2) << a << " ";
         }
      cout << endl;
      }

   cout << "multiplication table of GF(" << q << "):" << endl;
   for (i = 1; i < 4; i++) {
      for (j = 1; j < 4; j++) {

         a = F->mult(i, j);

         cout << setw(2) << a << " ";
         }
      cout << endl;
      }

   delete F;
}
```

The line `finite_field *F;` defines a pointer to an object `F` of type `finite_field`. The line `F = new finite_field;` allocates the object. The line `F->init(q, verbose_level);` initializes the finite field of order `q` (here, 4). The second parameter `verbose_level` is indicating how much the initialization function should print. Higher numbers give more print. Passing zero means do not print at all. The program produces the following output:

```
addition table of GF(4):
 0  1  2  3
 1  0  3  2
 2  3  0  1
 3  2  1  0
multiplication table of GF(4):
 1  2  3
 2  3  1
 3  1  2
```

If we change `verbose_level` to 1 and run the program again, we find the following output:

```
finite_field::init
finite_field::init_override_polynomial
finite_field::init_override_polynomial using poly 7
finite_field::init_override_polynomial() GF(4) = GF(2^2), polynomial = X^{2} + X + 1 = 7
finite_field::init finished
addition table of GF(4):
 0  1  2  3
 1  0  3  2
 2  3  0  1
 3  2  1  0
multiplication table of GF(4):
 1  2  3
 2  3  1
 3  1  2
```

This reveals the polynomial that is used to create the finite field. In this example, we use the polynomial $x^2 + x + 1$. It is very easy to use a different polynomial. If we want to create the finite field $\mathbb{F}_q$ over $\mathbb{F}_p$, we require an irreducible polynomial $m(x) = \sum_{i=0}^{n} a_i x^i \in \mathbb{F}_p[x]$ with $a_i \in \mathbb{F}_p$. Using the convention that $0 \le a_i < p$ for all i, we can define the integer $m(p) = \sum_{i=0}^{n} a_i p^i$. The function

```
    finite_field::init_override_polynomial(INT q,
    const BYTE *poly, INT verbose_level);
```

can be used to define a finite field using the polynomial $m(p)$. Here, `poly` points to a string which represents m in decimal notation. So, for instance, `init_override_polynomial(4, "7", verbose_level)` creates the same field as before, using the polynomial $x^2 + x + 1$, since $7 = 2^2 + 2 + 1$.

How are the elements of the finite field encoded? If $q = p$ is prime, the elements of $\mathbb{F}_p$ are the integers $\{0, 1, \ldots, p-1\}$. If $q = p^h$ for some prime p, and some integer $h > 1$, the assumption is made that the polynomial $m(x) \in \mathbb{F}_p[x]$ used to create $\mathbb{F}_q$ is primitive. That is, a root α of $m(x)$ is a generator for the multiplicative group $\mathbb{F}_q^{\times}$. Every element in $\mathbb{F}_q$ is of the form $w(\alpha)$ where $w(x) = \sum_{i=0}^{h-1} w_i x^i \in \mathbb{F}_p[x]$ is a polynomial of degree at most $h-1$. The element $w(\alpha)$ is represented by the integer $w(p) = \sum_{i=0}^{n} w_i p^i$. Thus 0 is represented by 0 and 1 is represented by 1. The primitive element α is represented by p.

7.3 Paley Graphs

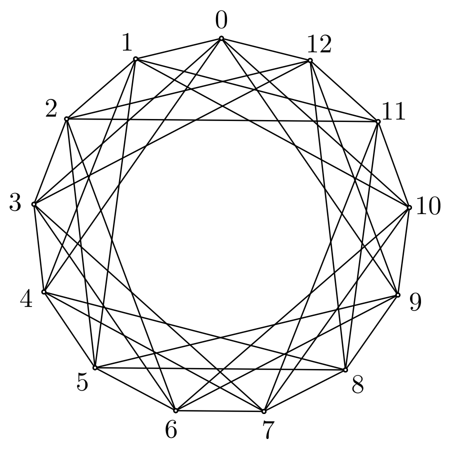
Another class of graphs are the Paley graphs. A Paley graph can be constructed using a finite field of order q, where q is an odd prime power. In addition, we require that q is congruent 1 modulo 4 (otherwise, we would create a tournament). The vertices of the Paley graph are the q elements of the field $\mathbb{F}_q$. Two vertices i and j are adjacent if $i - j$ is a nonzero square in $\mathbb{F}_q$. Here is a program to create Paley graphs of order q:

```
// paley.C
//
// Anton Betten
// January 19, 2015

#include "orbiter.h"
```

```
int main(int argc, char **argv)
{
   INT verbose_level = 0;
   finite_field *F;
   INT i, j, a;
   INT *Adj;
   INT f_q = FALSE;
   INT q;
   INT *f_is_square;

   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-q") == 0) {
         f_q = TRUE;
         q = atoi(argv[++i]);
         cout << "-q " << q << endl;
         }
      }

   if (!f_q) {
      cout << "Please use option -q <q>" << endl;
      exit(1);
      }

   if (EVEN(q)) {
      cout << "q must be odd" << endl;
      exit(1);
      }
   if (!DOUBLYEVEN(q - 1)) {
      cout << "q must be congruent to 1 modulo 4" << endl;
      }

   F = new finite_field;
   F->init(q, verbose_level);

   f_is_square = NEW_INT(q);
   INT_vec_zero(f_is_square, q);

   for (i = 0; i < q; i++) {
      j = F->mult(i, i);
      f_is_square[j] = TRUE;
      }

   Adj = NEW_INT(q * q);
   INT_vec_zero(Adj, q * q);

   for (i = 0; i < q; i++) {
      for (j = i + 1; j < q; j++) {
         a = F->add(i, F->negate(j));
         if (f_is_square[a]) {
            Adj[i * q + j] = 1;
```

```
                Adj[j * q + 1] = 1;
                }
            }
        }


    colored_graph *CG;
    BYTE fname[1000];

    CG = new colored_graph;
    CG->init_adjacency_no_colors(q, Adj, verbose_level);

    sprintf(fname, "Paley_%ld.colored_graph", q);

    CG->save(fname, verbose_level);

    delete CG;
    FREE_INT(Adj);
    FREE_INT(f_is_square);
    delete F;
}
```

Here is a drawing of the Paley graph created for $q = 13$:



7.4 Winnie Li Graphs

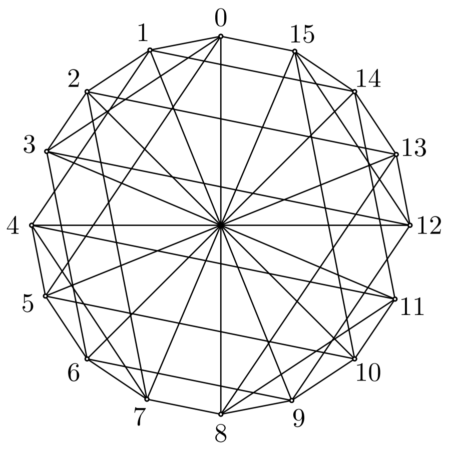
Another family of graphs was introduced by Winnie Li in the 1991 paper *Character Sums and Abelian Ramanujan Graphs*. Here is a program to create these graphs. The construction relies on the norm map from $\mathbb{F}_q$ over a subfield. The graph is a Cayley graph $\text{Cay}(G, S)$ where G is the additive group of $\mathbb{F}_q$ and S is the elements whose relative norm with respect to the subfield is one.

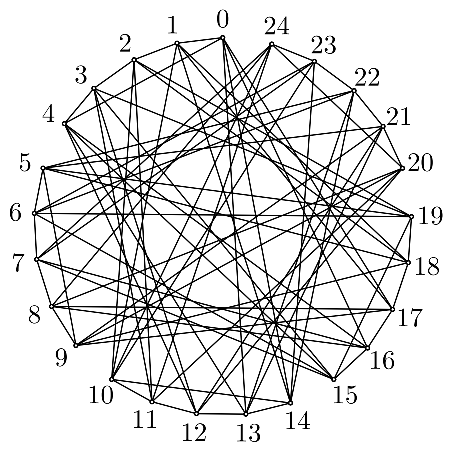
```
// winnie_li.C
//
// Anton Betten
// November 11, 2011
//
```

```
// creates the graphs described in the paper:
// Wen Ching Winnie Li: Character Sums and Abelian Ramanujan Graphs 1991

#include "orbiter.h"


INT t0;

void do_it(INT q, INT f_index, INT index, INT verbose_level);


int main(int argc, char **argv)
{
   INT i;
   INT verbose_level = 0;
   INT f_q = FALSE;
   INT q = 0;
   INT f_index = FALSE;
   INT index = 0;

   t0 = os_ticks();


   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-q") == 0) {
         f_q = TRUE;
         q = atoi(argv[++i]);
         cout << "-q " << q << endl;
         }
      else if (strcmp(argv[i], "-index") == 0) {
         f_index = TRUE;
         index = atoi(argv[++i]);
         cout << "-index " << index << endl;
         }
      }



   if (!f_q) {
      cout << "please specify -q <q>" << endl;
      exit(1);
      }

   do_it(q, f_index, index, verbose_level);

}

void do_it(INT q, INT f_index, INT index, INT verbose_level)
{
   //INT f_v = (verbose_level >= 1);
   finite_field *F;
   INT i, j, h, u, p, k, co_index, q1, relative_norm;
   INT *N1;
   INT *Adj;


   F = new finite_field;
   F->init(q, verbose_level - 1);
   p = F->p;

   if (!f_index) {
      index = F->e;
      }

   co_index = F->e / index;

   if (co_index * index != F->e) {
      cout << "the index has to divide the field degree" << endl;
      exit(1);
      }
   q1 = i_power_j(p, co_index);

   k = (q - 1) / (q1 - 1);

   cout << "q=" << q << endl;
   cout << "index=" << index << endl;
   cout << "co_index=" << co_index << endl;
   cout << "q1=" << q1 << endl;
   cout << "k=" << k << endl;

   relative_norm = 0;
   j = 1;
   for (i = 0; i < index; i++) {
      relative_norm += j;
      j *= q1;
      }
   cout << "relative_norm=" << relative_norm << endl;

   N1 = NEW_INT(k);
   j = 0;
   for (i = 0; i < q; i++) {
      if (F->power(i, relative_norm) == 1) {
         N1[j++] = i;
         }
      }
   if (j != k) {
      cout << "j != k" << endl;
      exit(1);
      }
   cout << "found " << k << " norm-one elements:" << endl;
   INT_vec_print(cout, N1, k);
   cout << endl;

   Adj = NEW_INT(q * q);
   for (i = 0; i < q; i++) {
      for (h = 0; h < k; h++) {
         j = N1[h];
         u = F->add(i, j);
```

```
          Adj[i * q + u] = 1;
          Adj[u * q + i] = 1;
          }
       }


   colored_graph *CG;
   BYTE fname[1000];

   CG = new colored_graph;
   CG->init_adjacency_no_colors(q, Adj, verbose_level);

   if (f_index) {
      sprintf(fname, "Winnie_Li_%ld_%ld.colored_graph", q, index);
      }
   else {
      sprintf(fname, "Winnie_Li_%ld.colored_graph", q);
      }

   CG->save(fname, verbose_level);






   delete CG;
   FREE_INT(Adj);
   FREE_INT(N1);
   delete F;


}
```

The graph for $q = 9$ is the Paley graph. Here are the graphs created for $q = 16$ (left) and $q = 25$ (right).



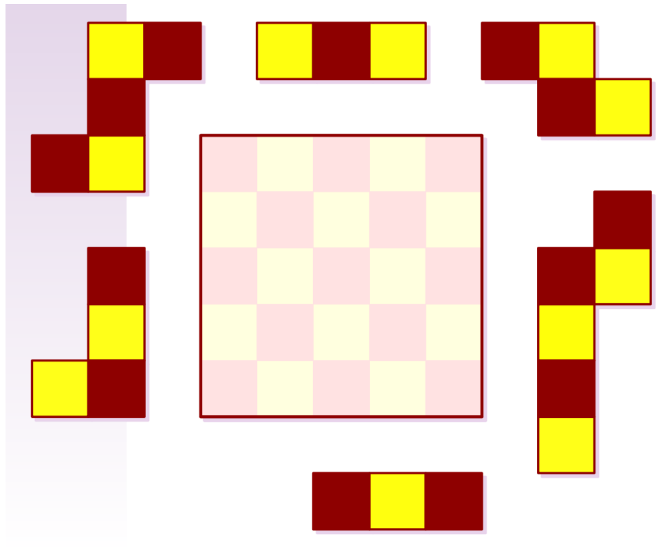


7.5 Solving Puzzles

On `puzzles.com`, we find the following checkerboard puzzle:

> Rearrange the six pieces so that to form the 5×5 checkerboard shown in the center of the illustration.

The following illustration goes along with it:



We wish to write a program to solve this puzzle for us. The way we approach this problem is the following. We wish to express the required conditions as a system of equation. The variables in the system correspond to the pieces in the various positions. The equations in the system capture the condition that each spot on the checkerboard is occupied by exactly one piece and that each piece is used exactly once. We label the 25 spots using the integers 0 through 24 in the following way:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

The conditions lead to a system of equations

$$A\mathbf{x} = \mathbf{1}$$

where A is a matrix over $\{0, 1\}$ and $\mathbf{x}$ is a $\{0, 1\}$-vector. The right hand side is the all-one vector. A piece in a certain position will be identified with the spots that it covers. That way, pieces in positions become subsets of the set $\{0, \ldots, 24\}$. In order to obtain all possible positions of a piece, we start by putting the piece in the upper left corner of the board, making sure that the colors match. Once this is done, we record the translation vectors that are allowed. A translation vector is simply an integer that can be added to the set so that the resulting set is the set of spots occupied by the moved piece. A piece can also be rotated. We record the 4 rotations using the integers $\{0, 1, 2, 3\}$. Once all translations and rotations are defined, we set up the coefficient matrix A. The first 25 rows of A are the conditions that each spot on the board is occupied by exactly one piece. Then, we add 6 more rows to A, each row corresponding to an equation that ensures that a certain piece is chosen exactly once. Once the system A has been computed, we store it on file. Here is the code:

```
// puzzle.C
//
// Anton Betten
// September 9, 2014
//

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;


int main(void)
{
   int S1[] = {1,2,6,10,11,-1};
   int S2[] = {2,7,11,12,-1};
   int S3[] = {1,2,3,-1};
   int S4[] = {0,1,2,-1};
   int S5[] = {0,1,6,7,-1};
   int S6[] = {2,6,7,11,16,21,-1};
   int *S[6];
   int S_length[6];

   int T1[] = {0,2,6,10,12,-1};
   int T2[] = {0,2,4,6,10,12,-1};
   int T3[] = {0,4,6,10,14,16,20,-1};
   int T4[] = {0,2,6,10,12,16,20,22,-1};
   int T5[] = {0,2,6,10,12,16,-1};
   int T6[] = {0,2,-1};
   int *T[6];
   int T_length[6];

   int R1[] = {0,1,-1};
   int R2[] = {0,1,2,3,-1};
   int R3[] = {0,1,-1};
   int R4[] = {0,1,-1};
   int R5[] = {0,1,2,3,-1};
   int R6[] = {0,1,2,3,-1};
   int *R[6];
   int R_length[6];

   int rotate[4 * 25];

   int i, j, ii, jj, h;
   int *M;


   S[0] = S1;
   S[1] = S2;
   S[2] = S3;
   S[3] = S4;
   S[4] = S5;
   S[5] = S6;
   T[0] = T1;
   T[1] = T2;
```

```
T[2] = T3;
T[3] = T4;
T[4] = T5;
T[5] = T6;
R[0] = R1;
R[1] = R2;
R[2] = R3;
R[3] = R4;
R[4] = R5;
R[5] = R6;
for (i = 0; i < 6; i++) {
   for (j = 0; ; j++) {
      if (S[i][j] == -1) {
         S_length[i] = j;
         break;
         }
      }
   for (j = 0; ; j++) {
      if (T[i][j] == -1) {
         T_length[i] = j;
         break;
         }
      }
   for (j = 0; ; j++) {
      if (R[i][j] == -1) {
         R_length[i] = j;
         break;
         }
      }
   }
for (i = 0; i < 5; i++) {
   for (j = 0; j < 5; j++) {
      rotate[0 * 25 + i * 5 + j] = i * 5 + j;
      }
   }
for (i = 0; i < 5; i++) {
   jj = 4 - i;
   for (j = 0; j < 5; j++) {
      ii = j;
      rotate[1 * 25 + i * 5 + j] = ii * 5 + jj;
      }
   }
for (i = 0; i < 25; i++) {
   rotate[2 * 25 + i] = rotate[1 * 25 + rotate[1 * 25 + i]];
   }
for (i = 0; i < 25; i++) {
   rotate[3 * 25 + i] = rotate[2 * 25 + rotate[1 * 25 + i]];
   }

cout << "rotate:" << endl;
for (h = 0; h < 4; h++) {
   for (j = 0; j < 25; j++) {
      cout << setw(3) << rotate[h * 25 + j] << " ";
      }
   cout << endl;
   }
```

```
int var_start[6 + 1];
int var_length[6 + 1];

var_start[0] = 0;
for (h = 0; h < 6; h++) {
   var_length[h] = R_length[h] * T_length[h];
   var_start[h + 1] = var_start[h] + var_length[h];
   }
cout << "i : var_start[i] : var_length[i]" << endl;
for (h = 0; h < 6; h++) {
   cout << h << " : " << var_start[h] << " : " << var_length[h] << endl;
   }

int nb_eqns;
int nb_vars;
int nb_eqn1;
int nb_eqn2;

nb_vars = var_start[6];
nb_eqn1 = 5 * 5;
nb_eqn2 = 6;
nb_eqns = nb_eqn1 + nb_eqn2;

cout << "nb_vars=" << nb_vars << endl;
cout << "nb_eqn1=" << nb_eqn1 << endl;
cout << "nb_eqn2=" << nb_eqn2 << endl;
cout << "nb_eqns=" << nb_eqns << endl;

M = new int[nb_eqns * nb_vars];
for (i = 0; i < nb_eqns; i++) {
   for (j = 0; j < nb_vars; j++) {
      M[i * nb_vars + j] = 0;
      }
   }

int j0, r, t, rr, tt, s, x, y, z;

for (h = 0; h < 6; h++) {
   j0 = var_start[h];

   cout << "h=" << h << "/" << 6 << " j0=" << j0 << ":" << endl;
   for (r = 0; r < R_length[h]; r++) {
      rr = R[h][r];
      cout << "h=" << h << "/" << 6 << " r=" << r << "/" << R_length[h] << " rr=" << rr << ":" << endl;
      for (t = 0; t < T_length[h]; t++) {
         tt = T[h][t];
         cout << "h=" << h << "/" << 6 << " r=" << r << "/" << R_length[h] << " rr=" << rr << " t=" << t << "/"
         for (s = 0; s < S_length[h]; s++) {
            x = S[h][s];
            y = x + tt;
            z = rotate[rr * 25 + y];

            cout << "h=" << h << "/" << 6 << " r=" << r << "/" << R_length[h] << " rr=" << rr << " t=" << t << "
            M[z * nb_vars + j0 + r * T_length[h] + t] = 1;
            }
         }
      }
   }
for (h = 0; h < 6; h++) {
   j0 = var_start[h];

   for (r = 0; r < R_length[h]; r++) {
      for (t = 0; t < T_length[h]; t++) {
         M[(nb_eqn1 + h) * nb_vars + j0 + r * T_length[h] + t] = 1;
         }
      }
   }

for (i = 0; i < nb_eqns; i++) {
   for (j = 0; j < nb_vars; j++) {
      cout << M[i * nb_vars + j];
      }
   cout << endl;
   }

{
const char *fname = "puzzle25.txt";
ofstream fp(fname);
fp << nb_eqns << " " << nb_vars << endl;
for (i = 0; i < nb_eqns; i++) {
   for (j = 0; j < nb_vars; j++) {
      fp << M[i * nb_vars + j];
      if (j < nb_vars - 1) {
         fp << " ";
         }
      }
   fp << endl;
   }
fp << "END" << endl;
}
{
const char *fname = "puzzle25_compressed.txt";
ofstream fp(fname);
int d;
fp << nb_eqns << " " << nb_vars << endl;
for (i = 0; i < nb_eqns; i++) {
   d = 0;
   for (j = 0; j < nb_vars; j++) {
      if (M[i * nb_vars + j]) {
         d++;
         }
      }
   fp << d;
   for (j = 0; j < nb_vars; j++) {
      if (M[i * nb_vars + j]) {
         fp << " " << j;
         }
      }
   fp << endl;
   }
fp << "END" << endl;
```

```
    }

}
```

Here is the output of the program (slightly cut in order to save space). At the bottom, we see the matrix A.

```
rotate:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24
  4   9  14  19  24   3   8  13  18  23   2   7  12  17  22   1   6  11  16  21   0   5  10  15  20
 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
 20  15  10   5   0  21  16  11   6   1  22  17  12   7   2  23  18  13   8   3  24  19  14   9   4
i : var_start[i] : var_length[i]
0 : 0 : 10
1 : 10 : 24
2 : 34 : 14
3 : 48 : 16
4 : 64 : 24
5 : 88 : 8
nb_vars=96
nb_eqn1=25
nb_eqn2=6
nb_eqns=31
h=0/6 j0=0:
h=0/6 r=0/2 rr=0:
h=0/6 r=0/2 rr=0 t=0/5 tt=0:
h=0/6 r=0/2 rr=0 t=0/5 tt=0 s=0/5 x=1 y=1 z=1 entry=(1,0)
h=0/6 r=0/2 rr=0 t=0/5 tt=0 s=1/5 x=2 y=2 z=2 entry=(2,0)
h=0/6 r=0/2 rr=0 t=0/5 tt=0 s=2/5 x=6 y=6 z=6 entry=(6,0)
h=0/6 r=0/2 rr=0 t=0/5 tt=0 s=3/5 x=10 y=10 z=10 entry=(10,0)
h=0/6 r=0/2 rr=0 t=0/5 tt=0 s=4/5 x=11 y=11 z=11 entry=(11,0)
h=0/6 r=0/2 rr=0 t=1/5 tt=2:
h=0/6 r=0/2 rr=0 t=1/5 tt=2 s=0/5 x=1 y=3 z=3 entry=(3,1)

output removed

000000001000000000000000000101000000000000000000000100000000000000101000000000000000000000000000001
100000000000000000010000000101000010000000000010010000000000000000100000000000000000101000000000101
100001000010000000000000000100100011000000000000001100000000010000010000000100000001000000010000000
010000000000000000000000000100000011000000001000000100000000000000001000000000000000000000001000001000
010000000000100000000000000000000001000000000000000010000001000000000000001000000000000000000001000000
000000001000000000000100000010000000100000000000100000000000000100000000000000000100100000000100000
100000011000100000010000001000001000100000000010000100000000000100101000000010100001001000010100101
001001001010000010100000011001010001100000001000001000000000100001010000000100000101001010101000 01
011001000000010000100000100000010000100000100000001000000010000000100001010000000101000010011010 01
000001000000100000000000010000000010010000100000000000000010000000001000010000000000000000101000001
100000000100000000010010001100000000000000000000010001000000000011000100000000100000001000000000010
100100010010100000011001010010000000001000000011000010000000000100000100000101001010101000100000110
011100111010001010001010001010001000010000000100000011000000110000010100010100010100010100000000 0
010010010001010010000010100000011000010000110000000010000010000000010101010000000100000101011010 00
000011000000100011000000000000000000100000000100000000000100011000000000000001000000010000000100100000
000000000100100000000101000000000000000100000000010000000000000000100000000000101000010000000001010 0
001100000010010000000100000100001000000001000000010000001000000010000010100001001000010100010010110
001000100100011001010010000010100000000110000100000000010000001000000101001010101000000010000011010
000010110000001000001000010000010000000010001000000000010000100000000010010000101000000010101011010
000000100000000010000000000000000010000001010000000000000000100000000001001000000000000000000000010
000100000000000000000101000000000000000000000000000000000001000000001000000000000000000001000000000100
000100000000000100000010000000000000000000100000100000001000000000000000000000010010000000000010000000
000010000100001001000110000000000000000000100000000000000110000100000000100000001000000010000001000
000010000000000010100000000000000100000000001001000000000000100000000000000101000010000000000000101000 0
000000100000000010100000000000000000000000000000000000001010000000000000000000100000000000000000010000
111111111100000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000011111111111111111111111100000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000011111111111111000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000111111111111111100000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000011111111111111111111111100000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111111
```

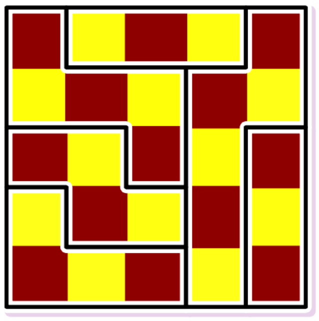
The file `puzzle25_compressed.txt` is this:

```
31 96
7 8 27 29 48 62 64 95
12 0 18 27 29 34 45 48 64 81 83 93 95
14 0 5 10 26 29 33 34 48 49 59 65 73 81 88
9 1 26 33 34 42 49 65 86 92
7 1 11 33 49 56 70 89
9 8 20 27 35 47 62 80 83 90
21 0 7 8 12 18 25 31 35 45 50 61 64 66 73 75 80 83 88 90 93 95
25 2 5 8 10 16 18 25 26 29 31 35 36 44 50 59 64 66 73 79 81 84 86 88 90 95
21 1 2 5 13 18 24 31 36 42 50 58 65 70 72 79 81 86 89 90 92 95
12 5 11 24 33 36 41 56 65 70 87 89 95
14 0 9 20 23 27 28 47 51 62 63 67 75 83 94
25 0 3 7 10 12 19 20 23 25 28 37 45 46 51 61 67 73 75 78 80 82 84 88 93 94
28 1 2 3 6 7 8 10 14 16 20 22 26 28 32 37 44 51 52 59 60 66 68 72 74 78 80 84 86
25 1 4 7 11 13 16 22 24 31 32 37 42 43 52 58 66 68 70 72 79 85 87 89 90 92
14 4 5 11 15 16 32 41 52 56 57 71 79 87 90
12 9 12 21 23 38 47 63 75 77 82 91 93
21 2 3 9 12 19 25 30 38 46 53 61 67 69 74 77 82 84 88 91 93 94
25 2 6 9 13 14 17 19 22 28 30 38 39 44 53 60 67 69 72 74 76 78 85 91 92 94
21 4 6 7 13 19 24 30 39 43 53 58 68 71 76 78 85 87 89 91 92 94
9 6 15 32 39 41 57 68 71 94
7 3 21 23 54 63 82 93
9 3 14 21 40 46 54 74 77 88
14 4 9 14 17 21 22 40 54 55 60 69 77 85 92
12 4 15 17 30 40 43 55 69 71 76 89 91
7 6 15 17 55 57 76 91
10 0 1 2 3 4 5 6 7 8 9
24 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
14 34 35 36 37 38 39 40 41 42 43 44 45 46 47
16 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
24 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
8 88 89 90 91 92 93 94 95
END
```

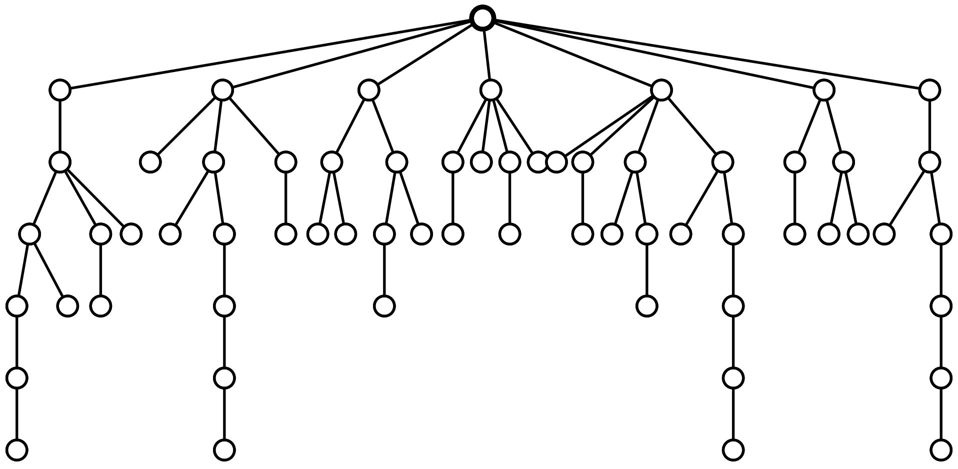
The program **dlx.out** takes this file and solves the associated system $A\mathbf{x} = \mathbf{1}$. The solutions are written to yet another file. There are exactly four solutions:

```
6 8 34 21 67 57 89
6 27 54 91 41 73 1
6 62 93 40 6 79 33
6 95 49 47 3 85 15
-1 4 63
```

The four solutions are essentially all one and the same solution. This one solution appears in all 4 different rotations.



The program `dlx.out` performs a backtrack search. The search tree looks as follows. Each node corresponds to one state of the variable `x` that is examined.



The four deep branches correspond to the four solutions. Altogether, the search examines 63 nodes.

7.6 Finite Projective Space

Suppose we want to work in n-dimensional projective space over a finite field. We can use the `finite_field` object from before to provide the points in PG(n, q). In this section, we will look at how we can work with set set of points of $PG(n, q)$. The idea is that we try to avoid forming the set of points, since this might be a very large set. Instead, we want to access points whenever we need them. Accessing points means writing down homogeneous coordinates. So, for instance, if we know that there are N points in PG(n, q), then we would like to have access to these points through the integers 0, 1, . . . , $N - 1$. Also, if we are given a point by means of homogeneous coordinates, we would like to identify the number from 0 to $N - 1$ that represents the point. The bijection between the set $0, \dots, N - 1$ and the vectors of homogeneous coordinates of points in PG(n, q) is an example of a **rank function**. A rank funktion always comes together with its inverse function, which is known as an *unrank function.* So, given an integer, the unrank function creates the homogeneous coordinates of the assocated point. Conversely, given homogeneous coordinates for a point, the rank function computes the associated integer. The functions to rank and unrank points in projective space are called `rank_point_in_PG` and `unrank_point_in_PG`. These functions are member functions of the class `finite_field`. Here is a code example illustrating the ranking and unranking of projective points. We start by creating the finite field $\mathbb{F}_q$. Then we compute N, the number of points in PG(n, q). We then loop over all integers i from 0 to $N - 1$. For a given integer i, we unrank the homogeneous coordinates of the i-th point. We immediately rank the homogeneous coordinates again to see if we get the same integer i:

```
// projective_space.C
//

#include "orbiter.h"


int main(int argc, char **argv)
{
   finite_field *F;
   INT verbose_level = 0;
   INT i, j, N;
   INT q = 4;
   INT n = 2; // projective dimension
   INT *v;

   F = new finite_field;
   F->init(q, verbose_level);


   N = F->nb_points_in_PG(n);

   v = NEW_INT(n + 1);
   for (i = 0; i < N; i++) {
      F->unrank_point_in_PG(v, n + 1, i);
      j = F->rank_point_in_PG(v, n + 1);
      cout << setw(3) << i << " : ";
      INT_vec_print(cout, v, n + 1);
      cout << " : " << j << endl;
      if (j != i) {
         cout << "j != i, this should not happen" << endl;
         exit(1);
         }
      }
   FREE_INT(v);
   delete F;
}
```

Here is the output of the program:

```
  0 : ( 1, 0, 0 ) : 0
  1 : ( 0, 1, 0 ) : 1
  2 : ( 0, 0, 1 ) : 2
  3 : ( 1, 1, 1 ) : 3
  4 : ( 1, 1, 0 ) : 4
  5 : ( 2, 1, 0 ) : 5
  6 : ( 3, 1, 0 ) : 6
  7 : ( 1, 0, 1 ) : 7
  8 : ( 2, 0, 1 ) : 8
  9 : ( 3, 0, 1 ) : 9
 10 : ( 0, 1, 1 ) : 10
 11 : ( 2, 1, 1 ) : 11
 12 : ( 3, 1, 1 ) : 12
 13 : ( 0, 2, 1 ) : 13
 14 : ( 1, 2, 1 ) : 14
 15 : ( 2, 2, 1 ) : 15
 16 : ( 3, 2, 1 ) : 16
 17 : ( 0, 3, 1 ) : 17
 18 : ( 1, 3, 1 ) : 18
 19 : ( 2, 3, 1 ) : 19
 20 : ( 3, 3, 1 ) : 20
```

The class `projective_space` provides a lot more functionality for projective spaces. For instance, we can access the lines in projective space. There are two ways to do so. For small spaces, it is convenient to precompute all lines and store the information in the form of tables. A line is stored as the set of points incident with it. This class provides access to points and lines, for instance. Each set contains the ranks of the $q + 1$ points that make up the particular line. The following program allocates and initializes a `projective_space` object and uses it to list the lines in $\mathrm{PG}(2, 4)$.

```
// projective_space2.C
//

#include "orbiter.h"


int main(int argc, char **argv)
{
   finite_field *F;
   projective_space *P;
   INT f_init_incidence_structure = TRUE;
   INT verbose_level = 0;
   INT i;
   INT q = 4;
   INT n = 2; // projective dimension

   F = new finite_field;
   F->init(q, verbose_level);


   P = new projective_space;
   P->init(n, F,
      f_init_incidence_structure,
      verbose_level);

   for (i = 0; i < P->N_lines; i++) {

      cout << "Line " << setw(3) << i << " is ";
      INT_vec_print(cout, P->Lines + i * P->k, P->k);
      cout << endl;
      }

   delete P;
   delete F;
}
```

Here is the output of the program:

```
Line 0 / 21:
Line   0 is ( 0, 1, 4, 5, 6 )
Line   1 is ( 0, 10, 3, 11, 12 )
Line   2 is ( 0, 17, 20, 18, 19 )
Line   3 is ( 0, 13, 15, 16, 14 )
Line   4 is ( 0, 2, 7, 8, 9 )
Line   5 is ( 7, 1, 3, 18, 14 )
Line   6 is ( 7, 10, 4, 16, 19 )
Line   7 is ( 7, 17, 15, 5, 12 )
Line   8 is ( 7, 13, 20, 11, 6 )
Line   9 is ( 4, 2, 3, 15, 20 )
Line  10 is ( 9, 1, 20, 16, 12 )
Line  11 is ( 9, 10, 15, 18, 6 )
Line  12 is ( 9, 17, 4, 11, 14 )
Line  13 is ( 9, 13, 3, 5, 19 )
Line  14 is ( 6, 2, 14, 19, 12 )
Line  15 is ( 8, 1, 15, 11, 19 )
Line  16 is ( 8, 10, 20, 5, 14 )
Line  17 is ( 8, 17, 3, 16, 6 )
Line  18 is ( 8, 13, 4, 18, 12 )
Line  19 is ( 5, 2, 18, 11, 16 )
Line  20 is ( 1, 2, 10, 13, 17 )
```

A second way to access lines is by means of generator matrices. In fact, this method is available for all subspaces of a fixed dimension. The subspaces of dimension k in $\mathbb{F}_q^n$ are in bijection to the set $0, \ldots, \left[{n \atop k}\right]_q - 1$ can be ranked and unranked. The class `grassmann` provides this functionality. The class `projective_space` contains one `grassmann` object to rank and unrank lines.

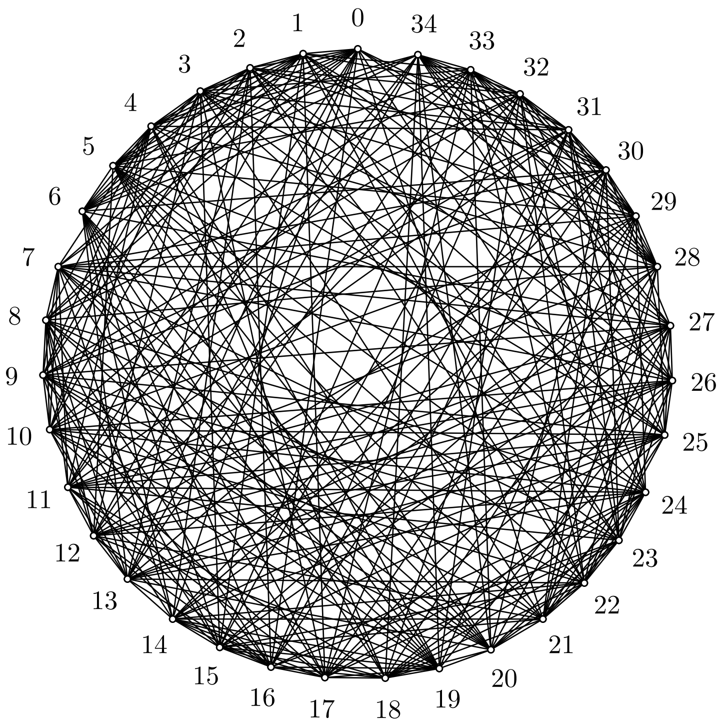
7.7 Grassmann Graphs

One class of graphs are the grassmann graphs. Fix n, k, q where $k \leq n$ and q is a prime power. The grassmann graph $G_{n,k,q}$ has as its vertices the k-dimensional subspaces of $\mathbb{F}_q^n$. Two vertices are connected by an edge if the intersection of the two associated subspaces has codimension one in each of the subspaces. In order to create this graph, we first initialize a `grassmann` object. The main purpose of the class `grassmann` is to privide rank and unrank functions for the set of k-dimensional subspaces of $\mathbb{F}_q^n$.

```
// grassmann.C
//

#include "orbiter.h"


int main(int argc, char **argv)
{
   finite_field *F;
   grassmann *Gr;
   INT verbose_level = 0;
   INT i, j, rr, N;
   INT *M1; // [k * n]
   INT *M2; // [k * n]
   INT *M; // [2 * k * n]
   INT *Adj;
   INT f_q = FALSE;
   INT q;
   INT f_k = FALSE;
   INT k = 0; // vector space dimension of subspaces
   INT f_n = FALSE;
   INT n = 0; // vector space dimension of whole space
   INT f_r = FALSE;
   INT r = 0;

   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-q") == 0) {
         f_q = TRUE;
         q = atoi(argv[++i]);
         cout << "-q " << q << endl;
         }
      else if (strcmp(argv[i], "-n") == 0) {
         f_n = TRUE;
         n = atoi(argv[++i]);
         cout << "-n " << n << endl;
         }
      else if (strcmp(argv[i], "-k") == 0) {
         f_k = TRUE;
         k = atoi(argv[++i]);
         cout << "-k " << k << endl;
         }
      else if (strcmp(argv[i], "-r") == 0) {
         f_r = TRUE;
         r = atoi(argv[++i]);
         cout << "-r " << r << endl;
         }
      }

   if (!f_q) {
      cout << "Please use option -q <q>" << endl;
      exit(1);
      }
   if (!f_n) {
      cout << "Please use option -n <n>" << endl;
      exit(1);
      }
   if (!f_k) {
      cout << "Please use option -k <k>" << endl;
      exit(1);
      }
   if (!f_r) {
      cout << "Please use option -r <r>" << endl;
      exit(1);
      }


   F = new finite_field;
   F->init(q, verbose_level);
```

```
   Gr = new grassmann;
   Gr->init(n, k, F, verbose_level);

   N = generalized_binomial(n, k, q);

   M1 = NEW_INT(k * n);
   M2 = NEW_INT(k * n);
   M = NEW_INT(2 * k * n);

   Adj = NEW_INT(N * N);
   INT_vec_zero(Adj, N * N);

   for (i = 0; i < N; i++) {

      Gr->unrank_INT_here(M1, i, 0 /* verbose_level */);

      for (j = i + 1; j < N; j++) {

         Gr->unrank_INT_here(M2, j, 0 /* verbose_level */);

         INT_vec_copy(M1, M, k * n);
         INT_vec_copy(M2, M + k * n, k * n);

         rr = F->rank_of_rectangular_matrix(M, 2 * k, n, 0 /* verbose_level */);
         if (rr == r) {
            Adj[i * N + j] = 1;
            Adj[j * N + i] = 1;
            }
         }
      }


   colored_graph *CG;
   BYTE fname[1000];

   CG = new colored_graph;
   CG->init_adjacency_no_colors(N, Adj, verbose_level);

   sprintf(fname, "grassmann_graph_%ld_%ld_%ld_%ld.colored_graph", n, k, q, r);

   CG->save(fname, verbose_level);




   delete CG;
   FREE_INT(M1);
   FREE_INT(M2);
   FREE_INT(M);
   delete Gr;
   delete F;
}
```

Here is a drawing of the graph created:



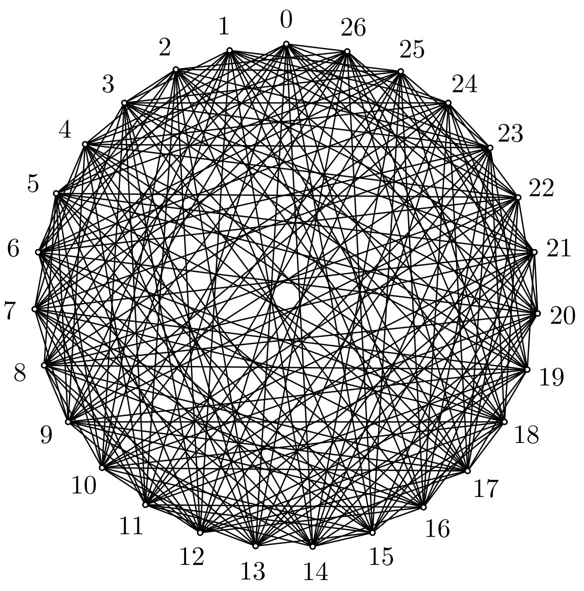
7.8 The Schlaefli Graph

The Schlaefli graph is defined as follows. Consider the variety $x^3 + y^3 + z^3 + w^3 = 0$ over PG(3, 4). Vertices are the 27 projective lines contained in the variety. Two vertices are adjacent if the associated lines are disjoint. Here is a program to create the Schlaefli graphs:

```
// schlaefli.C
//
// Anton Betten
// January 21, 2015

#include "orbiter.h"


INT evaluate_cubic_form(finite_field *F, INT *v);

int main(int argc, char **argv)
{
   INT verbose_level = 0;
   INT i, j, rr, sz, N;
   finite_field *F;
   grassmann *Gr;
   INT *Adj;
   INT *M1;
   INT *M2;
   INT *M;
INT v[2];
INT w[4];
INT *List;
INT f_q = FALSE;
INT q;
INT n = 4;
INT k = 2;

for (i = 1; i < argc; i++) {
   if (strcmp(argv[i], "-v") == 0) {
      verbose_level = atoi(argv[++i]);
      cout << "-v " << verbose_level << endl;
      }
   else if (strcmp(argv[i], "-q") == 0) {
      f_q = TRUE;
      q = atoi(argv[++i]);
      cout << "-q " << q << endl;
      }
   }

if (!f_q) {
   cout << "Please use option -q <q>" << endl;
   exit(1);
   }

F = new finite_field;
F->init(q, verbose_level);

Gr = new grassmann;
Gr->init(n, k, F, verbose_level);

M1 = NEW_INT(k * n);
M2 = NEW_INT(k * n);
M = NEW_INT(2 * k * n);

N = generalized_binomial(n, k, q);

List = NEW_INT(N);
sz = 0;

for (i = 0; i < N; i++) {
   Gr->unrank_INT_here(M1, i, 0 /* verbose_level */);

   for (j = 0; j < q + 1; j++) {
      F->unrank_point_in_PG(v, 2, j);
      F->mult_vector_from_the_left(v, M1, w, k, n);
      if (evaluate_cubic_form(F, w)) {
         break;
         }
      }
   if (j == q + 1) {
      List[sz++] = i;
      }
   }
cout << "We found " << sz << " lines" << endl;
```

```
   Adj = NEW_INT(sz * sz);
   INT_vec_zero(Adj, sz * sz);

   for (i = 0; i < sz; i++) {
      Gr->unrank_INT_here(M1, List[i], 0 /* verbose_level */);

      for (j = i + 1; j < sz; j++) {
         Gr->unrank_INT_here(M2, List[j], 0 /* verbose_level */);

         INT_vec_copy(M1, M, k * n);
         INT_vec_copy(M2, M + k * n, k * n);

         rr = F->rank_of_rectangular_matrix(M, 2 * k, n, 0 /* verbose_level */);
         if (rr == 2 * k) {
            Adj[i * sz + j] = 1;
            Adj[j * sz + i] = 1;
            }
         }
      }

   colored_graph *CG;
   BYTE fname[1000];

   CG = new colored_graph;
   CG->init_adjacency_no_colors(sz, Adj, verbose_level);

   sprintf(fname, "Schlaefli_%ld.colored_graph", q);

   CG->save(fname, verbose_level);

   delete CG;
   FREE_INT(List);
   FREE_INT(Adj);
   FREE_INT(M1);
   FREE_INT(M2);
   FREE_INT(M);
   delete Gr;
   delete F;
}

INT evaluate_cubic_form(finite_field *F, INT *v)
{
   INT a, i;

   a = 0;
   for (i = 0; i < 4; i++) {
      a = F->add(a, F->power(v[i], 3));
      }
   return a;
}
```

Here is a drawing of the Schlaefli graph.



7.9 Ramanujan Graphs

An interesting family of graphs was introduced by Lubotzky, Phillips and Sarnak in 1988. These are Ramanujan graphs. The graphs need a choice of two primes, p and q, both of which are required to be congruent 1 modulo 4. The graphs are created as Cayley graphs inside a certain group G. Let G be a group and let S be a subset of G. We require that S is closed under inverses. The cayley graph $\mathrm{Cay}(G, S)$ is defined as follows: The vertices of the graph are the elements of the group G. The edges are determined by the set S. Namely, a vertex g is adjacent to a vertex h if there is an element $s \in S$ with $h = gs$. Observe that since vertices are associated with elements in G, the equation $h = gs$ is computed inside the group G. The condition that S is closed under inverses means that $s^{-1} \in S$ also, and hence that h is adjacent to g since $hs^{-1} = g$. Thus, the edges in the Caley graph are well defined. The set S is called the connection set, since it gives the neighbors of the vertex associated to the identity element. Every other vertex looks that same. The reason for this is that the group G acts as a group of automorphisms on the graph. This action is vertex transitively. In the example described by Lubotzky, Phillips and Sarnak, the group G is either $\mathrm{PGL}(2, q)$ or $\mathrm{PSL}(2, q)$, depending on the value of the Legendre symbol for p and q. The connection set S is derived from the set of $p + 1$ solutions of $a_0^2 + a_1^2 + a_2^2 + a_3^2 = p$ over integers (a_0, a_1, a_2, a_3). The paper shows that each vertex is incident with $p + 1$ other vertices. The example shown here demonstrates that this is slightly incorrect. For $p = 29$ and $q = 5$, we get a graph where each vertex is incident with only 27 vertices.

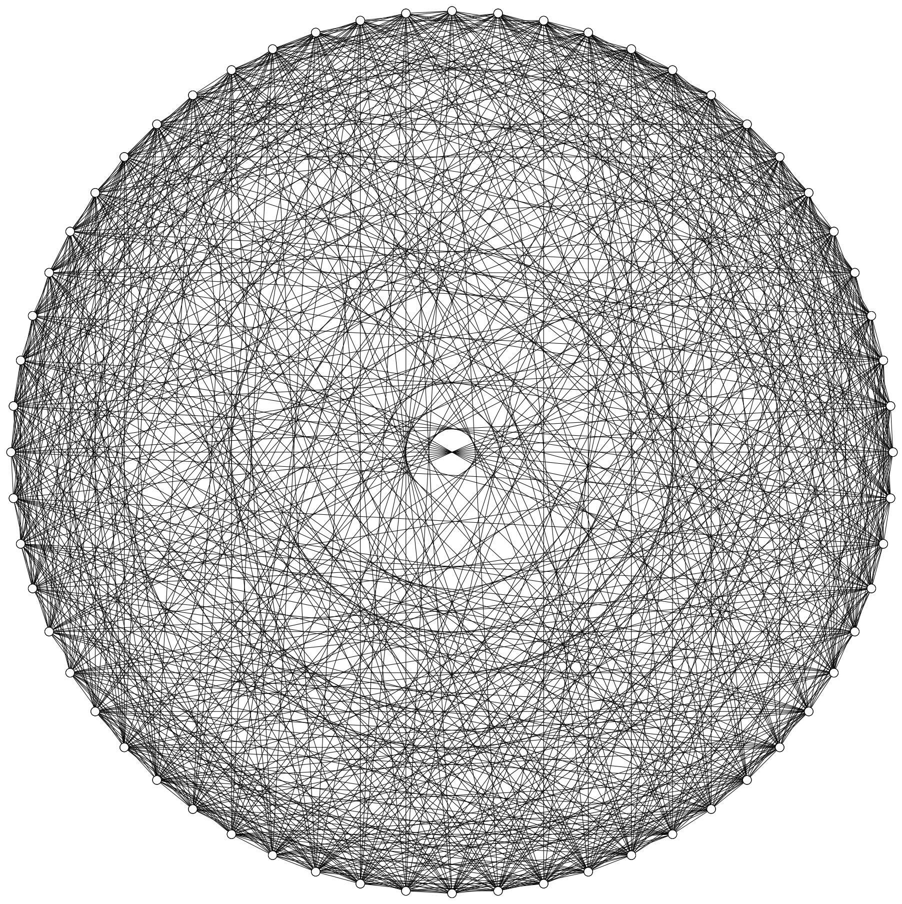
```
// sarnak.C
//
// Anton Betten
// November 2, 2011
//
// creates the graphs described in the paper:
//
// @article {MR963118,
//     AUTHOR = {Lubotzky, A. and Phillips, R. and Sarnak, P.},
//      TITLE = {Ramanujan graphs},
//    JOURNAL = {Combinatorica},
//  FJOURNAL = {Combinatorica. An International Journal of the J\´anos Bolyai
//              Mathematical Society},
//    VOLUME = {8},
//      YEAR = {1988},
//    NUMBER = {3},
//     PAGES = {261--277},
//      ISSN = {0209-9683},
//     CODEN = {COMBDI},
//   MRCLASS = {05C75 (05C25 05C50)},
//  MRNUMBER = {963118 (89m:05099)},
//MRREVIEWER = {Dave Witte Morris},
//       DOI = {10.1007/BF02126799},
//       URL = {http://0-dx.doi.org.catalog.library.colostate.edu/10.1007/BF02126799},
//}

#include "orbiter.h"


INT t0;

void do_it(INT p, INT q, INT verbose_level);



int main(int argc, char **argv)
{
   INT i;
   INT verbose_level = 0;
   INT f_p = FALSE;
   INT p = 0;
   INT f_q = FALSE;
   INT q = 0;

   t0 = os_ticks();


   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-p") == 0) {
         f_p = TRUE;
         p = atoi(argv[++i]);
         cout << "-p " << p << endl;
         }
      else if (strcmp(argv[i], "-q") == 0) {
         f_q = TRUE;
         q = atoi(argv[++i]);
         cout << "-q " << q << endl;
         }
      }



   if (!f_p) {
      cout << "please specify -p <p>" << endl;
      exit(1);
      }
   if (!f_q) {
      cout << "please specify -q <q>" << endl;
      exit(1);
      }

   do_it(p, q, verbose_level);

}

void do_it(INT p, INT q, INT verbose_level)
{
   INT f_v = (verbose_level >= 1);
   INT f_vv = (verbose_level >= 2);
   INT i, j, h, l, f_special = FALSE;



   l = Legendre(p, q, 0);
   cout << "Legendre(" << p << ", " << q << ")=" << l << endl;


   finite_field *F;
   action *A;
   INT f_semilinear = FALSE;
   INT f_basis = TRUE;

   F = new finite_field;
   F->init(q, 0);
   //F->init_override_polynomial(q, override_poly, verbose_level);

   A = new action;

   if (l == 1) {
      f_special = TRUE;

      cout << "Creating projective special linear group:" << endl;
      A->init_projective_special_group(2, F,
         f_semilinear,
         f_basis,
         verbose_level - 2);
      }
   else {
      cout << "Creating projective linear group:" << endl;
      A->init_projective_group(2, F,
         f_semilinear,
         f_basis,
         verbose_level - 2);
      }



   sims *Sims;
```

```
Sims = A->Sims;


longinteger_object go;
INT goi;
Sims->group_order(go);

cout << "found a group of order " << go << endl;
goi = go.as_INT();
cout << "found a group of order " << goi << endl;




INT a0, a1, a2, a3;
INT sqrt_p;

INT *sqrt_mod_q;
INT I;
INT *A4;
INT nb_A4 = 0;

A4 = NEW_INT((p + 1) * 4);
sqrt_mod_q = NEW_INT(q);
for (i = 0; i < q; i++) {
   sqrt_mod_q[i] = -1;
   }
for (i = 0; i < q; i++) {
   j = F->mult(i, i);
   sqrt_mod_q[j] = i;
   }
cout << "sqrt_mod_q:" << endl;
INT_vec_print(cout, sqrt_mod_q, q);
cout << endl;

sqrt_p = 0;
for (i = 1; i < p; i++) {
   if (i * i > p) {
      sqrt_p = i - 1;
      break;
      }
   }
cout << "p=" << p << endl;
cout << "sqrt_p = " << sqrt_p << endl;


for (I = 0; I < q; I++) {
   if (F->add(F->mult(I, I), 1) == 0) {
      break;
      }
   }
if (I == q) {
   cout << "did not find I" << endl;
   exit(1);
   }
cout << "I=" << I << endl;
```

```
for (a0 = 1; a0 <= sqrt_p; a0++) {
   if (EVEN(a0)) {
      continue;
      }
   for (a1 = -sqrt_p; a1 <= sqrt_p; a1++) {
      if (ODD(a1)) {
         continue;
         }
      for (a2 = -sqrt_p; a2 <= sqrt_p; a2++) {
         if (ODD(a2)) {
            continue;
            }
         for (a3 = -sqrt_p; a3 <= sqrt_p; a3++) {
            if (ODD(a3)) {
               continue;
               }
            if (a0 * a0 + a1 * a1 + a2 * a2 + a3 * a3 == p) {
               cout << "solution " << nb_A4 << " : " << a0 << ", " << a1 << ", " << a2 << ", " << a3 << ", " <<
               if (nb_A4 == p + 1) {
                  cout << "too many solutions" << endl;
                  exit(1);
                  }
               A4[nb_A4 * 4 + 0] = a0;
               A4[nb_A4 * 4 + 1] = a1;
               A4[nb_A4 * 4 + 2] = a2;
               A4[nb_A4 * 4 + 3] = a3;
               nb_A4++;
               }
            }
         }
      }
   }

cout << "nb_A4=" << nb_A4 << endl;
if (nb_A4 != p + 1) {
   cout << "nb_A4 != p + 1" << endl;
   exit(1);
   }

INT_matrix_print(A4, nb_A4, 4);

vector_ge *gens;
INT *Elt1;
INT *Elt2;
INT *Elt3;
INT M4[4];
INT det, s, sv;

Elt1 = NEW_INT(A->elt_size_in_INT);
Elt2 = NEW_INT(A->elt_size_in_INT);
Elt3 = NEW_INT(A->elt_size_in_INT);

gens = new vector_ge;
gens->init(A);
gens->allocate(nb_A4);
```

```
cout << "making connection set:" << endl;
for (i = 0; i < nb_A4; i++) {

   if (f_vv) {
      cout << "making generator " << i << ":" << endl;
      }
   a0 = A4[i * 4 + 0] % q;
   a1 = A4[i * 4 + 1] % q;
   a2 = A4[i * 4 + 2] % q;
   a3 = A4[i * 4 + 3] % q;
   if (a0 < 0) {
      a0 += q;
      }
   if (a1 < 0) {
      a1 += q;
      }
   if (a2 < 0) {
      a2 += q;
      }
   if (a3 < 0) {
      a3 += q;
      }
   M4[0] = F->add(a0, F->mult(I, a1));
   M4[1] = F->add(a2, F->mult(I, a3));
   M4[2] = F->add(F->negate(a2), F->mult(I, a3));
   M4[3] = F->add(a0, F->negate(F->mult(I, a1)));

   if (f_vv) {
      cout << "M4=";
      INT_vec_print(cout, M4, 4);
      cout << endl;
      }

   if (f_special) {
      det = F->add(F->mult(M4[0], M4[3]), F->negate(F->mult(M4[1], M4[2])));

      if (f_vv) {
         cout << "det=" << det << endl;
         }

      s = sqrt_mod_q[det];
      if (s == -1) {
         cout << "determinant is not a square" << endl;
         exit(1);
         }
      sv = F->inverse(s);
      for (j = 0; j < 4; j++) {
         M4[j] = F->mult(sv, M4[j]);
         }
      if (f_vv) {
         cout << "M4=";
         INT_vec_print(cout, M4, 4);
         cout << endl;
         }
      }
      A->make_element(Elt1, M4, verbose_level - 1);

      if (f_v) {
         cout << "s_" << i << "=" << endl;
         A->element_print_quick(Elt1, cout);
         }

      A->element_move(Elt1, gens->ith(i), 0);
      }


   INT *Adj;

   Adj = NEW_INT(goi * goi);

   INT_vec_zero(Adj, goi * goi);

   cout << "Computing the Cayley graph:" << endl;
   for (i = 0; i < goi; i++) {
      Sims->element_unrank_INT(i, Elt1);
      for (h = 0; h < nb_A4; h++) {
         A->element_mult(Elt1, gens->ith(h), Elt2, 0);
#if 0
         cout << "i=" << i << " h=" << h << endl;
         cout << "Elt1=" << endl;
         A->element_print_quick(Elt1, cout);
         cout << "g_h=" << endl;
         A->element_print_quick(gens->ith(h), cout);
         cout << "Elt2=" << endl;
         A->element_print_quick(Elt2, cout);
#endif
         j = Sims->element_rank_INT(Elt2);
         Adj[i * goi + j] = Adj[j * goi + i] = 1;
         }
      }

   cout << "The adjacency matrix of a graph with " << goi << " vertices has been computed" << endl;
   //INT_matrix_print(Adj, goi, goi);



   colored_graph *CG;
   BYTE fname[1000];

   CG = new colored_graph;
   CG->init_adjacency_no_colors(goi, Adj, verbose_level);

   sprintf(fname, "Sarnak_%ld_%ld.colored_graph", p, q);

   CG->save(fname, verbose_level);


   delete CG;
   delete gens;
   delete A;
   FREE_INT(A4);
   FREE_INT(Elt1);
   FREE_INT(Elt2);
   FREE_INT(Elt3);
   delete F;

}
```

Here is a drawing of the graph created for $q = 5$ and $p = 29$. The graph has 60 vertices which are in correspondence to the elements of PSL$(2, 5)$. Each vertex is incident with 27 vertices.



7.10 Determining The Equation of a Conic

In the projective plane, five points with the property that no three collinear determine a conic. Here is a program to compute the equation of the conic given 5 points.

```
// determine_conic.C
//
```

```
// Anton Betten
// Nov 16, 2010
//
// based on COMBINATORICS/conic.C
//
// computes the equation of a conic through 5 given points
// in PG(2, q)
// usage:
// -q <q> -pts <p1> <p2> <p3> <p4> <p5>
// OR
// -q <q> -pt_coords <x1> <y1> <z1> <x2> <y2> <z2> ... <x4> <y5> <z5> -1



#include "orbiter.h"


INT determine_conic(finite_field *F, INT *input_pts, INT nb_pts, INT *six_coeffs, INT verbose_level);
// returns FALSE is the rank of the coefficient matrix is not 5. TRUE otherwise.


int main(int argc, char **argv)
{
   INT verbose_level = 1;
   INT i;
   INT q = -1;
   INT nb_pts = 0;
   INT pts[1000];
   INT nb_pt_coords = 0;
   INT pt_coords[1000];
   INT f_poly = FALSE;
   const BYTE *override_poly = NULL;

   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-v") == 0) {
         verbose_level = atoi(argv[++i]);
         cout << "-v " << verbose_level << endl;
         }
      else if (strcmp(argv[i], "-q") == 0) {
         q = atoi(argv[++i]);
         cout << "-q " << q << endl;
         }
      else if (strcmp(argv[i], "-poly") == 0) {
         f_poly = TRUE;
         override_poly = argv[++i];
         cout << "-poly " << override_poly << endl;
         }
      else if (strcmp(argv[i], "-pts") == 0) {
         while (TRUE) {
            pts[nb_pts] = atoi(argv[++i]);
            if (pts[nb_pts] == -1) {
               break;
               }
            nb_pts++;
            }
         cout << "-pts ";
```

```
            INT_vec_print(cout, pts, nb_pts);
            cout << endl;
            }
        else if (strcmp(argv[i], "-pt_coords") == 0) {
            while (TRUE) {
                pt_coords[nb_pt_coords] = atoi(argv[++i]);
                if (pt_coords[nb_pt_coords] == -1) {
                    break;
                    }
                nb_pt_coords++;
                }
            cout << "-pt_coords ";
            INT_vec_print(cout, pt_coords, nb_pt_coords);
            cout << endl;
            }
        }

    INT f_v = (verbose_level >= 1);
    INT *input_pts;
    INT six_coeffs[6];
    finite_field *F;

    if (q == -1) {
        cout << "please use option -q <q>" << endl;
        exit(1);
        }

    F = new finite_field;

    F->init(q, 0);
    F->init_override_polynomial(q, override_poly, verbose_level);

    if (nb_pts) {
        if (nb_pts < 5) {
            cout << "please give exactly 5 points using -pts <p1> ... <p5>" << endl;
            exit(1);
            }
        input_pts = NEW_INT(nb_pts);
        INT_vec_copy(pts, input_pts, nb_pts);
        }
    else if (nb_pt_coords) {
        INT a;

        nb_pts = nb_pt_coords / 3;
        if (nb_pt_coords < 15) {
            cout << "please give at least 15 point coordinates using -pt_coords <p1> ... <p15>" << endl;
            exit(1);
            }
        input_pts = NEW_INT(nb_pts);
        for (i = 0; i < nb_pts; i++) {
            cout << "point " << i << " has coordinates ";
            INT_vec_print(cout, pt_coords + i * 3, 3);
            cout << endl;
            a = F->rank_point_in_PG(pt_coords + i * 3, 3);
            input_pts[i] = a;
            cout << "and rank " << a << endl;
```

```
      }
   }
else {
   cout << "Please specify points using -pts or using -pt_coordinates" << endl;
   exit(1);
   }



cout << "input_pts: ";
INT_vec_print(cout, input_pts, nb_pts);
cout << endl;

determine_conic(F, input_pts, nb_pts, six_coeffs, verbose_level);

if (f_v) {
   cout << "determine_conic the conic is ";
   INT f_first = TRUE;
   if (six_coeffs[0]) {
      cout << six_coeffs[0] << "*X^2";
      f_first = FALSE;
      }
   if (six_coeffs[1]) {
      if (!f_first) {
         cout << " + ";
         }
      cout << six_coeffs[1] << "*Y^2";
      f_first = FALSE;
      }
   if (six_coeffs[2]) {
      if (!f_first) {
         cout << " + ";
         }
      cout << six_coeffs[2] << "*Z^2";
      f_first = FALSE;
      }
   if (six_coeffs[3]) {
      if (!f_first) {
         cout << " + ";
         }
      cout << six_coeffs[3] << "*XY";
      f_first = FALSE;
      }
   if (six_coeffs[4]) {
      if (!f_first) {
         cout << " + ";
         }
      cout << six_coeffs[4] << "*XZ";
      f_first = FALSE;
      }
   if (six_coeffs[5]) {
      if (!f_first) {
         cout << " + ";
         }
      cout << six_coeffs[5] << "*YZ";
      f_first = FALSE;
```

```
         }
      cout << endl;
      }

   delete F;
   FREE_INT(input_pts);

}

INT determine_conic(finite_field *F, INT *input_pts, INT nb_pts, INT *six_coeffs, INT verbose_level)
// returns FALSE is the rank of the coefficient matrix is not 5. TRUE otherwise.
{
   INT f_v = (verbose_level >= 1);
   INT f_vv = (verbose_level >= 2);
   INT *coords; // [nb_pts * 3];
   INT *system; // [nb_pts * 6];
   INT kernel[6 * 6];
   INT base_cols[6];
   INT i, x, y, z, rk;
   INT kernel_m, kernel_n;

   if (f_v) {
      cout << "determine_conic" << endl;
      }
   if (nb_pts < 5) {
      cout << "determine_conic need at least 5 points" << endl;
      exit(1);
      }


   coords = NEW_INT(nb_pts * 3);
   system = NEW_INT(nb_pts * 6);
   for (i = 0; i < nb_pts; i++) {
      F->unrank_point_in_PG(coords + i * 3, 3, input_pts[i]);
      }
   if (f_vv) {
      cout << "determine_conic points:" << endl;
      print_integer_matrix_width(cout, coords, nb_pts, 3, 3, F->log10_of_q);
      }
   for (i = 0; i < nb_pts; i++) {
      x = coords[i * 3 + 0];
      y = coords[i * 3 + 1];
      z = coords[i * 3 + 2];
      system[i * 6 + 0] = F->mult(x, x);
      system[i * 6 + 1] = F->mult(y, y);
      system[i * 6 + 2] = F->mult(z, z);
      system[i * 6 + 3] = F->mult(x, y);
      system[i * 6 + 4] = F->mult(x, z);
      system[i * 6 + 5] = F->mult(y, z);
      }
   if (f_v) {
      cout << "determine_conic system:" << endl;
      print_integer_matrix_width(cout, system, nb_pts, 6, 6, F->log10_of_q);
      }
   rk = F->Gauss_simple(system, nb_pts, 6, base_cols, verbose_level - 2);
   if (rk != 5) {
      if (f_v) {
         cout << "determine_conic system underdetermined" << endl;
         }
      return FALSE;
      }
   F->matrix_get_kernel(system, 5, 6, base_cols, rk,
      kernel_m, kernel_n, kernel);
   if (f_v) {
      cout << "determine_conic conic:" << endl;
      print_integer_matrix_width(cout, kernel, 1, 6, 6, F->log10_of_q);
      }
   for (i = 0; i < 6; i++) {
      six_coeffs[i] = kernel[i];
      }
   FREE_INT(coords);
   FREE_INT(system);
   return TRUE;
}
```

Suppose we are in the plane defined over $\mathbb{F}_5$, and we are looking for the equation of the conic through the points $(0,0,1)$, $(1,1,1)$, $(2,4,1)$, $(3,4,1)$, $(4,1,1)$. Here is the output of the program:

```
-v 1
-q 5
-pt_coords ( 0, 0, 1, 1, 1, 1, 2, 4, 1, 3, 4, 1, 4, 1, 1 )
finite_field::init_override_polynomial
finite_field::init_override_polynomial() GF(5) = GF(5^1)
point 0 has coordinates ( 0, 0, 1 )
and rank 2
point 1 has coordinates ( 1, 1, 1 )
and rank 3
point 2 has coordinates ( 2, 4, 1 )
and rank 28
point 3 has coordinates ( 3, 4, 1 )
and rank 29
point 4 has coordinates ( 4, 1, 1 )
and rank 15
input_pts: ( 2, 3, 28, 29, 15 )
determine_conic
determine_conic system:
0 0 1 0 0 0
1 1 1 1 1 1
4 1 1 3 2 4
4 1 1 2 3 4
1 1 1 4 4 1
determine_conic conic:
1 0 0 0 0 4
determine_conic the conic is 1*X^2 + 4*YZ
```

The program correctly finds that the given points lie on the conic $X^2 - YZ$.

8 Classification Problems

In this section, we will look at some applications of orbiter to the classificaton of combinatorial objects. We will classify graphs, hyperovals and BLT-sets.

8.1 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/APPS/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

```
ORBITER/SRC/APPS/GRAPH/graph.out -n 4 -v 1
```

The output produced by this command contains the following lines:

```
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
```

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

k	0	1	2	3	4	5	6
# graphs	1	1	2	3	2	1	1

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

```
make g4
make g5
make g6
make g7
make g8
make g9
```

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

n	1	2	3	4	5	6	7	8	9
# graphs	1	2	4	11	34	156	1044	12346	274668

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 issueing 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/APPS/GRAPH/graph.C
ORBITER/SRC/APPS/GRAPH/graph.h
ORBITER/SRC/APPS/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 of 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):

```
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`):

```
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 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

```
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 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 Sym_n acting on the vertices. This is what the action `A_base` is for. The action `A_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_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 Sym_n is done in the action on n points. `orbiter` uses the action `A_on_edges` whenever it is classifying subsets, but maintains the group and group elements in the action `A_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_on_edges->induced_action_on_ordered_pairs(*A_base,
  A_base->Sims, 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 preceeding 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

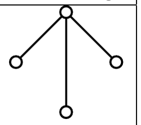
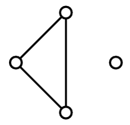
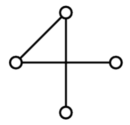
```
ORBITER/SRC/APPS/GRAPH/graph.out -n 4 -v 1 -W
```

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):

```
# 3
3 0 1 2 6 aaaaaaadaaaaaaadaaaaaaaaaaaaaaabaaaaaaacaaaaaaab
    aaaaaaadaaaaaaacaaabadacaaacabadaaadacab
3 0 1 3 6 aaaaaaadaaaaaaacaaaaaaaaaaaaaaabaaaaaaacaaaaaaad
    aaaaaaacaaaaaaabaaacabadabacaaad
3 0 1 4 2 aaaaaaadaaaaaaabaaaaaaaaaaaaaaabaaaaaaacaaaaaaac
    aaaaaaabaaaaaaababaaadac
-1 3 4 in 0:00(6^2, 2) average is 4 + 2 / 3
```

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 lagacy 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

Representative	Edges	Stabilizer Order	Description	Drawing
$\{0,1,2\}$	$\{01,02,03\}$	6	3-Claw	
$\{0,1,3\}$	$\{01,02,04\}$	6	Triangle	
$\{0,1,4\}$	$\{01,02,12\}$	2	Path	

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 $\frac{6+6+2}{3} = 4 + \frac{2}{3}$.

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

```
make g8r3
```

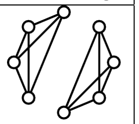
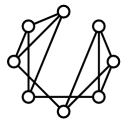
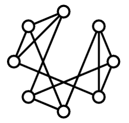
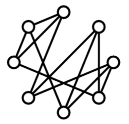
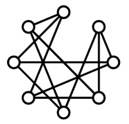
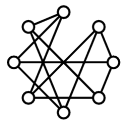
which is an abbreviation for

```
ORBITER/SRC/APPS/GRAPH/graph.out -n 8 -depth 12 -regular 3 -v 2 -W
```

classifies cubic graphs on 8 vertices. We find 6 graphs. Reading the file

```
graph_8_r3_lvl_12
```

we find that these six graphs are represented by

Representative	Stabilizer Order	Drawing
$\{0, 1, 2, 7, 8, 13, 22, 23, 24, 25, 26, 27\}$	1152	
$\{0, 1, 2, 7, 8, 14, 19, 23, 24, 25, 26, 27\}$	16	
$\{0, 1, 2, 7, 9, 15, 18, 20, 24, 25, 26, 27\}$	4	
$\{0, 1, 2, 7, 9, 15, 20, 21, 23, 24, 25, 26\}$	12	
$\{0, 1, 2, 9, 10, 14, 16, 19, 20, 24, 26, 27\}$	48	
$\{0, 1, 2, 9, 10, 14, 16, 19, 21, 24, 25, 27\}$	16	

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.

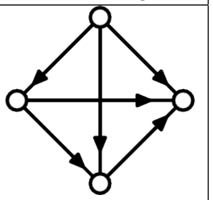
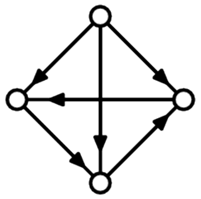
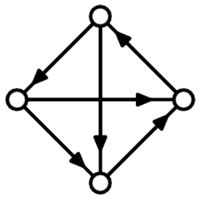
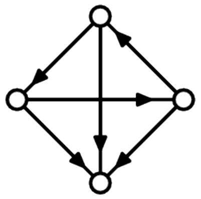
Next, let us have a look at tournaments. A tournament is a directed graph such that any two vertices are connected by exactly one directed edge (thus, a tournament is a direction of a complete graph). Let us classify tournaments with `orbiter`. The command

```
make t4
```

which is an abbreviation for

```
ORBITER/SRC/APPS/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:

Representative	Stabilizer Order	Drawing
$\{0, 2, 4, 6, 8, 10\}$	1	
$\{0, 2, 4, 6, 9, 10\}$	3	
$\{0, 2, 5, 6, 8, 10\}$	1	
$\{0, 2, 5, 6, 8, 11\}$	3	

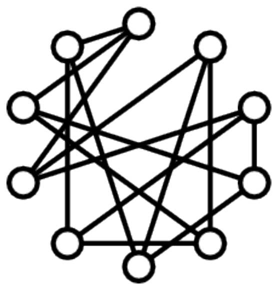
No piece in graph theory is complete without the unique graph on 10 vertices that is 3-regular of girth 5, better known as the Petersen graph. In `orbiter`, we issue the command

```
ORBITER/SRC/APPS/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

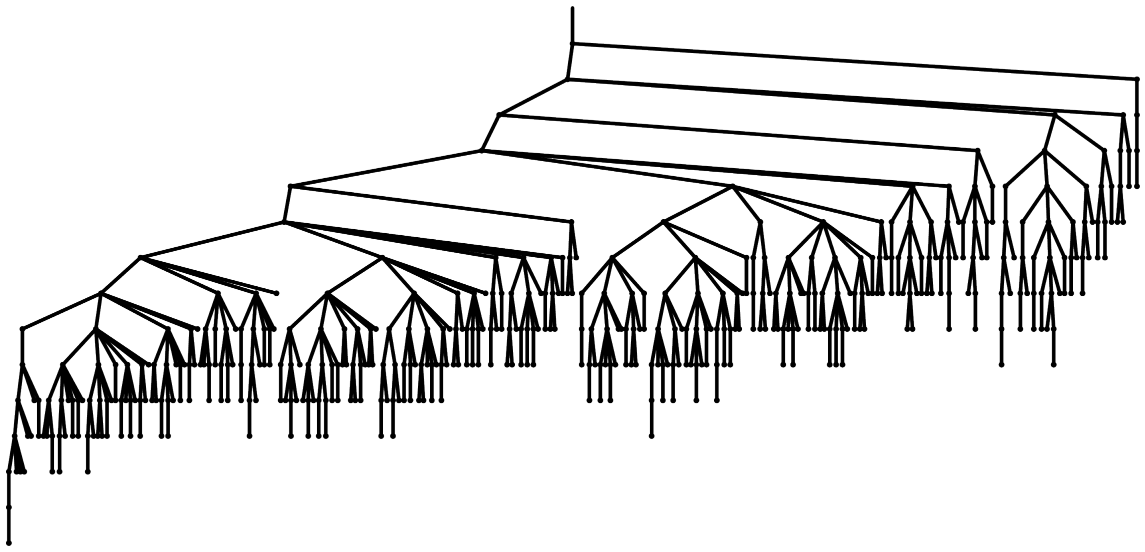
$$\{0, 1, 2, 11, 12, 20, 21, 28, 29, 31, 33, 36, 38, 41, 42\}$$

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



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_base` and `A_on_edges` represent the permutation group whose orbits on subsets we plan to classify. The next two arguments `A_base->strong_generators` and `A_base->tl` represent the actual generating system for the group. The argument `target_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 `prefix` is the prefix for the name of the output files. The last argument `verbose_level - 1` specifies how verbose the function `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 `generator` of a test function. This is done by means of a function pointer. So, we find the command

```
gen->init_check_func(::check_conditions,
    (void *)this /* candidate_check_data */);
```

which informs `generator` of the presence of the function `::check_conditions`. This test function is a global function because function pointers are a feature of C, and they can only be pointers to gloabl functions, not member functions of a class. This leaves the issue of telling the global test function the instance of the class `graph_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 `generator`. Whenever the instance of `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 `my_generator`), and issues the call of the appropriate member function. Here is the definition of the global test function:

```
INT check_conditions(ostream &ost, INT len, INT *S,
    void *data, INT verbose_level)
{
    graph_generator *Gen = (graph_generator *) data;

    if (Gen->f_tournament) {
        return Gen->check_conditions_tournament(len, S, verbose_level);
        }
    else {
        return Gen->check_conditions(len, S, verbose_level);
    }
}
```

As we can see, the void pointer `data` is the fourth argument. The second and thir 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 verboseness of the test function. The line

```
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.

8.2 Classifying Arcs, Ovals and Hyperovals

A set of points in the Desarguesian projective plane $\mathrm{PG}(2,q)$ 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 $\mathrm{PG}(2,q)$ 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 $\mathrm{PG}(2,64)$ seems to be beyond reach.

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

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

The class `arc_generator` maintains the projective plane $\mathrm{PG}(2,q)$ together with its group $\mathrm{P\Gamma L}(3,q)$. 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));
```

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`.

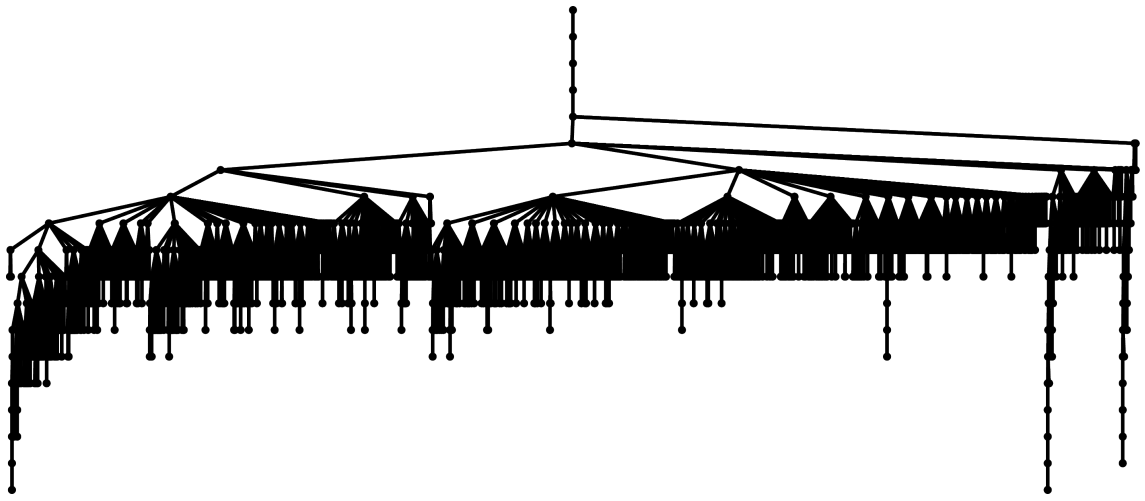
Inside `ORBITER/DATA/HYPEROVAL`, we could issue any of the commands

```
make 8
make 16
```

to classify the hyperovals in $\mathrm{PG}(2,8)$ and $\mathrm{PG}(2,16)$, respectively. This algorithm classifies the orbits of $\mathrm{P\Gamma L}(2,q)$ on arcs in $\mathrm{PG}(2,q)$. 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 $\mathrm{PG}(2,8)$, the only hyperoval is the hyperconic. In $\mathrm{PG}(2,16)$, 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 $\mathrm{PG}(2,16)$ has 4214 nodes and looks like this:



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.

8.3 Classifying BLT-Sets

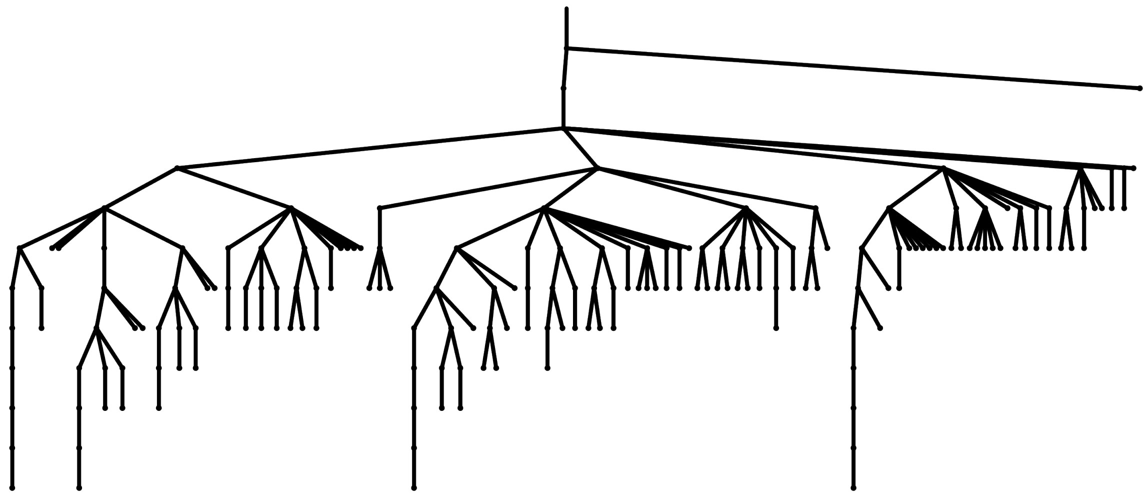
We will describe one application that falls in the area of finite geometry. A $Q(4,q)$ space is a four-dimensional projective space over the field $\mathbb{F}_q$ equipped with a non-degenerate quadratic form. In such a space, sets of points that are known as BLT-sets are of interest. A BLT-set is a set 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/APPS/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/APPS/BLT/blt_generator.C
```

The class `blt_generator` maintains the quadric $Q(4,q)$ together with its group $\mathrm{P\Gamma O}(5,q)$. 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 $\mathbb{F}_p$ that is used to create the field $\mathbb{F}_q$ (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;
```

A List of Files

```
ORBITER/
ORBITER/makefile
ORBITER/makefile~
ORBITER/RUN/
ORBITER/RUN/BLT/
ORBITER/RUN/BLT/13/
ORBITER/RUN/BLT/13/makefile
ORBITER/RUN/BLT/makefile
ORBITER/RUN/BLT/makefile~
ORBITER/RUN/CODES/
ORBITER/RUN/CODES/2/
ORBITER/RUN/CODES/2/makefile
ORBITER/RUN/CODES/4/
ORBITER/RUN/CODES/4/makefile
ORBITER/RUN/CODES/makefile
ORBITER/RUN/GRAPH/
ORBITER/RUN/GRAPH/CUBIC6/
ORBITER/RUN/GRAPH/CUBIC6/makefile
ORBITER/RUN/GRAPH/makefile
ORBITER/RUN/GRAPH/makefile~
ORBITER/RUN/GRAPH/PETERSEN/
ORBITER/RUN/GRAPH/PETERSEN/makefile
ORBITER/RUN/GRAPH/T4/
ORBITER/RUN/GRAPH/T4/makefile
ORBITER/RUN/GRAPH/T4/makefile~
ORBITER/RUN/HYPEROVAL/
ORBITER/RUN/HYPEROVAL/16/
ORBITER/RUN/HYPEROVAL/16/makefile
ORBITER/RUN/HYPEROVAL/16/makefile~
```

```
ORBITER/RUN/HYPEROVAL/32/
ORBITER/RUN/HYPEROVAL/32/makefile
ORBITER/RUN/HYPEROVAL/32/makefile~
ORBITER/RUN/HYPEROVAL/makefile
ORBITER/RUN/HYPEROVAL/makefile~
ORBITER/RUN/makefile
ORBITER/RUN/makefile~
ORBITER/SRC/
ORBITER/SRC/APPS/
ORBITER/SRC/APPS/BLT/
ORBITER/SRC/APPS/BLT/blt.C
ORBITER/SRC/APPS/BLT/blt.h
ORBITER/SRC/APPS/BLT/blt_generator.C
ORBITER/SRC/APPS/BLT/makefile
ORBITER/SRC/APPS/CODES/
ORBITER/SRC/APPS/CODES/code_generator.C
ORBITER/SRC/APPS/CODES/codes.C
ORBITER/SRC/APPS/CODES/codes.h
ORBITER/SRC/APPS/CODES/makefile
ORBITER/SRC/APPS/COMBINATORICS/
ORBITER/SRC/APPS/COMBINATORICS/all_k_subsets.C
ORBITER/SRC/APPS/COMBINATORICS/burnside.C
ORBITER/SRC/APPS/COMBINATORICS/cayley.C
ORBITER/SRC/APPS/COMBINATORICS/cayley_sym_n.C
ORBITER/SRC/APPS/COMBINATORICS/change_1_dollar.C
ORBITER/SRC/APPS/COMBINATORICS/conic.C
ORBITER/SRC/APPS/COMBINATORICS/design.C
ORBITER/SRC/APPS/COMBINATORICS/fibonacci.C
ORBITER/SRC/APPS/COMBINATORICS/field.C
ORBITER/SRC/APPS/COMBINATORICS/grassmann.C
ORBITER/SRC/APPS/COMBINATORICS/group_ring.C
ORBITER/SRC/APPS/COMBINATORICS/hadamard.C
ORBITER/SRC/APPS/COMBINATORICS/hamming.C
ORBITER/SRC/APPS/COMBINATORICS/johnson.C
ORBITER/SRC/APPS/COMBINATORICS/johnson_table.C
ORBITER/SRC/APPS/COMBINATORICS/lehmer.c
ORBITER/SRC/APPS/COMBINATORICS/makefile
ORBITER/SRC/APPS/COMBINATORICS/nondecreasing.C
ORBITER/SRC/APPS/COMBINATORICS/paley.C
ORBITER/SRC/APPS/COMBINATORICS/partitions.C
ORBITER/SRC/APPS/COMBINATORICS/pentomino_5x5.C
ORBITER/SRC/APPS/COMBINATORICS/problem11.C
ORBITER/SRC/APPS/COMBINATORICS/puzzle.C
ORBITER/SRC/APPS/COMBINATORICS/sarnak.C
ORBITER/SRC/APPS/COMBINATORICS/schlaefli.C
ORBITER/SRC/APPS/COMBINATORICS/sequences.C
ORBITER/SRC/APPS/COMBINATORICS/shrikhande.C
ORBITER/SRC/APPS/COMBINATORICS/test.C
ORBITER/SRC/APPS/COMBINATORICS/test2.C
ORBITER/SRC/APPS/COMBINATORICS/test3.C
ORBITER/SRC/APPS/COMBINATORICS/winnie_li.C
```

```
ORBITER/SRC/APPS/GRAPH/
ORBITER/SRC/APPS/GRAPH/graph.C
ORBITER/SRC/APPS/GRAPH/graph.h
ORBITER/SRC/APPS/GRAPH/graph_generator.C
ORBITER/SRC/APPS/GRAPH/makefile
ORBITER/SRC/APPS/HYPEROVAL/
ORBITER/SRC/APPS/HYPEROVAL/arc_generator.C
ORBITER/SRC/APPS/HYPEROVAL/hyperoval.C
ORBITER/SRC/APPS/HYPEROVAL/hyperoval.h
ORBITER/SRC/APPS/HYPEROVAL/makefile
ORBITER/SRC/APPS/HYPEROVAL/test_hyperoval.C
ORBITER/SRC/APPS/makefile
ORBITER/SRC/APPS/makefile~
ORBITER/SRC/APPS/TOOLS/
ORBITER/SRC/APPS/TOOLS/all_cliques.C
ORBITER/SRC/APPS/TOOLS/all_cycles.C
ORBITER/SRC/APPS/TOOLS/all_rainbow_cliques.C
ORBITER/SRC/APPS/TOOLS/dlx.C
ORBITER/SRC/APPS/TOOLS/draw_colored_graph.C
ORBITER/SRC/APPS/TOOLS/layered_graph_main.C
ORBITER/SRC/APPS/TOOLS/makefile
ORBITER/SRC/APPS/TOOLS/plot_decomposition_matrix.C
ORBITER/SRC/APPS/TOOLS/solve_diophant.C
ORBITER/SRC/APPS/TOOLS/treedraw.C
ORBITER/SRC/APPS/TRANSLATION_PLANE/
ORBITER/SRC/APPS/TRANSLATION_PLANE/extending.C
ORBITER/SRC/APPS/TRANSLATION_PLANE/makefile
ORBITER/SRC/APPS/TRANSLATION_PLANE/translation_plane.h
ORBITER/SRC/APPS/TRANSLATION_PLANE/translation_plane_main.C
ORBITER/SRC/LIB/
ORBITER/SRC/LIB/ACTION/
ORBITER/SRC/LIB/ACTION/action.C
ORBITER/SRC/LIB/ACTION/action.h
ORBITER/SRC/LIB/ACTION/action_by_conjugation.C
ORBITER/SRC/LIB/ACTION/action_by_representation.C
ORBITER/SRC/LIB/ACTION/action_by_restriction.C
ORBITER/SRC/LIB/ACTION/action_by_right_multiplication.C
ORBITER/SRC/LIB/ACTION/action_by_subfield_structure.C
ORBITER/SRC/LIB/ACTION/action_cb.C
ORBITER/SRC/LIB/ACTION/action_global.C
ORBITER/SRC/LIB/ACTION/action_induce.C
ORBITER/SRC/LIB/ACTION/action_init.C
ORBITER/SRC/LIB/ACTION/action_on_andre.C
ORBITER/SRC/LIB/ACTION/action_on_bricks.C
ORBITER/SRC/LIB/ACTION/action_on_cosets.C
ORBITER/SRC/LIB/ACTION/action_on_determinant.C
ORBITER/SRC/LIB/ACTION/action_on_factor_space.C
ORBITER/SRC/LIB/ACTION/action_on_grassmannian.C
ORBITER/SRC/LIB/ACTION/action_on_k_subsets.C
ORBITER/SRC/LIB/ACTION/action_on_orthogonal.C
ORBITER/SRC/LIB/ACTION/action_on_sets.C
```

```
ORBITER/SRC/LIB/ACTION/action_on_spread_set.C
ORBITER/SRC/LIB/ACTION/action_on_wedge_product.C
ORBITER/SRC/LIB/ACTION/backtrack.C
ORBITER/SRC/LIB/ACTION/desarguesian_spread.C
ORBITER/SRC/LIB/ACTION/group.C
ORBITER/SRC/LIB/ACTION/interface.C
ORBITER/SRC/LIB/ACTION/interface_matrix_group.C
ORBITER/SRC/LIB/ACTION/interface_perm_group.C
ORBITER/SRC/LIB/ACTION/makefile
ORBITER/SRC/LIB/ACTION/matrix_group.C
ORBITER/SRC/LIB/ACTION/page_storage.C
ORBITER/SRC/LIB/ACTION/perm_group.C
ORBITER/SRC/LIB/ACTION/product_action.C
ORBITER/SRC/LIB/ACTION/schreier.C
ORBITER/SRC/LIB/ACTION/schreier_sims.C
ORBITER/SRC/LIB/ACTION/sims.C
ORBITER/SRC/LIB/ACTION/sims2.C
ORBITER/SRC/LIB/ACTION/sims_global.C
ORBITER/SRC/LIB/ACTION/strong_generators.C
ORBITER/SRC/LIB/ACTION/union_find.C
ORBITER/SRC/LIB/ACTION/union_find_on_k_subsets.C
ORBITER/SRC/LIB/ACTION/vector_ge.C
ORBITER/SRC/LIB/DISCRETA/
ORBITER/SRC/LIB/DISCRETA/base.C
ORBITER/SRC/LIB/DISCRETA/bitmatrix.C
ORBITER/SRC/LIB/DISCRETA/bt_key.C
ORBITER/SRC/LIB/DISCRETA/btree.C
ORBITER/SRC/LIB/DISCRETA/database.C
ORBITER/SRC/LIB/DISCRETA/design.C
ORBITER/SRC/LIB/DISCRETA/design_parameter.C
ORBITER/SRC/LIB/DISCRETA/design_parameter_source.C
ORBITER/SRC/LIB/DISCRETA/discreta.h
ORBITER/SRC/LIB/DISCRETA/discreta_global.C
ORBITER/SRC/LIB/DISCRETA/domain.C
ORBITER/SRC/LIB/DISCRETA/geometry.C
ORBITER/SRC/LIB/DISCRETA/global.C
ORBITER/SRC/LIB/DISCRETA/group_selection.C
ORBITER/SRC/LIB/DISCRETA/hollerith.C
ORBITER/SRC/LIB/DISCRETA/integer.C
ORBITER/SRC/LIB/DISCRETA/longinteger.C
ORBITER/SRC/LIB/DISCRETA/makefile
ORBITER/SRC/LIB/DISCRETA/matrix.C
ORBITER/SRC/LIB/DISCRETA/memory.C
ORBITER/SRC/LIB/DISCRETA/number_partition.C
ORBITER/SRC/LIB/DISCRETA/page_table.C
ORBITER/SRC/LIB/DISCRETA/perm_group_gens.C
ORBITER/SRC/LIB/DISCRETA/permutation.C
ORBITER/SRC/LIB/DISCRETA/solid.C
ORBITER/SRC/LIB/DISCRETA/unipoly.C
ORBITER/SRC/LIB/DISCRETA/vector.C
ORBITER/SRC/LIB/GALOIS/
```

```
ORBITER/SRC/LIB/GALOIS/a_domain.C
ORBITER/SRC/LIB/GALOIS/andre_construction.C
ORBITER/SRC/LIB/GALOIS/andre_construction_line_element.C
ORBITER/SRC/LIB/GALOIS/andre_construction_point_element.C
ORBITER/SRC/LIB/GALOIS/brick_domain.C
ORBITER/SRC/LIB/GALOIS/buekenhout_metz.C
ORBITER/SRC/LIB/GALOIS/classify.C
ORBITER/SRC/LIB/GALOIS/clique_finder.C
ORBITER/SRC/LIB/GALOIS/colored_graph.C
ORBITER/SRC/LIB/GALOIS/combinatorics.C
ORBITER/SRC/LIB/GALOIS/data.C
ORBITER/SRC/LIB/GALOIS/data_BLT.C
ORBITER/SRC/LIB/GALOIS/data_BLT_old.C
ORBITER/SRC/LIB/GALOIS/data_DH.C
ORBITER/SRC/LIB/GALOIS/data_file.C
ORBITER/SRC/LIB/GALOIS/data_hyperovals.C
ORBITER/SRC/LIB/GALOIS/data_packings_PG_3_3.C
ORBITER/SRC/LIB/GALOIS/data_TP.C
ORBITER/SRC/LIB/GALOIS/decomposition.C
ORBITER/SRC/LIB/GALOIS/diophant.C
ORBITER/SRC/LIB/GALOIS/dlx.C
ORBITER/SRC/LIB/GALOIS/draw.C
ORBITER/SRC/LIB/GALOIS/fancy_set.C
ORBITER/SRC/LIB/GALOIS/finite_field.C
ORBITER/SRC/LIB/GALOIS/finite_field_linear_algebra.C
ORBITER/SRC/LIB/GALOIS/finite_field_representations.C
ORBITER/SRC/LIB/GALOIS/finite_field_tables.C
ORBITER/SRC/LIB/GALOIS/finite_ring.C
ORBITER/SRC/LIB/GALOIS/galois.h
ORBITER/SRC/LIB/GALOIS/galois_global.C
ORBITER/SRC/LIB/GALOIS/geometric_object.C
ORBITER/SRC/LIB/GALOIS/geometric_operations.C
ORBITER/SRC/LIB/GALOIS/gl_classes.C
ORBITER/SRC/LIB/GALOIS/graph_layer.C
ORBITER/SRC/LIB/GALOIS/graph_node.C
ORBITER/SRC/LIB/GALOIS/grassmann.C
ORBITER/SRC/LIB/GALOIS/grassmann_embedded.C
ORBITER/SRC/LIB/GALOIS/hermitian.C
ORBITER/SRC/LIB/GALOIS/hjelmslev.C
ORBITER/SRC/LIB/GALOIS/incidence_structure.C
ORBITER/SRC/LIB/GALOIS/INT_matrix.C
ORBITER/SRC/LIB/GALOIS/INT_vector.C
ORBITER/SRC/LIB/GALOIS/layered_graph.C
ORBITER/SRC/LIB/GALOIS/longinteger_domain.C
ORBITER/SRC/LIB/GALOIS/longinteger_object.C
ORBITER/SRC/LIB/GALOIS/makefile
ORBITER/SRC/LIB/GALOIS/memory.C
ORBITER/SRC/LIB/GALOIS/memory_object.C
ORBITER/SRC/LIB/GALOIS/mindist.C
ORBITER/SRC/LIB/GALOIS/mp_graphics.C
ORBITER/SRC/LIB/GALOIS/naugraph.c
```

```
ORBITER/SRC/LIB/GALOIS/nautil.c
ORBITER/SRC/LIB/GALOIS/nauty.c
ORBITER/SRC/LIB/GALOIS/nauty.h
ORBITER/SRC/LIB/GALOIS/nauty_interface.C
ORBITER/SRC/LIB/GALOIS/norm_tables.C
ORBITER/SRC/LIB/GALOIS/number_theory.C
ORBITER/SRC/LIB/GALOIS/orthogonal.C
ORBITER/SRC/LIB/GALOIS/orthogonal_points.C
ORBITER/SRC/LIB/GALOIS/partitionstack.C
ORBITER/SRC/LIB/GALOIS/plot.C
ORBITER/SRC/LIB/GALOIS/projective.C
ORBITER/SRC/LIB/GALOIS/projective_space.C
ORBITER/SRC/LIB/GALOIS/rainbow_cliques.C
ORBITER/SRC/LIB/GALOIS/rank_checker.C
ORBITER/SRC/LIB/GALOIS/set_of_sets.C
ORBITER/SRC/LIB/GALOIS/sorting.C
ORBITER/SRC/LIB/GALOIS/spreadsheet.C
ORBITER/SRC/LIB/GALOIS/subfield_structure.C
ORBITER/SRC/LIB/GALOIS/super_fast_hash.C
ORBITER/SRC/LIB/GALOIS/tensor.C
ORBITER/SRC/LIB/GALOIS/tree.C
ORBITER/SRC/LIB/GALOIS/tree_node.C
ORBITER/SRC/LIB/GALOIS/unipoly.C
ORBITER/SRC/LIB/GALOIS/unusual.C
ORBITER/SRC/LIB/GALOIS/util.C
ORBITER/SRC/LIB/GALOIS/vector_hashing.C
ORBITER/SRC/LIB/INCIDENCE/
ORBITER/SRC/LIB/INCIDENCE/dynamic_memory.C
ORBITER/SRC/LIB/INCIDENCE/geo_parameter.C
ORBITER/SRC/LIB/INCIDENCE/geometric_tests.C
ORBITER/SRC/LIB/INCIDENCE/inc_gen_global.C
ORBITER/SRC/LIB/INCIDENCE/incidence.h
ORBITER/SRC/LIB/INCIDENCE/incidence_global.C
ORBITER/SRC/LIB/INCIDENCE/makefile
ORBITER/SRC/LIB/INCIDENCE/mckay.C
ORBITER/SRC/LIB/INCIDENCE/packing.C
ORBITER/SRC/LIB/INCIDENCE/point_line.C
ORBITER/SRC/LIB/INCIDENCE/refine_3design.C
ORBITER/SRC/LIB/INCIDENCE/refine_columns.C
ORBITER/SRC/LIB/INCIDENCE/refine_rows.C
ORBITER/SRC/LIB/INCIDENCE/tdo_data.C
ORBITER/SRC/LIB/INCIDENCE/tdo_scheme.C
ORBITER/SRC/LIB/makefile
ORBITER/SRC/LIB/makefile~
ORBITER/SRC/LIB/orbiter.h
ORBITER/SRC/LIB/orbiter.h~
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/compute_stabilizer.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/extension.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_classify.C
```

```
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_combinatorics.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_draw.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_init.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_io.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/generator_trace.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/makefile
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle_downstep.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle_downstep_subspace_action.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle_io.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle_upstep.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/oracle_upstep_subspace_action.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/recognize.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/set_stabilizer_compute.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/snakes_and_ladders_global.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/snakesandladders.h
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/upstep_work.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/upstep_work_subspace_action.C
ORBITER/SRC/LIB/SNAKES_AND_LADDERS/upstep_work_trace.C
ORBITER/SRC/LIB/TOP_LEVEL/
ORBITER/SRC/LIB/TOP_LEVEL/analyze_group.C
ORBITER/SRC/LIB/TOP_LEVEL/choose_points_or_lines.C
ORBITER/SRC/LIB/TOP_LEVEL/decomposition.C
ORBITER/SRC/LIB/TOP_LEVEL/elliptic_curve.C
ORBITER/SRC/LIB/TOP_LEVEL/exact_cover.C
ORBITER/SRC/LIB/TOP_LEVEL/exact_cover_solver.C
ORBITER/SRC/LIB/TOP_LEVEL/extra.C
ORBITER/SRC/LIB/TOP_LEVEL/factor_group.C
ORBITER/SRC/LIB/TOP_LEVEL/incidence_structure.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph_database.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph_files.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph_global.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph_testing.C
ORBITER/SRC/LIB/TOP_LEVEL/isomorph_trace.C
ORBITER/SRC/LIB/TOP_LEVEL/knarr.C
ORBITER/SRC/LIB/TOP_LEVEL/makefile
ORBITER/SRC/LIB/TOP_LEVEL/orbit_of_sets.C
ORBITER/SRC/LIB/TOP_LEVEL/orbit_of_subspaces.C
ORBITER/SRC/LIB/TOP_LEVEL/orbit_rep.C
ORBITER/SRC/LIB/TOP_LEVEL/polar.C
ORBITER/SRC/LIB/TOP_LEVEL/projective_space.C
ORBITER/SRC/LIB/TOP_LEVEL/recoordinatize.C
ORBITER/SRC/LIB/TOP_LEVEL/representatives.C
ORBITER/SRC/LIB/TOP_LEVEL/search_blocking_set.C
ORBITER/SRC/LIB/TOP_LEVEL/singer_cycle.C
ORBITER/SRC/LIB/TOP_LEVEL/subspace_orbits.C
ORBITER/SRC/LIB/TOP_LEVEL/top_level.h
ORBITER/SRC/LIB/TOP_LEVEL/translation_plane.C
ORBITER/SRC/LIB/TOP_LEVEL/translation_plane2.C
ORBITER/SRC/LIB/TOP_LEVEL/w3q.C
```

```
ORBITER/SRC/LIB/TOP_LEVEL/young.C
ORBITER/SRC/makefile
```

References

[1] Anton Betten, Michael Braun, Harald Fripertinger, Adalbert Kerber, Axel Kohnert, and Alfred Wassermann. *Error-correcting linear codes*, volume 18 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 2006. Classification by isometry and applications, With 1 CD-ROM (Windows and Linux).

[2] Adalbert Kerber. *Applied finite group actions*, volume 19 of *Algorithms and Combinatorics*. Springer-Verlag, Berlin, second edition, 1999.