University Of Oxford, Somerville College

Final Honour School of Mathematics and Computer Science

Constructing Distance-Regular Graphs in SageMath

Ivo Maffei

Supervisor: Dr Dmitrii Pasechnik

Trinity Term 2020

Abstract

Graphs are combinatorial objects that are widely studied and used in almost every field of computer science. However, many graphs are described via mathematical objects and lack concrete implementations. The aim of this project is to build a database of distance-regular graphs in the computer algebra system SageMath.[18] We explore the definitions of those graphs and translate them into practical algorithms that are subsequently implemented within SageMath. Throughout this process we will meet various combinatorial objects related to distance-regular graphs and, when needed, implement them as well.

Acknoledgements

I would like to thank my parents for their immense support that made this four-year dream possible. Without their help, none of this would have happened and I will always be immensely grateful to them. I want to thank my siblings for their moral support during this long journey and especially my sister for the great help during the application process. I thank my grandparents for always caring about me. They took upon themselves the responsibility to restore my strengths after each exhausting term and they succeeded brilliantly. I would like to thank all of my friends for their support during the hard periods and the great memories we collected throughout this unforgettable journey.
Finally, I would like to thank my supervisor for guiding me through this project and my tutors at Somerville for helping me choose my academic path.

To my parents

Contents

1 Introduction **6**

1.1 Report Structure 7

2 Background knowledge and notation **8**

2.1 Graph Theory 8

2.2 Design Theory 11

2.3 Linear Algebra 13

3 Project Approach **15**

3.1 Other combinatorial objects 16

3.2 Efficiency 16

3.3 Testing 17

4 New constructions **18**

4.1 Graphs with Unbounded Diameter 22

4.1.1 Double, Half and Fold 22

4.1.2 Dual Polar graphs 23

4.1.3 Doubled Odd graph 25

4.1.4 Bilinear, Alternating and Hermitian form graphs 25

4.1.5 Halved Cube 26

4.1.6 Grassmann and Doubled Grassmann graphs 26

4.2 Graphs with Unbounded Order 27

4.2.1 Generalised Polygons 27

4.2.2 Generalised Quadrangles 29

4.2.3 Unitary Nonisotropic graph 34

4.2.4 Hermitian cover 35

4.2.5 Pasechnik and Brouwer-Pasechnik 36

4.2.6 Transversal Designs 37

4.2.7 BIBD graphs 39

4.2.8 Taylor graphs 40
4.2.9 Denniston graphs 41
4.2.10 Association Schemes 43
4.2.11 Preparata graph 44
4.2.12 Symplectic covers 45
4.2.13 Coset Graphs of Linear Codes 46
4.2.14 Kasami codes 47
4.2.15 AB graph 50

5 Conclusion **51**

5.1 Future work 51

6 References **53**

A Code **55**

1 Introduction

Graphs are among the most widely used and studied concepts in combinatorics, and distance-regular graphs are a peculiar class of graphs with many symmetric properties. In particular, they are symmetric with respect to distances, i.e. given a pair of vertices the number of vertices at distance i from the first and j from the second only depends on the distance between the chosen pair. However, distance-regular graphs are not only important within graph theory because, as we will see throughout this report, many other combinatorial objects are closely linked to them. It follows that many researchers studying combinatorics, coding theory, design theory, finite geometry or even representation theory will meet distance-regular graphs. Among the first concerns for such researchers will be whether a distance-regular graph exists for a given set of parameters and if so to construct an example for further analysis. However, not many resources for such problems are available. One can find an online database of feasible intersection arrays,[4] yet no graphs are given and for many entries existence has not been proven. Alternatively, distanceregular.org[1] is the only database where graphs are given together with their parameters, yet it is limited in size. As a result, a dedicated researcher will likely have to consult the monograph 'Distance-Regular Graphs' by Brouwer *et al.* (1989)[5] or the one by van Dam *et al.* (2016).[19] This project aims to ease such burden by providing a database of constructions for distance-regular graphs in the open-source computer algebra system SageMath.[18] Moreover, this report describes most constructions in their entirety without relying on expertise in any field of combinatorics. In this regard, we go beyond what is available in the two aforementioned monographs and also provide a variety of graphs which goes much further than the online database distanceregular.org.[1] However, building a complete database of distance-regular graphs is a herculean task so we had to limit the scope of this project. The first restriction we impose is to construct a graph per set of parameters. That is, when more examples are known we will not try to enumerate them all, but only provide one. This design choice emulates SageMath's approach to generating other combinatorial objects. Despite this seeming a huge limitation, the sheer number of distance-regular graphs left is still beyond what one can hope to accomplish during an academic year. As such we limited ourselves to construct all infinite families of graphs described in the monographs by Brouwer *et al.* (1989)[5] and by van Dam *et al.* (2016).[19] Nevertheless, the reader should be pleased to know that we went further and also constructed many "sporadic" examples.

1.1 Report Structure

1. Chapter 2 provides the reader with the background material needed for the remainder of this report;
2. Chapter 3 describes the broad design choices that lead to the current structure of our SageMath module;
3. Chapter 4 contains descriptions for the constructions implemented;
4. Chapter 5 summarises the results achieved and includes a reflection on future work.

2 Background knowledge and notation

In this chapter we cover the most important definitions that will be used across many constructions. However, we do assume some very basic knowledge of graphs, linear algebra, group theory and finite fields. Throughout this report we try to minimise the amount of theory required for understanding our constructions and so we often develop ad-hoc definitions and theorems, which therefore may not appear as “standard” to experts in the field.

2.1 Graph Theory

In this section we will follow the approach of Brouwer *et al.* (1989).[5]
Let Γ be a graph. Here we will only deal with connected graphs, so from now on assume Γ is connected. We denote its set of vertices by V_Γ and its set of edges by E_Γ. We will only deal with unordered graphs and write $u \sim v$ to mean $(u, v), (v, u) \in E_\Gamma$. We call $|V_\Gamma|$ and $|E_\Gamma|$ respectively the *order* and *size* of Γ. Moreover, if Γ is regular, then the degree of its vertices is called the *valency* of Γ. We define $\Gamma_i(v) = \{w \in V_\Gamma \mid d(v, w) = i\}$ where $d(v, w)$ denotes the distance between the two vertices.

Definition 2.1. *We say that* Γ *is a* distance-regular *graph with diameter* d *if it has an intersection array, i.e. there are numbers* $[b_0, ...b_{d-1}; c_1, ..., c_d]$ *such that given any two vertices* u, v *at distance* i *we have* $b_i = |\Gamma_1(u) \cap \Gamma_{i+1}(v)|$ *and* $c_i = |\Gamma_1(u) \cap \Gamma_{i-1}(v)|$.

From the above definition, we note that $c_1 = 1$ and that b_0 is the valency of Γ. The numbers b_i, c_i together with $a_i = b_0 - b_i - c_i$ are called the *intersection numbers* of Γ.

If Γ is distance-regular with diameter 1, then it is a complete graph, while if it has diameter 2, it is called strongly regular. Strongly regular graphs are often studied separately from other distance-regular graphs and here we will always assume that the diameter is greater than 2.

We now define a few ways to create new graphs from old ones.

Definition 2.2. *Given a graph* Γ *its* bipartite double $\mathrm{D}\Gamma$ *is defined as follow*

$$V_{\mathrm{D}\Gamma} = V_\Gamma \times \{0,1\}$$
$$E_{\mathrm{D}\Gamma} = \{ \ (\ (u,a),(v,b)\ ) \mid (u,v) \in E_\Gamma \wedge a \neq b \ \}$$

Similarly,

Definition 2.3. *The* extended bipartite double $\mathrm{ED}\Gamma$ *is*

$$V_{\mathrm{ED}\Gamma} = V_{\mathrm{D}\Gamma}$$
$$E_{\mathrm{ED}\Gamma} = \{ \ (\ (u,a),(u,b)\ ) \mid u \in V_\Gamma \wedge a \neq b \ \} \cup E_{\mathrm{D}\Gamma}$$

As the name says, both $D\Gamma$ and $\mathrm{ED}\Gamma$ are bipartite. Quite a few distance-regular graphs are the (extended) bipartite double of other distance-regular (or strongly regular) graphs. Conversely to the above definition, one has

Definition 2.4. *Let* Γ *be a bipartite graph where* $V_\Gamma = X \cup Y$ *such that* X, Y *are the two parts of* Γ*. The* half *of* Γ *is* $\frac{1}{2}\Gamma$ *defined by*

$$V_{\frac{1}{2}\Gamma} = X$$
$$E_{\frac{1}{2}\Gamma} = \{ \ (x,y) \mid d(x,y) = 2 \ \mathit{in} \ \Gamma \ \}$$

If Γ is distance-regular, Brouwer *et al.* (1989)[5] prove that so is $\frac{1}{2}\Gamma$. Note that halving and (extended) bipartite doubling are not inverses of each other.

A more general approach is the to consider the *distance graph.*

Definition 2.5. *Given a graph* Γ *the* distance-i *graph* Γ_i *is defined by*

$$V_{\Gamma_i} = V_\Gamma$$
$$E_{\Gamma_i} = \{ \ (u,v) \mid d(u,v) = i \ \mathit{in} \ \Gamma \ \}$$

With the above definition we can talk about antipodal graphs:

Definition 2.6. *A graph* Γ *of diameter* d *is called* antipodal *if* Γ_d *is a disjoint union of cliques.*

Definition 2.7. *Given an antipodal graph* Γ *of diameter* d*, its* antipodal quotient *(or* folding*) is* $\hat{\Gamma}$ *where*

$$V_{\hat{\Gamma}} = \{c \mid c \textit{ is a maximal clique of } \Gamma_d\}$$

$$E_{\hat{\Gamma}} = \{(c_1, c_2) \mid \textit{ there is an edge between } c_1, c_2 \textit{ in } \Gamma\}$$

The relation between Γ and $\hat{\Gamma}$ is quite important, so we have a special name for it

Definition 2.8. *Let* Γ *be a regular antipodal graph of diameter* d *and assume all maximal cliques in* Γ_d *have size* r*. If* $\hat{\Gamma}$ *is regular with the same valency of* Γ*, then we call* Γ *an* antipodal r-cover *of* $\hat{\Gamma}$*.*

There isn't any complete classification of distance-regular graphs, but Brouwer *et al.* (1989)[5] identify three important subclasses and they state that all known graphs of diameter greater than 8 belong to at least one of those. This may not hold true to this day, but we confirm the result for all known families with unbounded diameter.

Definition 2.9. *We say that* Γ *has* classical parameters (d, b, α, β) *if it has diameter* d *and*

$$b_i = \left(\begin{bmatrix} d \\ 1 \end{bmatrix}_b - \begin{bmatrix} i \\ 1 \end{bmatrix}_b \right) \left(\beta - \alpha \begin{bmatrix} i \\ 1 \end{bmatrix}_b \right) \tag{2.1.1}$$

$$c_i = \begin{bmatrix} i \\ 1 \end{bmatrix}_b \left(1 + \alpha \begin{bmatrix} i-1 \\ 1 \end{bmatrix}_b \right) \tag{2.1.2}$$

where

$$\begin{bmatrix} i \\ j \end{bmatrix}_b = \begin{cases} \displaystyle\prod_{l=0}^{j-1} \frac{b^i - b^l}{b^j - b^l} & b \neq 1 \\ \displaystyle\binom{i}{j} & b = 1 \end{cases}$$

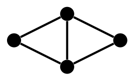
Definition 2.10. *We call* Γ *a* psuedo partition graph *of diameter* d *if there are numbers* α *and* $m \in \{2d, 2d+1\}$ *such that*

$$b_i = (m-i)(1+\alpha(m-i-1)) \qquad c_i = i(1+\alpha(i-1)) \qquad \textit{for } 0 \le i < d \tag{2.1.3}$$

$$c_d = d(2d+2-m)(1+\alpha(d-1)) \tag{2.1.4}$$

Definition 2.11. *We say that* Γ *is a* near polygon *if ithere is a number* λ *such that*

$$b_i = b_0 - (\lambda+1)c_i \qquad \textit{for } 0 \le i < d \tag{2.1.5}$$

and it does not have a subgraph of the form: 

Finally, let me introduce the class of distance-transitive graphs since this notion will be useful for some constructions.

Definition 2.12. *Given a graph* Γ *its automorphism group* $\mathrm{Aut}(\Gamma)$ *is the group of all permutations* $\pi : V_\Gamma \to V_\Gamma$ *such that* $v \sim w \iff \pi(v) \sim \pi(w)$.

Definition 2.13. *A graph* Γ *with diameter* d *is* distance-transitive *if* $\mathrm{Aut}(\Gamma)$ *acts transitively on the set* E_{Γ_i} *for* $0 \le i \le d$. *That is for any four vertices* v_1, v_2, u_1, u_2 *such that* $d(v_1, u_1) = d(v_2, u_2)$ *there is* $\pi \in \mathrm{Aut}(\Gamma)$ *satisfying* $\{\pi(v_1), \pi(u_1)\} = \{v_2, u_2\}$.

In particular, $\mathrm{Aut}(\Gamma)$ acts transitively on E_Γ. Hence, given $\mathrm{Aut}(\Gamma)$ and $(u, v) \in E_\Gamma$, one can easily obtain the whole E_Γ.

2.2 Design Theory

Definition 2.14. *An* incidence structure *(or* block design*) is tuple* $(P, \mathcal{B}, I)$ *where* P, $\mathcal{B}$ *are sets and* I *is a symmetric relation between the two. We call the elements of* P *points, the elements of* $\mathcal{B}$ *blocks and* I *is called the incidence relation.*

We will often represent the elements of $\mathcal{B}$ as subsets of P and interpret I as $\in$. However, to avoid loosing generality we allow for duplicated blocks. Some authors (Brouwer *et al.*, 1989)[5] define an incidence structure using $I = \in$ and require blocks to be unique, yet SageMath does not require uniqueness and so we won't. When talking about the incidence structures we will sometimes refer to blocks as lines and use expressions such as "a point lies on a line" or "two lines intersect". In particular, we say that two points are *collinear* if there is a block incident to both.

Each incident structure has a *dual*:

Definition 2.15. *Given an incidence structure $D = (P, \mathcal{B}, I)$ its dual is the incidence structure $D^* = (\mathcal{B}, P, I)$.*

The incidence structure we will use the most in this report comes from the idea of projective geometry. There are various ways to define when an incidence structure is a projective space, but here we will use the following:

Definition 2.16. *Let V be a vector space. The projective space $P_k(V) = (P, \mathcal{L})$ is an incidence structure where $P = \{\langle v \rangle \mid v \in V \setminus \{0\}\}$ and $\mathcal{L} = \{L \mid L \leq V \wedge \dim(L) = k\}$. We call P the set of projective points and, if $k = 2$, we call $\mathcal{L}$ the set of projective lines.*

More generally we call a d-subspace of V a projective $(d-1)$-subspace of V. Since we will almost always pick $V = (\mathbb{F}_q)^n$, we will write $PG(n-1, q)$ for $P_2((\mathbb{F}_q)^n)$ and $PG(n-1, e-1, q)$ for $P_e((\mathbb{F}_q)^n)$. Note also that we will represent $L \in \mathcal{L}$ as a subset of P.

Given an incidence structure $D = (P, \mathcal{B})$ one can obtain quite a few graphs out of it. In particular, we define the following three methods:

Definition 2.17. *Given an incidence structure $(P, \mathcal{B})$ its* point graph *is the graph with vertices P where $p \sim q$ if they are collinear.*

Definition 2.18. *Given an incidence structure $(P, \mathcal{B})$ its* line graph *is the graph with vertices $\mathcal{B}$ where $B \sim B'$ if they have a common point.*

Definition 2.19. *Given an incidence structure* $(P, \mathcal{B})$ *its* incidence graph *is the bipartite graph with vertices* $P \times \mathcal{B}$ *where* $p \sim B$ *if* $p \in B$.

Note that the point graph of the dual of D is the line graph of D and that their incidence graphs are equal.

2.3 Linear Algebra

Most of the constructions in this project will involve some manipulation of vector spaces. Following Grove (2002),[14] we will define a few different *forms* which will appear quite often in this report.

Definition 2.20. *Let* V *be a vector space over a field* K. *Then a* bilinear form *is a map* $\langle -, - \rangle : V \times V \to K$ *satisfying*

$$\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle \qquad \langle \lambda v, w \rangle = \lambda \langle v, w \rangle$$

$$\langle v, u + w \rangle = \langle v, u \rangle + \langle v, w \rangle \qquad \langle v, \lambda w \rangle = \lambda \langle v, w \rangle$$

Note that we can associate a matrix A to any given bilinear form $\langle -, - \rangle$ so that $\langle v, w \rangle = v^T A w$. Conversely, any matrix defines a bilinear form in the above way.

Definition 2.21. *Let* V *be a vector space over a field* K. *Then a map* $Q : V \to K$ *is a* quadratic form *if*

$$Q(\lambda v) = \lambda^2 Q(v)$$

$$B(v, w) = Q(v + w) - Q(v) - Q(w) \text{ is bilinear}$$

We say that a vector $v \neq 0$ *is* isotropic *if* $Q(v) = 0$. *In a similar manner, we say that* $W \leq V$ *is* totally isotropic *if* $\forall w \in W \;\; Q(w) = 0$.

Intuitively, we can associate a matrix A to Q so that $Q(v) = v^T A v$.

Definition 2.22. *Let V be a vector space over a field $\mathbb{F}_{q^2}$. Then we say that $\langle -,- \rangle : V \times V \to \mathbb{F}_{q^2}$ is a* Hermitian form *if*

$$\langle v+u, w\rangle = \langle v,w\rangle + \langle u,w\rangle \quad \langle \lambda v, w\rangle = \lambda\langle v,w\rangle$$
$$\langle v,w\rangle = \overline{\langle w,v\rangle} \quad \textit{where } \overline{x} = x^q$$

Similarly to the bilinear forms, we can find a matrix A such that $\langle v,w\rangle = v^T A\overline{w}$. Conversely, any matrix A with the property that $A = \overline{A}^T$ defines an Hermitian form. The notion of isotropicity extends to Hermitian forms where we say that $v \neq 0$ is isotropic if $\langle v,v\rangle = 0$.

The above two forms are associated to some important groups that we define below.

Definition 2.23. *Given a quadratic form, the* general orthogonal *group $GO(n,q)$ is the group of all linear maps on $(\mathbb{F}_q)^n$ that preserve the quadratic form.*

It turns out that there are essentially three different quadratic forms and so three non-isomorphic groups that are often denoted by $GO(2k+1,q)$, $GO^+(2k,q)$ and $GO^-(2k,q)$.

Definition 2.24. *Given an Hermitian form, the* general unitary *group $GU(n,q)$ is the group of all linear maps on $(\mathbb{F}_{q^2})^n$ that preserve the Hermitian form.*

This time all Hermitian form are essentially equivalent and so there is only one group $GU(n,q)$.

We will often talk about the action of the above groups on a vector space V. Unless stated otherwise, such action is assumed to be the standard action on projective points. That is, if $M \in G$, then M acts on the projective point $\langle v\rangle$ via $M \cdot \langle v\rangle = \langle Mv\rangle$. In particular, if v is "normalised", i.e. its first non-zero entry is 1, then M sends v to the "normalisation" of Mv.

3 Project Approach

This chapter will explain the general structure of the SageMath module implemented and address a few design choices.

Following SageMath's praxis, all constructions will be available through the function `distance_regular_graph`. This can be used to build a graph given its intersection array or to check whether such graph exists. Moreover, it is the only function that should be available outside the module. We want the design of such function to be as simple as possible and flexible enough to make it easy to add/modify constructions. Hence, all sporadic constructions are added to a dictionary object mapping intersection arrays to constructions. Similarly, we have a list of pairs $(\texttt{is\_f}, \texttt{f})$ where

$$\texttt{is\_f}(\texttt{array}) = \begin{cases} \texttt{params} & \text{if } \texttt{f}(\texttt{params}).\texttt{intersection\_array}() == \texttt{array} \\ \texttt{False} & \text{otherwise} \end{cases} \tag{3.0.1}$$

```
def distance_regular_graph( array ):
    check array makes sense
    if array in sporadic_database:
        return sporadic_database[array]
    for (is_f, f) in list_of_constructions:
        t = is_f(array)
        if t is not False: return f(t)
    check feasibility of array
```

This approach is very flexible as it allows the addition of new constructions easily. In order to check whether an intersection array is feasible, i.e. there might be a graph for it, we decided to take advantage of another SageMath package. In particular, Vidali developed the SageMath module `drg`[23] and used it to prove nonexistence results.[22] During the course of the project we reported a few bugs, but we did not contribute to the package. Despite the great work of Vidali, the feasibility check is quite slow. Hence, we decided to postpone its use after we known that the graph can't be built. However, we will perform some basic checks at the start of the function to avoid repeating those checks in all "selection" functions `is_f`.

Most of the constructions we will see build a graph by finding its set of edges. This can be achieved in essentially three ways:

1. Generate E_Γ using other mathematical objects (such as groups);
2. Iterate through $V_\Gamma \times V_\Gamma$ and check for adjacency;
3. Iterate through V_Γ and construct a set of neighbours for any given vertex.

It follows that when describing a construction is enough to show a way to efficiently check for adjacency or how to construct the set of neighbours for any given vertex.

3.1 Other combinatorial objects

As we will see in section 4, some constructions rely on first building other combinatorial objects. Often these objects are already constructed in SageMath, yet SageMath won't provide *all* known examples of such objects. It follows that we would need to implement some constructions of those combinatorial objects to obtain our distance-regular graphs. However, we need to limit the scope of this project, hence new constructions will be added only when necessary for testing purposes. On the other hand, we will make sure that the graph-related code won't need to change if new objects are added.

3.2 Efficiency

Efficiency of the constructions is an important matter as naive implementations can easily be very slow. However, since the start of the project, the concept of "slow" changed drastically. SageMath is not built with speed in mind as the choice of Python should highlight. Most SageMath's object are very flexible and carry a lot of features, but this has the drawback that they end up being relatively slow to handle. Moreover, code clarity is always preferred to hacks and tricks to improve performance, hence SageMath can be significantly slower than other systems like GAP.[12] It follows that it is quite common to have SageMath run for a few minutes when building big objects. In particular, the slower constructions of the project are those that require other combinatorial objects. It follows that we mainly focused on the correctness of our implementations and worry about their efficiency only when they were slow even for SageMath's standards. Nevertheless, to try to mitigate the performance issues, we developed everything using Cython[2] which is a language with Python like syntax that gets compiled to C. Despite our efforts, some constructions may take up to 40 minutes when building graphs of size in the order of 10^5. See A.3 for sample outputs and timings.

3.3 Testing

All constructions used are proved to be correct by their original designers. In addition, in section 4 we will prove that our algorithms are equivalent to the given constructions. Nevertheless, the actual implementation may have bugs. It follows that testing was a crucial aspect of the project. Our main concern is that the graphs built have the intersection array they are expected to. Hence we developed testing scripts with this purpose. For any infinite construction, we built functions that generate all small enough possible inputs and their expected intersection arrays. Thus, testing boils down to iterating through all such inputs, constructing the graph and then checking that its intersection array matches the expected result. This checks that all constructions work as expected. Moreover, this ensures that the invariant 3.0.1 is respected. Throughout this process we discovered some bugs in the `drg` package[23] developed by Vidali. However, all such bugs were reported and fixed. See A.2 for the actual testing scripts used.

4 New constructions

Explaining and proving the correctness of all constructions will require much more space than what is available. Hence, we only focus on the constructions for infinite families of graphs. However, to provide the reader with a general overview, we grouped all the constructions in the tables 1, 2 and 3. Table 2 contains the families of graphs with unbounded diameter that can be built with `distance_regular_graph`. Other such families are known, yet any family omitted will have their parameters covered by some other family listed in the table. Table 3 lists all constructions of families of graphs with bounded diameter but unbounded order. Table 3 may not be a comprehensive list as table 2 is. However, it does include all constructions mentioned in the monographs by Brouwer *et al.* (1989)[5] and by van Dam *et al.* (2016).[19] Table 1, instead, includes all new "sporadic" constructions.

Name of graph	**intersection array**
Ivanov Ivanov Faradjev	$[7, 6, 4, 4, 4, 1, 1, 1; 1, 1, 1, 2, 4, 4, 6, 7]$
Double of truncated binary Golay code graph	$[22, 21, 20, 16, 6, 2, 1; 1, 2, 6, 16, 20, 21, 22]$
Double of binary Golay code graph	$[23, 22, 21, 20, 3, 2, 1; 1, 2, 3, 20, 21, 22, 23]$
Twice shortened binary Golay code graph	$[21, 20, 16, 6, 2, 1; 1, 2, 6, 16, 20, 21]$
Thrice shortened binary Golay code graph	$[21, 20, 16, 9, 2, 1; 1, 2, 3, 16, 20, 21]$
Shortened binary Golay code graph	$[22, 21, 20, 3, 2, 1; 1, 2, 3, 20, 21, 22]$
Shortened extended ternary Golay code graph	$[22, 20, 18, 2, 1; 1, 2, 9, 20, 22]$
Double of Hoffman Singleton graph	$[7, 6, 6, 1, 1; 1, 1, 6, 6, 7]$
Double of Sims Gewirtz graph	$[10, 9, 8, 2, 1; 1, 2, 8, 9, 10]$
Double of strongly regular graph $(77, 16, 0, 4)$	$[16, 15, 12, 4, 1; 1, 4, 12, 15, 16]$

Double og Higman Sims graph	$[22, 21, 16, 6, 1; 1, 6, 16, 21, 22]$
Coxeter graph	$[3, 2, 2, 1; 1, 1, 1, 2]$
Lint Schrijver graph	$[6, 5, 5, 4; 1, 1, 2, 6]$
Doubly truncated Witt graph	$[7, 6, 4, 4; 1, 1, 1, 6]$
Distance 3 doubly truncated Golay code graph	$[9, 8, 6, 3; 1, 1, 3, 8]$
J_2 graph	$[10, 8, 8, 2; 1, 1, 4, 5]$
Foster graph	$[6, 4, 2, 1; 1, 1, 4, 6]$
Conway Smith graph	$[10, 6, 4, 1; 1, 2, 6, 10]$
Shortened ternary Golay code graph	$[20, 18, 4, 1; 1, 2, 18, 20]$
Locally GQ $(4, 2)$ graph	$[45, 32, 12, 1; 1, 6, 32, 45]$
$3O_7(3)$ graph	$[117, 80, 24, 1; 1, 12, 80, 117]$
Extended binary Golay code graph	$[24, 23, 22, 21; 1, 2, 3, 24]$
Leonard graph	$[12, 11, 10, 7; 1, 2, 5, 12]$
A.25 Cocliques of Hoffman Singleton graph	$[15, 14, 10, 3; 1, 5, 12, 15]$
Truncated binary Golay code graph	$[22, 21, 20; 1, 2, 6]$
Binary Golay code graph	$[23, 22, 21; 1, 2, 3]$
Large Witt graph	$[30, 28, 24; 1, 3, 15]$
Truncated Witt graph	$[15, 14, 12; 1, 1, 9]$
Extended ternary Golay code graph	$[24, 22, 20; 1, 2, 12]$
Doubly truncated binary Golay code graph	$[21, 20, 16; 1, 2, 12]$

Table 1: New sporadic graphs implemented

Name of family	**parameters**	**intersection array**
Johnson graph*	n, d, $n \geq 2d$	cl.p. $(d, 1, 1, n-d)$
4.1.1 Folded Johnson graph	n	$b_i = (n-i)^2$, $c_i = i^2$ $c_d = d^2(2d+2-n)$, $d = \lfloor \frac{n}{2} \rfloor$
Odd graph*	n	$b_i = n - c_i$, $c_i = \lfloor \frac{i+1}{2} \rfloor$, $d = n-1$
4.1.3 Doubled odd graph	n	$b_i = n + 1 - c_i$, $c_i = \lfloor \frac{n+1}{2} \rfloor$, $d = 2n+1$
Hamming graph*	d, e	cl.p. $(d, 1, 0, e-1)$
4.1.5 Halved Cube	n	cl.p. $(\lfloor \frac{n}{2} \rfloor, 1, 2, 2\lceil \frac{n}{2} \rceil - 1)$
Folded cube*	n	$b_i = n - i$, $c_i = i$ $c_d = d(2d+2-n)$, $d = \lfloor \frac{n}{2} \rfloor$
4.1.1 Folded halved cube	n	$b_i = (n-i)(2n-2i-1)$, $c_i = i(2i-1)$ $c_d = d(2d+2-n)(2d-1)$, $d = \lfloor \frac{n}{2} \rfloor$
4.1.4 Bilinear form graph	d, e, q, $d \leq e$	cl.p. $(d, q, q-1, q^e - 1)$
4.1.4 Alternating form graph	n, q	cl.p. $(\lfloor \frac{n}{2} \rfloor, q^2, q^2 - 1, q^\alpha - 1)$ $\alpha = 2\lceil \frac{n}{2} \rceil - 1$
4.1.4 Hermitian form graph	d, q^2	cl.p. $(d, -q, -q-1, -(-q)^d - 1)$
Polygon*	n	$d = \lfloor \frac{n}{2} \rfloor$, $b_0 = 2$, $c_d = 2d + 2 - n$ $b_i = 1$, $c_i = 1$ for $1 \leq i \leq d-1$
4.1.2 Dual polar graph[†]	d, q, e	cl.p $(d, q, 0, q^e)$, $e \in \{0, 1, 2, \frac{3}{2}, \frac{1}{2}\}$
Dual polar graph*	$e = \frac{1}{2}$, d, q^2	cl.p. $\left(d, -q, q\dfrac{1+q}{1-q}, q\dfrac{(-q)^d + 1}{1-q}\right)$
4.1.1 Halved dual polar graph	n, q	cl.p. $\left(\lfloor \frac{n}{2} \rfloor, q^2, q^2 + q, \begin{bmatrix} m+1 \\ 1 \end{bmatrix}_q - 1\right)$ $m = 2\lceil \frac{n}{2} \rceil - 1$
4.1.6 Grassmann graph	n, d, q, $n \geq 2d$	cl.p. $\left(d, q, q, \begin{bmatrix} n-d+1 \\ 1 \end{bmatrix}_q - 1\right)$
4.1.6 Doubled Grassmann graph	e, q	$b_i = \begin{bmatrix} e+1 \\ 1 \end{bmatrix}_q - c_i$, $c_i = \begin{bmatrix} \lfloor \frac{i+1}{2} \rfloor \\ 1 \end{bmatrix}_q$ $d = 2e+1$

Table 2: Families of graphs with intersection arrays of unbounded length; cl.p. stands for classical parameters; q is assumed to be a prime power
* the family was already implemented in SageMath
[†] for $e = 1, \frac{1}{2}, \frac{3}{2}$ the family was already imlpemented in SageMath

Name of family	**parameters**	**intersection array**
4.2.1 Generalised dodecagon*	s, t	$[st+s, st, st, st, st, st; 1, 1, 1, 1, 1, t+1]$
4.2.1 Generalised octagon†*	s, t	$[st+s, st, st, st; 1, 1, 1, t+1]$
4.2.1 Generalised hexagon*	s, t	$[st+s, st, st; 1, 1, t+1]$
4.2.2 GQ with spread†	s, t	$[st, s(t-1), 1; 1, t-1, st]$
4.2.3 Unitarty nonisotropic	q	$[q^2-q, q^2-q-2, q+1; 1, 1, q^2-2q]$ $q > 2$
4.2.4 Hermitian cover*	q, r where $r \mid q^2-1$	$[q^3, (q^3-1)(r-1)/r, 1, 1, (q^3-1)/r, q^3]$ $[q^3, (r-1)(q+1)m, 1, 1, (q+1)m, q^3]$ where $m = (q^2-1)/r$
4.2.5 Brouwer-Pasechnik	q	$[q^3-1, q^3-q, q^3-q^2+1; 1, q, q^2-1]$
4.2.5 Pasechnik	q	$[q^3, q^3-1, q^3-q, q^3-q^2+1; 1, q, q^2-1, q^3]$
4.2.6 TD graph†	m, u	$[mu, mu-1, (m-1)u, 1; 1, u, mu-1, mu]$
4.2.7 BIBD graphs†	v, k	$[k, k-1, k-\lambda; 1, \lambda, k]$ where $\lambda = \frac{k(k-1)}{v-1}$
4.2.8 Taylor graph†	k, μ	$[k, \mu, 1; 1, \mu, k]$
4.2.9 Denniston graphs	n where $n = 2^i$ for some $i > 0$	$[n^2-n+1, n(n-1), n(n-1), n;$ $1, 1, (n-1)^2, n^2-n+1]$
4.2.10 Association schemes†	n, r	$[rn, (r-1)(n-1)/r, 1; 1, (n-1)/r, n]$
4.2.11 Preparata graphs	t, i	$[2^{2t}-1, 2^{2t}-2^{i+1}, 1; 1, 2^{i+1}, 2^{2t}-1]$
4.2.12 Symplectic cover	q, n, r where $r \mid q$; $n \geq 2$ even	$[q^n-1, (r-1)q^n/r, 1; 1, q^n/r, q^n-1]$
4.2.14 Kasami gaphs*	s, t prime powers of 2	$[q^2-1, q^2-q, 1; 1, q, q^2-1]$ if $s = q^2, t = q$ $[q^{2j+1}-1, q^{2j+1}-q, q^{2j}(q-1)+1; 1, q, q^{2j}-1]$ if $s = q^{2j+1}, t = q^m$
4.2.14 Extended Kasami graphs*	s, t prime powers of 2	$[q^2, q^2-1, q^2-q, 1; 1, q, q^2-1, q^2]$ if $s = q^2, t = q$ $[q^{2j+1}, q^{2j+1}-1, q^{2j+1}-q, q^{2j}(q-1)+1;$ $1, q, q^{2j}-1, q^{2j+1}]$ if $s = q^{2j+1}, t = q^m$
4.2.15 AB graphs	n	$[2^n-1, 2^n-2, 2^{n-1}+1; 1, 2, 2^{n-1}-1]$

Table 3: Families of graphs with fixed diameter and unbounded order
* other constraints on the parameters are needed, look at the related section in chapter 4
† the construction relies on the existence of other combinatorial objects, which might not exist for certain parameters

4.1 Graphs with Unbounded Diameter

4.1.1 Double, Half and Fold

In section 2.1 we introduced a few ways to obtain new graphs from old ones. The implementation of functions which compute the bipartite double, extended bipartite double or the fold of a graph is quite straightforward since we simply follow the definitions. However, computing the half of a graph can be slightly optimised.

Let Γ be a bipartite graph with parts X, Y. The naive approach to compute $\frac{1}{2}\Gamma$ is to compute a part X and then to iterate through it to find all pairs of vertices at distance 2. However, under the assumption that Γ is connected so is $\frac{1}{2}\Gamma$. This gives a more efficient way to compute $\frac{1}{2}\Gamma$ by exploring the whole of $\frac{1}{2}\Gamma$ by performing BFS. This is exactly what we do.

```
pick $v \in \Gamma$
queue.add(v)
while queue is not empty:
    v = queue.pop()
    candidates = [ x for c in $\Gamma_1(v)$ for x in $\Gamma_1(c)$ ]
    for w in candidates:
        if $(v,w) \notin E_\Gamma$: #then d(v,w) = 2
            if $w$ is already in $\frac{1}{2}\Gamma$: continue
            add $w$, $(v,w)$ to $\frac{1}{2}\Gamma$
            queue.add(w)
```

Note that by doing so we don't need to compute X independently via another BFS.

4.1.2 Dual Polar graphs

There are six families of graphs which fall under the name of dual polar graphs. Of these six, three were already implemented in SageMath. Here I will describe the algorithm used to implement the remaining three.

The graphs constructed will be denoted by $O(n, q)$, $O^{+}(n, q)$ and $O^{-}(n, q)$. Brouwer *et al.* (1989)[5] describe those graphs as follow:
Let V be a finite vector space of dimension n over $\mathbb{F}_q$ equipped with a nondegenerate quadratic form. The maximal totally isotropic subspaces of V constitute the vertices of the graph. They all have the same dimension k and two subspaces U, W are adjacent if $\dim(U \cap W) = k - 1$.

This construction generates three different families because there are essentially three different quadratic forms on V. In particular, if n is odd, then there is essentially only one quadratic form and we get the graph $O(n, q)$. However, if n is even, then there are two different choices for the quadratic form and so we get the graphs $O^{+}(n, q)$ and $O^{-}(n, q)$. Witt's theorem (Grove, 2002)[14] implies that a form-preserving linear map between subspaces of V can be extended to a form-preserving linear map $V \to V$. Since any linear map between two totally isotropic subspaces is trivially form-preserving, this implies that the groups $GO(n, q)$, $GO^{+}(n, q)$ and $GO^{-}(n, q)$ act transitively on the totally isotropic subspaces. It follows that we can build the graphs using the following template:

```
$G = GO^e(n, q)$
find $K$ a maximal totally isotropic subspace
vertices = orbit of $G$ on $K$
for $v, u$ in vertices:
    check if $v \sim u$
```

The groups $GO^e(n, q)$ can be built in SageMath using GAP,[12] so the hard part is finding K. To do so, we designed the following algorithm:

```
let $M$ be the matrix of the quadratic form
let $K$ be the kernel of the matrix
while $K$ is not maximal:
    find an isotropic vector $v$ not in $K$
    if Span($K \cup \{v\}$) is totally isotropic:
        add $v$ to $K$
```

The above works quite well since the kernel of M already gives us a good start and to find the isotropic vectors we can keep a set of "candidates" which we update as K expands. Moreover, to test whether $\mathrm{Span}(K \cup \{v\})$ is totally isotropic we only need to check a condition on the basis of K as shown by the below lemma.

Lemma 4.1. *Let $V = (\mathbb{F}_q)^n$ be a vector space with a quadratic form $Q(x) = x^T Mx$. Let $W = \langle v_1, \ldots, v_k \rangle$ be a subspace of V. Then W is totally isotropic if and only if $\forall i \neq j \quad v_i M v_i = 0$ and $v_i M v_j + v_j M v_i = 0$.*

Proof. ($\Rightarrow$) Since W is totally isotropic, then $v_i \in W$ for any i and so $0 = Q(v_i) = v_i M v_i$. Similarly for all $i \neq j$ we have $v_i + v_j \in W$ so

$$\begin{aligned} 0 &= Q(v_i + v_j) \\ &= (v_i + v_j)M(v_i + v_j) \\ &= v_i M v_i + v_j M v_j + v_i M v_j + v_j M v_i \\ &= v_i M v_j + v_j M v_i \end{aligned}$$

($\Leftarrow$) Let $v \in W$. We need to show $Q(v) = vMv = 0$. Note that $v = \sum_{i=1}^{k} \lambda_i v_i$. So

$$\begin{aligned} Q(v) &= (\sum_{i=1}^{k} \lambda_i v_i) M (\sum_{j=1}^{k} \lambda_j v_j) \\ &= \sum_{i=1}^{k} \sum_{j=1}^{k} \lambda_i \lambda_j (v_i M v_j) \\ &= \sum_{\substack{i,j=1 \\ i<j}}^{k} (\lambda_i \lambda_j v_i M v_j + \lambda_j \lambda_i v_j M v_i) + \sum_{i=1}^{k} \lambda_i^2 v_i M v_i \\ &= \sum_{\substack{i,j=1 \\ i<j}}^{k} \lambda_i \lambda_j (v_i M v_j + v_j M v_i) + 0 = 0 \end{aligned}$$

□

4.1.3 Doubled Odd graph

The doubled odd graph $DO(n)$ is the bipartite double of the Odd graph. However, there is a quicker construction as described by Brouwer *et al.* (1989).[5]

Let $X = \{1, \dots, 2n+1\}$. Let all subsets of size n or $n+1$ be the vertices of $DO(n)$. Two sets X, Y are adjacent if $X \subset Y$ or $Y \subset X$.

The trivial implementation that iterates trough the subsets of X turns out to be quite slow in SageMath. Hence, we represent a subset of X via a binary vector v of length $2n+1$ where $v[i] = 1$ means that i is in the set. So we iterate through the vectors of weight $n+1$ and generate their neighbours by flipping one entry of v from 1 to 0.

4.1.4 Bilinear, Alternating and Hermitian form graphs

Definition 4.2. *Let V be a vector space over K. An* alternating form *on V is a (bilinear) map $\langle v, w \rangle = v^T A w$ defined by skew-symmetric matrices A whose diagonal entries are 0.*

The bilinear form graph $Bil(d, e, q)$, the alternating form graph $Alt(n, q)$ and the Hermitian form graph $Her(n, q^2)$ all have similar constructions, so we introduce a fictitious graph called the "form graph" $Form(V, W, k)$.

The vertices of $Form(V, W, k)$ are all the appropriate forms on $V \times W$, where $f \sim g$ if $\text{rank}(f - g) = k$.

Note that there is no such graph as the "form graph", yet using the description above we have

$$\begin{aligned} Bil(d, e, q) &= Form((\mathbb{F}_q)^d, (\mathbb{F}_q)^e, 1) \text{ using bilinear forms} \\ Alt(n, q) &= Form((\mathbb{F}_q)^n, (\mathbb{F}_q)^n, 2) \text{ using alternating forms} \\ Her(n, q^2) &= Form((\mathbb{F}_{q^2})^n, (\mathbb{F}_{q^2})^n, 1) \text{ using Hermitian forms} \end{aligned}$$

Let me clarify that those three families of graphs are often defined as here (Brouwer *et al.*, 1989)[5] but without introducing any "form graph". The concrete algorithm we implemented represents a form f by its matrix A_f and precomputes the set $S = \{A_f \mid \text{rank}(A_f) = k\}$. This allows to define the neighbours of A_f by $\{A_f + B_g \mid B_g \in S\}$. Computing the rank of a matrix is relatively expensive and that's why precomputing S is a significant improvement over checking $\text{rank}(f - g)$ for any pair of vertices.

4.1.5 Halved Cube

First recall that the cube graph is just another name for the Hamming graph $H(n, 2)$. It follows that we could compute $\frac{1}{2}H(n, 2)$ by using the `halve` function described in 4.1.1. However, Godsil (2003)[13] proves that $\frac{1}{2}H(n, 2) = H(n-1, 2)_{\text{1-or-2}}$, i.e. the distance 1-or-2 graph of $H(n-1, 2)$. This is a significant improvement since $H(n-1, 2)$ has half the size of $H(n, 2)$. So we build $H(n-1, 2)$ and from this we only need to add edges between vertices at distance 2.

4.1.6 Grassmann and Doubled Grassmann graphs

Van Dam *et al.* (2016)[19] describe the Grassmann graph $J_q(n, e)$ as follow:

Let $V = (\mathbb{F}_q)^n$, then the vertices of $J_q(n, e)$ are the e-dimensional subspaces of V. Two vertices U, W are adjacent if $\dim(U \cap W) = e - 1$.

Computing the intersection $U \cap W$ boils down to finding the kernel of a matrix composed by the basis matrices of U and W. It turns out that, for the relatively small input values we use, it is quicker to compute $U \cap W$ as the intersection of sets. In particular, we use the projective space $PG_q(n-1, e-1)$ to construct all e-dimensional subspaces as sets of projective points. Then computing $U \cap W$ is relatively fast and one ensures $\dim(U \cap W) = e - 1$ by $|U \cap W| = \frac{q^e - 1}{q-1}$.

The Double Grassmann $DJ_q(e)$ is the bipartite double of the distance-e graph of $J_q(2e+1, e)$ (van Dam *et al.*, 2016).[19] However, this construction is not optimal as it involves computing the matrix of all distances of $J_q(2e+1, e)$. Brouwer *et al.* (1989)[5] give a different description of the graph:
Let $V = (\mathbb{F}_q)^{2e+1}$. The vertices of $DJ_q(e)$ are the e-dimensional and $(e+1)$-dimensional subspaces of V. Let $U \sim W$ if $U \subset W$ or $W \subset U$.
Hence we can constrcut $DJ_q(e)$ by iterating through the $(e+1)$-dimensional subspaces of V and for each such subspace W we find its neighbours by iterating through the e-dimensional subspaces of W.

4.2 Graphs with Unbounded Order

4.2.1 Generalised Polygons

Van Maldeghem (2012)[21] defines a generalised polygon as follow

Definition 4.3. *We say that an incidence structure (P, L) is a (weak)* generalised n-gon *of oder (s, t) if*

1. *each point is incident with* $t + 1$ *lines for* $t \geq 1$
2. *each line is incident with $s + 1$ points for $s \geq 1$*
3. *the diameter of the incidence graph is n*

4. the girth of the incidence graph is $2n$

Our interest stems from the fact that the point graph of a generalised n-gon is distance-regular (Brouwer *et al.*, 1989).[5] Throughout this section, I will write $GP(n,s,t)$ for the generalised n-gon of order (s,t) and $\Gamma(n,s,t)$ for its point graph.

First note that a $GP(n,1,1)$ is an ordinary polygon. Feit and Higman (1964)[11] prove that, if we exclude ordinary polygons, then finite examples of generalised n-gons exist only for $n \in \{2,3,4,6,8,12\}$. In addition, note that the point graph of a generalised n-gon has diameter $\lfloor \frac{n}{2} \rfloor$ since its incidence graph has diameter n. So we are only interested in cases where $n \geq 6$. In particular, all known generalised n-gons for $n \in \{6,8,12\}$ are listed in the table below where q is a prime power.

n	order (s,t)
6	$(1,q)$, $(q,1)$, (q,q), (q,q^3), (q^3,q)
8	$(1,q)$, $(q,1)$, $(2^{2k+1},2^{4k+2})$, $(2^{4k+2},2^{2k+1})$
12	$(1,q)$, $(q,1)$

To further simplify our constructions we have that:

1. $\Gamma(2n,1,t)$ is the incidence graph of $GP(n,t,t)$;
2. The dual of a $GP(n,s,t)$ is a $GP(n,t,s)$;
3. Given $\Gamma(n,s,t)$ we can deduce $GP(n,s,t)$.

For 3 we have that the set of lines of $GP(n,s,t)$ is the set of singular lines of $\Gamma(n,s,t)$ (Brouwer *et al.*, 1989).[5] The set of singular lines is $L = \{\{v,w\}^{\perp\perp} \mid v \sim w\}$ where $S^{\perp} = \bigcap_{v \in S}(\{u \mid u \sim v\} \cup \{v\})$. So if we can compute $\Gamma(n,s,t)$, then we can use the above formula to compute the line graph of $GP(n,s,t)$ which is $\Gamma(n,t,s)$.

Note that $\Gamma(4,q,q)$ is a strongly-regular graph and $GP(3,q,q)$ is $PG(2,q)$, so the construction of $\Gamma(8,1,q)$ and $\Gamma(6,1,q)$ is straightforward. Therefore, the only "complicated" constructions are for $\Gamma(6,q,q)$, $\Gamma(6,q,q^3)$ and $\Gamma(8,2^{2k+1},2^{4k+2})$. All the above graphs are called graphs of Lie type since they are associated to the Lie groups $G_2(q)$, ${}^3D_4(q)$ and ${}^2F_4(q)$ (Brouwer *et al.*, 1989).[5] In particular, all the above groups act transitively on the edges of their respective graphs. The GAP package AtlasRep[25] contains constructions for these groups for small enough values of q and bigger values would lead to unfeasibly large graphs. So we build $\Gamma(6, q, q)$, $\Gamma(6, q, q^3)$ and $\Gamma(8, 2^{2k+1}, 2^{4k+2})$, by constructing the related group and then computing the orbit of the group on an edge that we previously found by hand. To help us find an edge we note that if $u \sim v$ and k is the valency of the graph, then v must be in a k-orbit of the stabiliser of u under the related group.

4.2.2 Generalised Quadrangles

There are quite a few different definitions for generalised quadrangles, but here we simply say that it is a generalised n-gon with $n = 4$. However, we will later need the following property from Payne and Thas (1984).[15]

Proposition 4.4. *Let $GQ = (P, \mathcal{L})$ be a generalised quadrangle, then given a line $l \in \mathcal{L}$ and a point $p \in P$ outside l, there is a unique line $l' \in \mathcal{L}$ incident with l which contains p.*

If the above were not true, then the generalised quadrangle would contain a triangle, which translate to a cycle of length 6 in the incidence graph. This can't happen since the incidence graph must have girth 8.

Definition 4.5. *Given a generalised quadrangle $GQ = (P, \mathcal{L})$, an* ovoid *of GQ is a set of points $O \subseteq P$ such that any line of GQ intersects O in exactly one point. A* spread *of GQ is a set of lines $S \subseteq \mathcal{L}$ such that S partitions P, i.e. any point is in exactly one line of S.*

One should note that the dual of a generalised quadrangle GQ of order (s, t) is a generalised quadrangle GQ' of order (t, s) and that an ovoid in GQ maps to a spread in GQ'.

Brouwer *et al.* (1989)[5] describe the following way to obtain distance-regular graphs from generalised quadrangles with a spread:
Given a generalised quadrangle $GQ = (P, \mathcal{L})$ of order (s, t) with $t > 1$ and a spread S. The point graph of the incidence structure $(P, \mathcal{L} \setminus S)$ is a distance-regular graph.

Computing the point graph of an incidence structure is straightforward so the issues lie in constructing generalised quadrangles with spreads (or ovoids) since SageMath doesn't have any such constructions. To limit the scope of this project, we implemented only one infinite family. In particular, Payne and Thas (1984)[15] describe a construction of the generalised quadrangle known as $H(d, q^2)$, where $d = 3, 4$, as follow:
Construct the projective space $PG(d, q^2)$ and let $H = \{\langle v\rangle \mid \langle v, v\rangle = 0\}$, where $\langle -, -\rangle$ is a nondegenerate Hermitian form. Then the points and lines of H form a generalised quadrangle of order (q^2, q) for $d = 3$ and (q^2, q^3) for $d = 4$.
Moreover, Thas and Payne (1994)[17] proved that $H(3, q^2)$ always has ovoids using the following theorem:

Theorem 4.6. *If the generalised quadrangle S of order (s, t) has a subquadrangle S' of order (s, t'), then any point $z \in S \setminus S'$ is collinear to all $1 + st'$ points of an ovoid O_z of S'*

In particular, we can embed $H(3, q^2)$ in $H(4, q^2)$, then by picking z in the latter but not in the former we can find an ovoid. The following lemma will turn out to be useful.

Lemma 4.7. *Using the notation of theorem 4.6 we have that z is collinear to $p \in S'$ if and only if p is in O_z.*

Proof. Therorem 4.6 already proves that all points in O_z are collinear to z, so we only need to prove the converse.
Assume for a contradiction that z is collinear to $p \in S'$, but $p \notin O_z$. Then p must lie on at least one line, say l. By definition of an ovoid, l intersects O_z in a unique point q. Then we have that z is collinear to q. Hence, z is collinear to two distinct points that lie on the same line. This contradicts 4.4. □

Constructing $H(3, q^2)$ is not complicated as we can use the same idea of section 4.1.2. In particular, recall the generalised unitary group $GU(d, q)$ from section 2.3. Then using Witt's theorem (Grove, 2002)[14] we have that $GU(d, q)$ acts transitively on all totally isotropic subspaces. Moreover, such group can be constructed in SageMath through the GAP package and the invariant Hermitian form preserved by that construction will have

$n \times n$ matrix $J_n = (\delta_{i+j,n+1})$. To help visualise J_n note that for $n = 4$ we have

$$J_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

In the GAP package, vectors are row vectors, so the Hermitian form is defined by $\langle x, y \rangle = xJ\overline{y}^T$ where $\overline{x} = x^q$. For the reminder of this section we adopt the same conventions of the GAP package to reduce the differences between the code and my explanation.

To construct $H(3, q^2)$ is enough to find an isotropic vector and a totally isotropic projective line since the action of $GU(4, q)$ will generate all other projective points and lines. Since we know J, the task is relatively easy. In particular, let e_i be the standard basis, then e_1 is isotropic and $\text{Span}(\{e_1, e_2\})$ is totally isotropic.

Next, we need an embedding of $H(3, q^2)$ in $H(4, q^2)$. That is, we need an injective map $\phi : PG(3, q^2) \to PG(4, q^2)$ such that the image of $H(3, q^2)$ is a subquadrangle of $H(4, q^2)$. A map that works is given by $\phi((a, b, c, d)) = (a, b, 0, c, d)$. In particular, ϕ is a valid embedding since $\langle v, v \rangle_{PG(3,q^2)} = 0 \iff \langle \phi(v), \phi(v) \rangle_{PG(4,q^2)} = 0$. Therefore, the points of $H(3, q^2)$ will map to points of $H(4, q^2)$ and the lines of $H(3, q^2)$ will be contained in the lines of $H(4, q^2)$.

Next we need $z \in H(4, q^2) \setminus H(3, q^2)$. So we have the following lemmas:

Lemma 4.8. *Assume $q = p^k$ where p is an odd prime and $k > 0$, and let $a = \frac{p-1}{2}$. Then the vector $v = (0, 1, 1, a, 0)$ is a point of $H(4, q^2)$.*

Proof. v is a point of $H(4, q^2)$ if and only if it is isotropic with respect to the Hermitian form given by J_5, which we shall write simply as J. So we can compute $vJ = (0, a, 1, 1, 0)$. Hence $vJ\overline{v}^T = (0, a, 1, 1, 0)(0, 1, 1, a^q, 0)^T$. Now note that a is an integer and so $a \in \mathbb{F}_q$. Thus $a^q = a$ by Fermat Little Theorem. It follows that $vJ\overline{v}^T = 0 + a + 1 + a + 0 = 2a + 1 = (p - 1) + 1 = p = 0$ since we are in a field with characteristic p. Thus v is isotropic. $\square$

Lemma 4.9. *Assume $q = 2^k$ where $k > 0$ and let g be a primitive root of $\mathbb{F}_{q^2}$. If $a = g^{q-1}$, then the vector $v = (0, 1, a + 1, a, 0)$ is a point of $H(4, q^2)$.*

Proof. As in the previous proof we only need to check that v is isotropic. I will write J for J_5. First note that in characteristic 2 we have $(x + y)^2 = x^2 + y^2 + 2xy = x^2 + y^2$. So inductively, $(x + y)^{2^k} = (x^2 + y^2)^{2^{k-1}} = (x^4 + y^4)^{2^{k-2}} = \cdots = x^{2^k} + y^{2^k}$. Hence, $(a + 1)^q = a^q + 1$. Then

$$\begin{aligned} vJ\overline{v}^T &= 0 + a^q + (a + 1)^{q+1} + a + 0 \\ &= a^q + a + (a + 1)^q(a + 1) \\ &= a^q + a + (a^q + 1)(a + 1) \\ &= a^q + a + a^{q+1} + a + a^q + 1 \\ &= a^{q+1} + 1 \quad \text{since we are in characteristic 2} \end{aligned}$$

Recall, $a = g^{q-1}$ so $a^{q+1} = g^{(q-1)(q+1)} = g^{q^2-1} = 1$. Thus $vJ\overline{v}^T = 1 + 1 = 0$. □

The above proof will work for any a such that $a^{q+1} = 1$. So something as simple as $a = 1$ will do. However, for the purpose of the algorithm we need $a + 1 \neq 0$, otherwise v would be in $H(3, q^2)$. Since GAP (and SageMath) represents the elements of $\mathbb{F}_q$ using a primitive root, then g is already computed, so $a = g^{q-1}$ is quite easy to calculate.

Now we can finally compute the ovoid. The naive approach of checking collinearity by looking at the lines of $H(4, q^2)$ is not the most efficient way as $H(4, q^2)$ has $(1+q^3)(1+q^5)$ lines (Payne and Thas, 1984).[15] Hence, I opted to check whether the two vectors span a totally isotropic line, since those are the lines of $H(4, q^2)$. In particular, the following lemma gives us a quick way of doing so.

Lemma 4.10. *Let $\langle x, y\rangle = xM\overline{y}^T$ be an Hermitian form on $(\mathbb{F}_{q^2})^d$. Then the span of $\{v, w\}$ is a totally isotropic subspace if and only if $vM\overline{v}^T = wM\overline{w}^T = vM\overline{w}^T = wM\overline{v}^T = 0$*

Proof. $\Leftarrow$ Let u be in the span of $\{v, w\}$. Then $u = \lambda v + \mu w$ for some scalars λ, μ. Then $\langle u, u\rangle = \langle \lambda v + \mu w, \lambda v + \mu w\rangle = \lambda\overline{\lambda}\langle v, v\rangle + \mu\overline{\mu}\langle w, w\rangle + \lambda\overline{\mu}\langle v, w\rangle + \mu\overline{\lambda}\langle w, v\rangle$. Note that

$\langle v,v\rangle = vM\overline{v}^T = 0$, $\langle w,w\rangle = wM\overline{w}^T = 0$, $\langle v,w\rangle = vM\overline{w}^T = 0$ and $\langle w,v\rangle = wM\overline{v}^T = 0$. So $\langle u,u\rangle = 0$.

$\Rightarrow$ First note that we must have $\langle v,v\rangle = \langle w,w\rangle = 0$ since v,w are in a totally isotropic subspace. Hence $vM\overline{v}^T = wM\overline{w}^T = 0$. Now let $u = v+\lambda w$ for some scalar λ. Then regardless of λ we must have $\langle u,u\rangle = 0$. Hence

$$\begin{aligned}
0 &= \langle v+\lambda w, v+\lambda w\rangle \\
&= \langle v,v\rangle + \lambda\overline{\lambda}\langle w,w\rangle + \lambda\langle w,v\rangle + \overline{\lambda}\langle v,w\rangle \\
&= \lambda\langle w,v\rangle + \overline{\lambda}\langle v,w\rangle && \text{since } \langle v,v\rangle = \langle w,w\rangle = 0 \\
&= \lambda\overline{\langle v,w\rangle} + \overline{\lambda}\langle v,w\rangle && \text{since } \langle -,-\rangle \text{ is Hermitian}
\end{aligned}$$

Let $\langle v,w\rangle = a$, then we get

$$0 = \lambda\overline{a} + \overline{\lambda}a \quad \text{for any } \lambda \tag{4.2.1}$$

Now we need to distinguish between the cases where q is even or odd.

First assume q is odd. Let $\lambda = a$ in 4.2.1. Then we get $0 = a\overline{a} + \overline{a}a = 2a\overline{a} = 2a^{q+1}$. Since $2 \neq 0$, we have $a^{q+1} = 0$ and so $a = 0$. Thus $\langle v,w\rangle = 0 = \langle w,v\rangle$. Hence $vM\overline{w}^T = wM\overline{v}^T = 0$.

Assume now that q is even. Assume further for a contradiction that $a \neq 0$. Let g be a primitive root of $\mathbb{F}_{q^2}$. Then $a = g^k$ for some k. Let $\lambda = 1$ in 4.2.1, then we get $0 = \overline{a} + a = a^q + a$. So $a^{q-1} + 1 = 0$ since $a \neq 0$. Thus $g^{k(q-1)} = 1 = g^0$. Hence:

$$k(q-1) \equiv 0 \mod (q^2-1) \tag{4.2.2}$$

since the order of $(\mathbb{F}_{q^2})^\times$ is q^2-1. Now let $\lambda = g$ in 4.2.1. Then we get $0 = ga^q + g^q a = g^{kq+1} + g^{q+k}$. Hence $g^{kq+1} = g^{q+k}$ and so $kq+1 \equiv q+k \mod (q^2-1)$, from which we deduce

$$k(q-1) \equiv q-1 \mod (q^2-1) \tag{4.2.3}$$

Now note that equations 4.2.2 and 4.2.3 imply $0 \equiv q-1 \mod (q^2-1)$ which is a contradiction for $q > 1$. Hence we must have that $a = 0$. $\square$

Using the above lemma and 4.7 we can find an ovoid by iterating through the points w of $H(3, q^2)$ and computing $\phi(w)J\bar{z}^T$, where z is our vector from 4.8 or 4.9. Moreover, recall that z has the form $(0, 1, \alpha, \beta, 0)$ and $\phi(w) = (a, b, 0, c, d)$. Hence $\phi(w)J\bar{v}^T = b\beta^q + c$. So by precomputing β^q, the check becomes very simple.

To sum up, we generate $H(3, q^2)$ using $GU(4, q)$. Then by 4.8 and 4.9 we have a point $z \in H(4, q^2) \setminus H(3, q^2)$. Finally, by 4.10 we can compute the ovoid. One we have the generalised quadrangle, we build its dual to obtain one with spread.

4.2.3 Unitary Nonisotropic graph

Brouwer *et al.* (1989)[5] construct the unitary nonisotropic graph as follow:
For $q > 2$, let $V = (\mathbb{F}_{q^2})^3$ equipped with the standard Hermitian form. Let the set of nonisotropic projective points be the vertices and say that v, w are adajcent if $\langle v, w \rangle = 0$.

In addition, they prove that the group $GU(3, q)$ acts transitively on the edges of such graph. Hence, we can build the graph with minimal effort. In particular, we use GAP to create the group $GU(3, q)$. Recall from section 4.2.2 that the Hermitian form preserved by that group is given by the matrix

$$J = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Then the vector $v = (1, 1, 0)$ is nonisotropic since $\langle v, v \rangle = 1$. So by considering the action of $GU(3, q)$ on the projective point v we get the set of vertices, i.e. the set of nonisotropic projective points. Then note that $w = (0, 1, -1)$ is also nonisotropic and $\langle w, v \rangle = 0$. So $v \sim w$. Thus by considering the action of $GU(3, q)$ on the edge (v, w) we can generate all the other edges.

4.2.4 Hermitian cover

Cameron (1991)[7] describes a way to obtain antipodal covers of complete graphs using several forms. However, only when using the 3-dimensional Hermitian form he obtains graphs with new parameters. Hence, we describe only such construction:
Let V be a 3-dimensional vector space over $\mathbb{F}_{q^2}$ with a nondegenerate Hermitian form. Let H be a subgroup of index r in the group $(\mathbb{F}_{q^2})^{\times}$. The set of vertices is $\{Hv \mid v \text{ is isotropic }\}$ and $Hv \sim Hw$ if $\langle v, w\rangle \in H$.

The above construction doesn't always yield a distance-regular graph, but Brouwer in the erratum for the book 'Distance-Regular Graphs' (1989)[4] states all the sufficient and necessary conditions on r and q for the graph to be distance-regular. In particular, one needs one of the following:

1. r is odd and $r \mid (q-1)$
2. q is odd and $r \mid \frac{1}{2}(q+1)$
3. q is even and $r \mid (q+1)$

However, the construction described by Brouwer is flawed as he forgot to mention that the vectors need to be isotropic. This oversight has been brought to his attention and will soon be corrected.

To obtain a quick construction of the graph described above, we, once again, resort to utilise the generalised unitary group $GU(3, q)$. In particular Brouwer[4] shows that $GU(3, q)$ acts transitively on the vertices and edges of the graph. Recall once more that the Hermitian form fixed by $GU(3, q)$ has matrix

$$J = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Hence, the standard vectors e_1 and e_3 are isotropic and $He_1 \sim He_3$. So by looking at the

action of $GU(3,q)$ on the edge (He_1, He_3) we can get all the edges of the graphs. However, this time the action of the group is not the usual action on projective points, so we need to define it ourselves. In particular, we need a way of "normalising" Hv to a vector w. We do so as follow:
Let g be a generator of $(\mathbb{F}_{q^2})^\times$. Then $H = \langle g^r \rangle$. So the quotient group is $\{H, gH, g^2H, \dots, g^{r-1}H\}$. Hence, we can "normalise" Hv to a vector w whose first non-zero entry is in $\{1, g, g^2, \dots, g^{r-1}\}$. So $M \in GO(3,q)$ acts on a normalised v by sending it to the normalisation of vM (we have a right action since v is a row vector in GAP).

Once we have defined the above action, the graph is constructed in the usual way by looking at the orbit of the edge (e_1, e_3).

4.2.5 Pasechnik and Brouwer-Pasechnik

Brouwer and Pasechnik (2011)[6] describe the Brouwer-Pasechnik graph as follow:
Let V be a 3 dimensional vector space over $\mathbb{F}_q$. Let $V \times V$ be the set of vertices. Two distinct vertices (v_1, v_2), (w_1, w_2) are adjacent if and only if $v_1 \times w_1 + v_2 - w_2 = 0$, where $\times$ denotes the cross product.

The construction is quite straightforward, but annoyingly $(v_1, v_2) \sim (v_1, v_2)$ using the above description. Brouwer and Pasechnik solve this issue by explicitly requiring $(v_1, v_2) \neq (w_1, w_2)$. However, we can slightly loosen this condition and have a neater construction.

Lemma 4.11. *Let $v_1 \times w_1 + v_2 - w_2 = 0$. Then $(v_1, v_2) = (w_1, w_2)$ if and only if $v_1 = w_1$*

Proof. The only interesting direction is $v_1 = w_1 \implies (v_1, v_2) = (w_1, w_2)$ and this can be proved easily. We have $w_2 = v_1 \times w_1 + v_2 = v_1 \times v_1 + v_2$. Now note that $v_1 \times v_1 = 0$ and so we get $w_2 = v_2$. $\square$

In the same paper, Brouwer and Pasechnik describe another family of graphs known as the Pasechnik graphs. They also prove that the Pasechnik graphs are the extended bipartite double graphs of the Brouwer-Pasechnik graphs. Thus we construct the Pasechnik graphs by computing the extended bipartite double of the graphs constructed above.

4.2.6 Transversal Designs

Transversal designs are peculiar incidence structures with many different characterisations and here we will follow the approach of Beth *et al.* (1999),[3] but simplify the definition for our purpose.

Definition 4.12. *An incidence structure* $(P, \mathcal{B})$ *is* divisible *if* P *can be partitioned into* point classes *such that any two points in the same class are incident with* λ_1 *blocks. A divisible incidence structure is called a* divisible design *if any two points in different classes are incident with* λ_2 *blocks.*
We write $\mathrm{GD}_{\lambda_1,\lambda_2}[k, g, v]$ *for divisible design where each block has size* k*, each point class has size* g *and there are* v *points.*

Definition 4.13. *A divisible design* $\mathrm{GD}_{0,\lambda}[k, g, v]$ *is called a* tansversal design $\mathrm{TD}_\lambda[k, g]$ *if every block intersects every point class.*
The dual incidence structure of a $\mathrm{TD}_\lambda[k, g]$ *is called a* $(g, k; \lambda)$*-net.*

Note that a $\mathrm{TD}_\lambda[k, g]$ must have kg points since the number of point classes must be k.

Definition 4.14. *A* $\mathrm{TD}_\lambda[k; g] = (P, \mathcal{B})$ *is called* symmetric *if its dual is still a* $\mathrm{TD}_\lambda[k; g]$ *and is called* resolvable *if there is a partition of* $\mathcal{B}$ *such that each part is a partition of* P.

Note that symmetry implies $k = g\lambda$.

Brouwer *et al.* (1989)[5] prove that the incidence graph of a symmetric transversal design is distance-regular. Hence, we need to construct some transversal designs. SageMath can already build quite a lot of $\mathrm{TD}_1[k, g]$, so we are interested in ways to build transversal designs with $\lambda > 1$.

To do so we introduce some related objects (Beth *et al.*, 1999):[3]

Definition 4.15. *An* orthogonal array $OA_\lambda(k, s)$ *is a* $(\lambda s^2 \times k)$ *matrix with entries from*

$S = \{0, \dots, s-1\}$ *such that any 2 columns contain all ordered pairs from* S *exactly* λ *times.*

Definition 4.16. *A* $(g, k; \lambda)$-difference matrix M *over a group* $(G, +)$ *of size* g *is a* $(k \times \lambda g)$ *matrix with entries from* G *such that for any two rows* $i \neq j$ *each element of* G *appears* λ *times in the vector* $M[i] - M[j]$.

Beth *et al.* (1999)[3] prove the following theorems:

Theorem 4.17. *A transversal design* $\mathrm{TD}_\lambda[\lambda g, g]$ *is symmetric if and only if it is resolvable.*

Theorem 4.18. *The existence of a* $OA_\lambda(k, s)$ *is equivalent to the existence of a* $\mathrm{TD}_\lambda[k, s]$.

In light of this, one says that a $OA_\lambda(k, s)$ is resolvable if its related transversal design is.

Theorem 4.19. *The existence of a* $(s, k; \lambda)$ *difference matrix implies the existence of a resolvable* $OA_\lambda(k, s)$.

SageMath has generators for both orthogonal arrays and difference matrices together with the maps: difference matrix $\rightarrow$ orthogonal array $\rightarrow$ transversal design. So we extended the generator of the orthogonal arrays to allow $\lambda > 1$ and constructed difference matrices with $k = \lambda g$, which will imply symmetric $\mathrm{TD}_\lambda[k, g]$. In particular, we implemented the following constructions:

1. difference matrices (p^i, p^{i+j}, p^j) where p is prime;
2. difference matrices $(q, 2q, 2)$ where q is a prime power;
3. difference matrices $(n/s, k, \lambda s)$ where we have a difference matrix (n, k, λ) over G and H is a normal subgroup of G of order s.

The construction 2 is uninteresting since Beth *et al.* (1999)[3] essentially give a formula for the entries of the matrix and the implementation is trivial. The other two constructions can be found in Drake (1979).[10] In particular, construction 3 is given by the following proposition:

Proposition 4.20. *Let $H = (h_{ij})$ be a difference matrix over G. Let $\phi : G \to G'$ be a surjective homomorphism, then $H' = (\phi(h_{ij}))$ is a difference matrix over G'.*

The homomorphism used in the third construction is $\phi : G \to G/H$ given by $g \mapsto gH$.

Construction 1 can be described as follow (Drake, 1979):[10]
Let K be the multiplication table of $\mathbb{F}_{p^{i+j}}$. Then K is a $(p^{i+j}, p^{i+j}, 1)$ difference matrix over $(\mathbb{F}_{p^{i+j}}, +)$. Then using 4.20 with $\phi : \mathbb{F}_{p^{i+j}} \to \mathbb{F}_{p^i}$ we obtain the desired difference matrix.

Finding such ϕ can be quite complicated, so what we do in the code is $\mathbb{F}_{p^{i+j}} \xrightarrow{\sim} (\mathbb{Z}_p)^{i+j} \xrightarrow{\phi} (\mathbb{Z}_p)^i$ where ϕ is just truncating the vectors to length i.

4.2.7 BIBD graphs

Brouwer *et al.* (1989)[5] define BIBDs as follow:

Definition 4.21. *A* Balanced Incomplete Block Design*, also called a* 2-design*, is an incidence structure $(P, \mathcal{B})$ where each block has size k and any two distinct points are exactly in λ different blocks.*

We often associate to a BIBID D the parameters (v, b, k, r, λ) where k, λ are as above, v is the number of points, b is the number of blocks and any point is in r different blocks. However, when describing a BIBD only the parameters (v, k, λ) are given since the other two can be deduced from the following equalities:

$$bk = vr$$
$$\lambda(v-1) = r(k-1)$$

Brower *et al.* (1989)[5] also prove that the incidence graph of a non-trivial *symmetric* BIBD is distance-regular. Here, *symmetric* (or *square*) simply means that the number of points is the same as the number of blocks.

SageMath already has a generator of BIBDs, but it only handles incidence structures with $\lambda = 1$. Hence, we modified the generator to allow for arbitrary values of λ. After some work, SageMath was able to generate several new BIBDs with $\lambda \neq 1$ without the need of any new construction. This is mainly due to the connections between BIBDs and other combinatorial objects that SageMath can build. Hence, little work was required to set up those constructions.

4.2.8 Taylor graphs

Brouwer *et al.* (1989)[5] define Taylor graphs to be any distance-regular graph with intersection array $[k, \mu, 1, 1, \mu, k]$. These graphs are related to some particular incidence structures called two-graphs.

Definition 4.22. *A* two-graph *is an incidence structure* $(P, \mathcal{B})$ *where each block has size 3 and any set of 4 points contains an even number of blocks. Moreover, we say that the two-graph is* regular *if every pair of points is contained in a constant number* λ *of blocks.*

Note that a regular two-graph is a BIBD.

Brouwer *et al.* (1989)[5] prove that there is a bijective correspondence between antipodal 2-covers of the complete graph and non-complete two-graphs. Moreover, we have that the antipodal cover is distance-regular, i.e. a Taylor graph, if and only if the two-graph is regular.

Brouwer *et al.* (1989)[5] also describe the bijection, which result in the following construction:
Let $D = (P, \mathcal{B})$ be a two-graph. Fix $\alpha \in P$. Let the vertex set be $\{(x,0),(x,1) \mid x \in P\}$ and define adjacency by

1. $(x,k) \sim (y,k)$ if $x \neq y$ and $\{x, y, \alpha\}$ is coherent

2. $(x, 0) \sim (y, 1)$ if $x \neq y$ and $\{x, y, \alpha\}$ is not coheren

A triple is coherent if it is a block, while a pair is always coherent. It follows that $(\alpha, 0) \sim (x, 0)$ and $(\alpha, 1) \sim (x, 1)$ for any $x \in P \setminus \{\alpha\}$.

To practically construct the Taylor graph given a two-graph $(P, \mathcal{B})$ we pick α and add all the edges with α in them. Then we iterate through $\mathcal{B}$ and keep track of the pairs (x, y) such that $\{x, y, \alpha\}$ is a coherent, i.e. a block. Once terminated, we iterate through all paris (x, y) and add edges according to the adjacency relation.

Now the focus shifts on constructing two-graphs. Luckily, SageMath can already construct an infinite family of regular two-graphs. Hence, we designed a generator function for two-graphs. The complement of a two-graph is the incidence structure with the same points, but has as blocks all the triples that are not blocks in the original two-graph (Brouwer *et al.*, 1989).[5] Hence, by including the complement of the already-available family, we obtain enough examples for our purpose.

4.2.9 Denniston graphs

Denniston (1969)[9] defines a maximal arc in $PG(2, q)$ as follow:

Definition 4.23. *A* n-maximal arc *in the projective plane* $PG(2, q)$ *is a set of points* S *such that any line intersects* S *in 0 or* n *points.*

In the same paper, Denniston also gives a construction of such an arc when n and q are powers of 2. In particular, the construction given is:
Let $Q(x, y)$ be an irreducible quadratic form over $(\mathbb{F}_q)^2$. Let H be a subgroup of the additive group of $\mathbb{F}_q$ of order n. Then the arc is given by the set $A = \{(1, x, y,) \mid Q(x, y) \in H\}$.

Here, a quadratic form is intended as we introduced in section 2.3, but we write $Q(x, y)$ for $Q(v)$ where v is the vector (x, y). Hence, $Q(x, y)$ becomes a homogeneous polynomial of degree 2, i.e. $Q(x, y) = \alpha x^2 + \beta xy + \gamma y^2$. Thus irreducible means that $Q(x, y)$ is irreducible as a polynomial over x, y.

Thas (1974)[16] describes a way of taking the complement of arcs in the projective plane to obtain peculiar incidence structures. In particular, let $PG(2, q) = (P, \mathcal{L})$ and let A be an n-arc in it. Then Thas studied the incidence structure $D = (P \setminus A, \mathcal{L}')$ where $\mathcal{L}' = \{L \mid L \in \mathcal{L} \wedge L \cap A = n\}$.

Brouwer *et al.* (1989)[5] prove that the incidence graph of the complement (as intended by Thas) of a Denniston's arc when $q = n^2$ is distance-regular.

Once we have the arc A, computing the complement is quite easy and so is constructing the incidence graph. Hence, here we will focus on finding the maximal arc. The first step is to find an irreducible quadratic form. To achieve this, we have the following lemmas.

Lemma 4.24. *Let $Q(x, y) = x^2 + axy + y^2$ be a quadratic form over $(\mathbb{F}_q)^2$. Then $Q(x, y)$ is reducible if and only if there is $b \in \mathbb{F}_q$ s.t. $a = b + \frac{1}{b}$.*

Proof. $\Rightarrow$ Assume $Q(x, y)$ is reducible. Then $Q(x, y) = (\alpha x + \beta y)(\gamma x + \delta y)$ for some $\alpha, \beta, \gamma, \delta \in \mathbb{F}_q$. Then $Q(x, y) = \alpha\gamma(x + \frac{\beta}{\alpha}y)(x + \frac{\delta}{\gamma}y)$ where we need $\alpha \neq 0 \neq \gamma$ since $Q(x, y)$ has a term x^2. Morever, we can deduce that $\alpha\gamma = 1$. So by relabelling we get $Q(x, y) = (x + \alpha y)(x + \beta y)$ for some $\alpha, \beta \in \mathbb{F}_q$. Unfolding the product we obtain $Q(x, y) = x^2 + (\alpha + \beta)xy + \alpha\beta y^2$. So we can deduce $\alpha\beta = 1$. Thus $\alpha = \frac{1}{\beta}$ since $\beta \neq 0$. Hence we get $Q(x, y) = x^2 + (\beta + \frac{1}{\beta})xy + y^2$. Thus, by comparing terms we obtain that $a = \beta + \frac{1}{\beta}$.
$\Leftarrow$ Assume $a = b + \frac{1}{b}$. Then $(x + by)(x + \frac{1}{b}y) = x^2 + axy + y^2 = Q(x, y)$ and so $Q(x, y)$ is reducible. $\square$

Lemma 4.25. *For any prime power q, there is $a \in \mathbb{F}_q$ such that there is no $b \in \mathbb{F}_q$ satisfying $a = b + \frac{1}{b}$.*

Proof. Consider the function $f : \mathbb{F}_q \to \mathbb{F}_q$ given by $f(x) = x + \frac{1}{x}$. Then $f(0)$ is not defined. So the domain of f has actually the size of $q - 1$. Thus the range of f can't have size bigger than $q - 1$. Therefore there is at least one element of $\mathbb{F}_q$ which is not in the range of f, such element is a suitable a. $\square$

In addition, one can note that $b+\frac{1}{b}=(\frac{1}{b})+\frac{1}{\frac{1}{b}}$. So the range of f is no bigger than $1+\frac{q-1}{2}$. Therefore, to find an irreducible quadratic form we compute the range of f and pick an element outside of it.

Now that we have $Q(x,y)$ computing the arc is quite straightforward, we just iterate through the pairs (x,y) and if $Q(x,y)\in\mathbb{F}_n$, then we add the point $(1,x,y)$ to the arc.

4.2.10 Association Schemes

Brouwer *et al.* (1989)[5] define association schemes in the following manner:

Definition 4.26. *A d-class* association scheme *is a pair $(X,\mathcal{R})$ where X is a set and $\mathcal{R}=\{R_0,...,R_d\}$ is a partition of $X\times X$ with the following properties:*

1. $R_0=\{(x,x)\mid x\in X\}$
2. *if $(x,y)\in R_i$, then $(y,x)\in R_i$*
3. *for any pair $(x,y)\in R_k$ there are p_{ij}^k $z\in X$ s.t. $(x,z)\in R_i$ and $(z,y)\in R_j$.*

The numbers p_{ij}^k are called the intersection numbers of the scheme.

Definition 4.27. *A d-class association scheme is called* pseudocylic *if there is a constant n such that $p_{ii}^0=n$ for $1\le i\le d$ and $\sum_{i=0}^d p_{ii}^k=n-1$ for $1\le k\le d$.*

Definition 4.28. *Let Q be a set. We say that $(Q,\oplus)$ is a* quasigroup *if $\oplus: Q\times Q\to Q$ is a binary operation on Q and, given any two of $\{a,b,a\oplus b\}$, the third can be uniquely determined.*

The reason for all those definitions is that one can construct graphs from association schemes with quasigroups. In particular, Brouwer *et al.* (1989)[5] propose the following construction:

Given a d-class association scheme $(X,\mathcal{R})$ and a commutative quasigroup $I=\{1,...,d\}$, let $(X \cup \{\infty\}) \times I$ be the vertex set, where ∞ is a symbol not in X. We say that two distinct vertices (a, i), (b, j) are adjacent if either $(a, b) \in R_{i \oplus j}$, or $a = \infty$ and $i = j$.

Brouwer *et al.* (1989)[5] also prove that if the d-class association scheme is pseudocyclic with intersection numbers satisfying $p^{k\oplus l}_{i\oplus l, j\oplus l} = p^k_{ij}$ for any $i, j, k, l \in \{1, \dots, d\}$, then the graph derived from it is distance-regular.

The construction described above is easily implemented, so we only need to worry on how to obtain association schemes. Unfortunately, SageMath is lacking in any support for association schemes, so we had implemented a simple class. We keep the set X as it could be a set of any type of objects, but $\mathcal{R}$ can be represented as a matrix M. More precisely, let $X = \{x_0, ..., x_n\}$, then $M[i, j] = k$ means that $(x_i, x_j) \in R_k$. Using our definition we could store only the upper diagonal entries of M, yet there are more general definitions of association schemes which would result in M not being symmetric.

Constructing a database of association schemes could easily be a stand-alone project, therefore we decided to construct only one infinite family called *cyclotomic schemes*. Following Brouwer *et al.* (1989)[5] one can construct a cyclotomic scheme as follow:
Let $X = \mathbb{F}_q$. Let K be a subgroup of $(\mathbb{F}_q)^\times$ and let g be a primitive root of $\mathbb{F}_q$. Let $r = \frac{q-1}{|K|}$, then for $1 \leq i \leq r$ define $R_i = \{(x, y) \mid x - y \in g^i K\}$.
When $|K|$ is even or q is a power of 2, the association scheme described is symmetric and pseudocylic. Let $I = \{1, ..., r\}$ and define $i \oplus j = i + j \mod r$ where 0 is interpreted as r. Then I is a commutative group (so a quasigroup) and together with the above schemes it give rise to distance-regular graphs.

4.2.11 Preparata graph

De Caen *et al.* (1995)[8] construct a graph related to the Preparata codes as follow:
Let A be a subgroup of $(\mathbb{F}_{2^{2t-1}}, +)$. The vertices are $\mathbb{F}_{2^{2t-1}} \times \mathbb{F}_2 \times (\mathbb{F}_{2^{2t-1}}/A)$. Two vertices (x_1, i, y_1), (x_2, j, y_2) are adjacent if and only if $(y_1 + y_2) + A = {x_1}^2 x_2 + x_1 {x_2}^2 + (i + j)({x_1}^3 + {x_2}^3) + A$.

It follows that we need an efficient way to handle operations modulo A as well as finding a suitable A given its size, since this is what is going to affect the intersection array. Once we can achieve this, the construction becomes routine. So here we will explain our approach to this issue. First note that $\mathbb{F}_{2^{2t-1}} \cong (\mathbb{F}_2)^{2t-1}$ as vector spaces (and so additive groups) and SageMath can already compute this isomorphism quickly. Hence, we can pick A to be the subspace generated by $\{e_1, ..., e_k\}$ where $|A| = 2^k$. To performs computations in the quotient group we keep track of the maps $f : x \mapsto x + A$ and $g : x + A \mapsto x$, where the latter sends the set $x + A$ to a unique representative x such that $f \circ g$ is the identity. Finding f and g is quite straightforward:

```
for each $x \in \mathbb{F}_{2^{2t-1}}$:
    compute $x + A$
    set $f(x) = x + A$
    set $g(x + A) = x$ #override if necessary
```

Once we have f and g, we can check $x+A = y+A$ by $g(x+A) = g(y+A)$. Moreover, we if let a set Q be the range of g, then, when we need to iterate through $(\mathbb{F}_{2^{2t-1}}/A)$, we just iterate through Q and use f to obtain $x+A$ from x. So we perform $(x+A)+(y+A) = (x+y)+A$ by $g(f(x + y))$. The rest of the construction follows easily.

4.2.12 Symplectic covers

Definition 4.29. *We say that a bilinear form $\langle -, - \rangle : V \times V \to K$ on an even dimensional vector space V is* symplectic *if $\langle v, w \rangle = -\langle w, v \rangle$ and $\langle v, v \rangle = 0$.*

For the purpose of this report, we will always use the form $\langle v, w \rangle = v^T A_{2n} w$ where A_{2n} is the $2n \times 2n$ matrix given by

$$A_{2n} = \begin{pmatrix} 0_n & I_n \\ -I_n & 0_n \end{pmatrix}$$

Browuer *et al.* (1989)[5] constructs antipodal covers of complete graphs using symplectic

forms in the following way:
Let $V = (\mathbb{F}_q)^n$ where n is even. Let $\langle -, - \rangle$ be a nondegenerate symplectic form on V and let A be any subgroup of the additive group $\mathbb{F}_q$. Then the vertex set is $(\mathbb{F}_q/A) \times V$ and two vertices $(\lambda + A, v)$, $(\mu + A, w)$ are adjacent if and only if $v \neq w$ and $\langle v, w \rangle \in \lambda - \mu + A$.

Note that the adjacency condition is equivalent to $\langle v, w \rangle = \lambda - \mu + a$ for some $a \in A$. Hence, $\lambda = \langle v, w \rangle + \mu - a$ which means that $\lambda + A = \langle v, w \rangle + \mu + A$. We have already discussed in section 4.2.11 how to find a subgroup of the additive group $\mathbb{F}_q$ and also how to perform operations on the quotient group. Thus it becomes relative straightforward to build the graphs described. However, we try to minimise the number of symplectic form operations by noting that $\langle v, w \rangle = -\langle w, v \rangle$. So we iterate through the ordered pairs (v, w) and generate all edges between $(\mu + A, v)$ and $(\lambda + A, w)$ (for any μ, λ) computing $\langle v, w \rangle$ once.

4.2.13 Coset Graphs of Linear Codes

Brouwer *et al.* (1989)[5] describe the following way of obtaining graphs from subspaces of vector spaces:
Let V be a vector space of dimension n over $\mathbb{F}_q$ and C a subspace. The coset graph $\Gamma(C)$ has vertices the cosets of C in V and 2 cosets are adjacent if they have representatives that differ in one coordinate.
Brouwer *et al.* (1989)[5] often refer to the subspace C as "a linear code" because C is often constructed as the codebook of a linear code. However, we are not interested in coding theory and so we will stick to the familiar terminology.

In order to use the above construction we need a way to handle the quotient vector space. So the actual algorithm implemented is the following:
Let U be the complement of C, i.e. $U \oplus C = V$. Then let $P : V \to U$ be the projection map $u + c \mapsto u$ for $u \in U$ and $c \in C$. The vertex set is U and the neighbours of $u \in U$ are $\{u + \lambda P(e_i) | \lambda \in \mathbb{F}_q \setminus \{0\}, 1 \leq i \leq n\}$ where e_i are the standard basis.

To show the equivalence of the two constructions we have the following lemma:

Lemma 4.30. *There is a bijection $\bar{P} : V/C \to U$ such that $v + C \sim w + C$ if and only if $\bar{P}(v + C) - \bar{P}(w + C) = \lambda P(e_i)$ for some i and some $\lambda \neq 0$.*

Proof. Let $\bar{P}(v+C) = P(v)$. Then $\bar{P}$ is well-defined and injective since $v+C = w+C \iff v - w = c \in C \iff P(v) = P(w)$. Moreover, $\bar{P}$ is surjective since for any $u \in U$ we have $\bar{P}(u + C) = P(u) = u$.
Now, $v+C \sim w+C$ if and only if there are v', w' such that $v'+C = v+C$, $w'+C = w+C$ and v', w' differ in only one coordinate. v', w' differ in only one coordinate if and only if $v' - w' = \lambda e_i$ for some i and $\lambda \neq 0$. So $v + C \sim w + C \iff v' - w' = \lambda e_i \iff P(v') - P(w') = \lambda P(e_i) \iff \bar{P}(v + C) - \bar{P}(w + C) = \lambda P(e_i)$. □

The above lemma shows that the two graphs described above are isomorphic. Moreover, the construction we proposed is quite efficient since $\lambda P(e_i)$ can be precomputed and so all edges are generated via a vector addition.

The only aspect left is to describe how to find U and $P(e_i)$. We compute U by noting that $U \cong V/C$ and SageMath can already compute such embedding. This method boils down to computing the row reduced echelon form of a few matrices, so it not very computationally expensive. Computing $P(e_i)$ can be more complicated. First note that for $e_i \in U$ $P(e_i) = e_i$ and so we may get some of them for free. For the others we note the following:
Let U have basis $\{u_1, \dots, u_k\}$ and C have basis $\{c_1, \dots, c_l\}$. Define a matrix A whose columns are $[u_1, \dots, u_k, c_1, \dots, c_l]$. Then $A\alpha = e_i$ implies that $\sum_{i=1}^{k} \alpha_i u_i + \sum_{i=1}^{l} \alpha_{k+i} c_i = e_i$ where $\alpha = (\alpha_1, ..., \alpha_{k+l})$. Hence $P(e_i) = \sum_{i=1}^{k} \alpha_i u_i$.
So we can find $P(e_i)$ by computing $\alpha = A^{-1} e_i$, where we have that A is invertible since $\{u_1, \dots, u_k, c_1, \dots, c_l\}$ is a basis for V.

4.2.14 Kasami codes

Brouwer *et al.* (1989)[5] prove that the coset graph of the Kasami codes is distance-regular. Using the construction from the previous section, we can build the coset graph quite quickly, so here is only a matter of constructing the Kasami codes.

Brouwer *et al.* (1989)[5] describe these codes as follow:

Definition 4.31. *Let s, t be powers of 2.*
Define $K(s,t) = \{v \in (\mathbb{F}_2)^s \mid \sum_{\alpha\in\mathbb{F}_s} v[\alpha] = \sum_{\alpha\in\mathbb{F}_s} v[\alpha]\alpha = \sum_{\alpha\in\mathbb{F}_s} v[\alpha]\alpha^{t+1} = 0\}$, where $v[\alpha]$ is the α^{th} entry of v starting from 0. If either $s = t^2$ or $t = 2^{mk}$ and $s = 2^{(2j+1)k}$ where $m \leq j$ and $\gcd(m, 2j+1) = 1$, then $K(s,t)$ is called the extended Kasami code. *The* Kasami codes *are obtained from the extended Kasami codes by truncating the first entry from all vectors.*

The trivial approach of going through $V = (\mathbb{F}_2)^s$ and checking the three sums is too slow as we need to check 2^s vectors and each check is $O(s)$ operations over a finite field. To solve this issue we note that

$$K(s,t) = \Big\{v \in (\mathbb{F}_2)^s \mid \sum_{\alpha\in\mathbb{F}_s} v[\alpha] = 0\Big\} \cap \Big\{v \in (\mathbb{F}_2)^s \mid \sum_{\alpha\in\mathbb{F}_s} v[\alpha]\alpha = 0\Big\} \\ \cap \Big\{v \in (\mathbb{F}_2)^s \mid \sum_{\alpha\in\mathbb{F}_s} v[\alpha]\alpha^{t+1} = 0\Big\} \quad (4.2.4)$$

The first subspace has an easy basis. For the other two finding a basis is more complicated as it depends on the additive structure of $\mathbb{F}_s$. However, the following lemma let us find a rather small spanning set.

Lemma 4.32. *Let V be a vector space of dimension $n > 0$ over $\mathbb{F}_2$. Let $\{\lambda_i \mid 1 \leq i \leq n\}$ be a set closed under addition. Then the subspace $U = \{v \mid \sum_{i=1}^n v_i\lambda_i = 0\}$, has spanning set $S = \{e_i + e_j + e_k \mid \lambda_k = \lambda_i + \lambda_j\}$.*

Proof. We denote $e_i+e_j+e_k$ by v_{ijk}. Note that $\sum_{l=1}^n v_{ijk}[l]\lambda_l = \lambda_i+\lambda_j+\lambda_k = \lambda_k+\lambda_k = 0$ since we are in characteristic 2. So $\text{Span}(S) \subseteq U$. For the converse, let $u \in U$ we proceed by induction on the number of non-zero entries of u.
Base cases:
If $u = 0$, we are done.
If u has only 1 non-zero entry, say $u[i]$. Then $u \in U$ implies $\lambda_i = 0$ so for $j \neq i$ $\lambda_j + \lambda_i = \lambda_j$ and so $e_j + e_i + e_j = e_i$ is in S. Hence $u = e_i \in S$.
Inductive step:

Assume u has at least 2 non zero entries $u[i], u[j]$ with $i \neq j$ and let $\lambda_k = \lambda_i + \lambda_j$ and $u' = u + v_{ijk}$. Then $u' \in U$ (by the previous part) and the non-zero entries of $u' = u + v_{ijk}$ are those of u taken away i, j and either removing or adding k. Thus u' has at least 1 less non-zero entry of u. By inductive hypothesis $u' \in \text{Span}(S)$ and so is u. □

If we can apply the above lemma to the extended Kasami codes, then we can easily find spanning sets for the sets in 4.2.4 and so deduce $K(s,t)$. So we need to show that $\{\alpha \mid \alpha \in \mathbb{F}_s\}$ and $\{\alpha^{t+1} \mid \alpha \in \mathbb{F}_s\}$ are closed under addition. The first set is a field so we are done. The second set is trickier and we solve the issue in the following lemma:

Lemma 4.33. *Let s, t be as in the extended Kasami codes. Then the set $S = \{\alpha^{t+1} \mid \alpha \in \mathbb{F}_s\}$ is closed under addition.*

Proof. We distinguish the two possible cases for s and t.
Assume $s = t^2$. Then $\mathbb{F}_t$ is a subfield of $\mathbb{F}_s$ and $\mathbb{F}_t = S$. In particular $\mathbb{F}_t = \{\alpha \in \mathbb{F}_s \mid \alpha^t = a\}$. Note that $(\alpha^{t+1})^t = \alpha^{t^2+t} = \alpha^s \alpha^t = \alpha\alpha^t = \alpha^{t+1}$.
Assume $t = q^m$, $s = q^{2j+1}$ with q a power of 2, $m \leq j$ and $\gcd(m, 2j+1) = 1$. Let g be a generator of $(\mathbb{F}_s)^\times$, then S is the subgroup generated by g^{t+1} together with 0. If $\gcd(t+1, s-1) = 1$, then g^{t+1} is itself a generator and so $S = \mathbb{F}_s$. So let's prove $\gcd(t+1, s-1) = 1$. Let p be a prime dividing $\gcd(t+1, s-1) = \gcd(q^m+1, q^{2j+1}-1)$, then $q^{2j+1} \equiv 1 \mod p$ and $q^m \equiv -1 \mod p$, which implies $q^{2m} \equiv 1 \mod p$. So the order of q (in $(\mathbb{Z}_p)^\times$) divides $2j+1$ and $2m$. Since $\gcd(2j+1, m) = 1$ we must have that the order of q is 1. Hence $q \equiv 1 \mod p$. Thus $p \mid q-1$. So

$$p \mid q-1, q^m+1, q^{2j+1}-1$$

Hence, $p \mid \gcd(q-1, q^m+1)$. Note $q^m + 1 = ((q-1)+1)^m + 1 \equiv 2 \mod q-1$. So $\gcd(q-1, q^m+1) = \gcd(q-1, 2) = 1$ since $q-1$ odd. Hence $p = 1$ which means $\gcd(t+1, s-1) = 1$ and the result follows. □

4.2.15 AB graph

This graph was first constructed by van Dam and Fon-Der-Flaass (2002)[20] and it requires some more background material. Hence, we follow the introduction of that paper and state a few definitions.

Definition 4.34. *Let V be an n dimensional vector space over $\mathbb{F}_2$. Let $f : V \to V$ be any function. The Fourier transform of f is $\mu_f : V \times V \to \mathbb{R}$ given by*

$$\mu_f(x, y) = \sum_{v \in V} (-1)^{\langle x,v \rangle} (-1)^{\langle y,f(v) \rangle}$$

where $\langle -, - \rangle$ is the standard dot product

Definition 4.35. *Let V be as above. A function $f : V \to V$ is called* almost bent *(AB) if*

$$\mu_f(x, y) \in \{\pm 2^{\frac{n+1}{2}}, 0\} \text{ for all } (x, y) \neq (0, 0)$$

Given an AB function f with $f(0) = 0$ the related graph is described as follow:
Let $V \times V$ be the vertices and $(v_1, v_2) \sim (u_1, u_2)$ if $f(v_1 + u_1) = v_2 + u_2$.

This construction is well-suited to build a graph since given a vertex (v_1, v_2) we can define its neighbours by $\{(u_1, u_2) \mid u_2 = f(v_1 + u_1) + v_2\}$. Note that we are in characteristic 2.

So the issue becomes finding an easy-to-compute function f. Luckily, van Dam and Fon-Der-Flaass (2002)[20] list all known (at that time) AB functions. Among those, for odd n we find the Gold's function $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ given by $f(x) = x^3$. Note that here $\mathbb{F}_{2^n}$ is viewed as a vector space over $\mathbb{F}_2$. Van Dam *et al.* (2016)[19] also quote some new result about AB functions, but no AB function on even dimensional vector spaces is known. This is quite unfortunate as for odd n the intersection array of the AB graphs is the same as some Kasami graphs (section 4.2.14). However, we decided to implement this construction regardless in the hope that AB functions on even n will soon be found. Moreover, it is quicker to construct the AB graph than its related Kasami graph.

5 Conclusion

In chapter 1 we set ourselves the aim to construct all infinite families mentioned in the two monographs Brouwer *et al.* (1989)[5] and van Dam *et al.* (2016).[19] We have achieved our target and went beyond by constructing several "sporadic" graphs. In particular, we implemented 26 new constructions which lead to 29 new families of graphs being available to SageMath together with 30 new "sporadic" graphs. These cover all infinite families mentioned in the two monographs and all graphs of diameter at least 4 listed in the 'Tables of Parameters' from 'Distance-Regular Graphs' by Brouwer *et al.* (1989).[5] We also expanded SageMath in other areas by adding 7 new constructions of other combinatorial objects. All the code developed in this project is available online and is slowly finding its way into the main release of SageMath. Thanks to the help of my project supervisor, this process will continue throughout the summer, so that all our efforts will not be in vain. Moreover, this process highlighted typos on common web resources such as WolframAlpha[24] and distanceregular.org[1] as well as an oversight in a construction described in the erratum by Brouwer.[4]

5.1 Future work

Our module is by no means perfect and a lot of more work can be done. In particular, we highlight the following few points that the SageMath's community should consider working on:

1. Complete the database of constructions by adding the few sporadic graphs missing together with any potential newly discovered infinite family;
2. Improve the efficiency and robustness of the `drg` module;
3. Add support for parallel computation;
4. Expand the constructions of related combinatorial objects.

Finally, let me point out that 3 has always been in the back of our mind since new hardware tends to expand the number of logical threads rather than CPU clock-speed. As a result, most constructions we implemented can be parallelised with little effort. In particular, whenever the set of edges is generated by checking adjacency on a pair of vertices or by constructing the set of neighbours for a given vertex, one can parallelise the algorithm via a simple bag-of-tasks pattern.

6 References

[1] Robert Bailey. Distanceregular.org. `https://www.distanceregular.org`.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[3] Thomas Beth, Deiter Jungnickel, and Hanfried Lenz. *Design Theory: Volume 1*. Cambridge University Press, 1999.

[4] A.E. Brouwer. Correction and additions to the book 'distance-regular graphs'. https://www.win.tue.nl/ aeb/drg/index.html.

[5] A.E. Brouwer, A.M. Cohen, and A. Neumaier. *Distance-Regular Graphs*. Springer-Verlag, Berlin, 1989.

[6] Andries Brouwer and Dmitrii Pasechnik. Two distance-regular graphs. *Journal of Algebraic Combinatorics*, 36, 07 2011.

[7] Peter J. Cameron. Covers of graphs and egqs. *Discrete Mathematics*, 97(1):83 – 92, 1991.

[8] D. de Caen, R. Mathon, and G. E. Moorhouse. A family of antipodal distance-regular graphs related to the classical preparata codes. *Journal of Algebraic Combinatorics*, 4(4):317–327, 1995.

[9] R.H.F. Denniston. Some maximal arcs in finite projective planes. *Journal of Combinatorial Theory*, 6(3):317 – 319, 1969.

[10] David A. Drake. Partial λ-geometries and generalized hadamard matrices over groups. *Canadian Journal of Mathematics*, 31(3):617–627, 1979.

[11] Walter Feit and Graham Higman. The nonexistence of certain generalized polygons. *Journal of Algebra*, 1(2):114 – 131, 1964.

[12] GAP – Groups, Algorithms, and Programming, Version 4.10.2. `https://www.gap-system.org`, Jun 2019.

[13] Christopher D. Godsil. Interesting graphs and their colourings. 2003. Available on: semanticsholar.org.

[14] Larry C Grove. *Classical groups and geometric algebra*, volume 39 of *Graduate Studies in Mathematics*. American Mathematical Soc., 2002.

[15] Stanley Payne and J. Thas. Finite generalized quadrangles. *Research Notes in Mathematics*, 110, 01 1984.

[16] J. A. Thas. Construction of maximal arcs and partial geometries. *Geometriae Dedicata*, 3(1):61–64, 1974.

[17] J. A. Thas and S. E. Payne. Spreads and ovoids in finite generalized quadrangles. *Geometriae Dedicata*, 52(3):227–253, 1994.

[18] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1.rc3)*, 2020. `https://www.sagemath.org`.

[19] Edwin R van Dam, Jack H Koolen, and Hajime Tanaka. Distance-regular graphs. *The Electronic Journal of Combinatorics*, Dynamic Surveys, 2016. `https://www.combinatorics.org/ojs/index.php/eljc/article/view/DS22`.

[20] E.R. van Dam and D. Fon-Der-Flaass. Codes, graphs, and schemes from nonlinear functions. *European Journal of Combinatorics*, 24(1):85 – 98, 2003.

[21] Hendrik Van Maldeghem. *Generalized polygons*. Springer Science & Business Media, 2012.

[22] Janoš Vidali. Using symbolic computation to prove nonexistence of distance-regular graphs. *Electronic Journal of Combinatorics*, 25(4):P4.21, 2018. `http://www.combinatorics.org/ojs/index.php/eljc/article/view/v25i4p21`.

[23] Janoš Vidali. `jaanos/sage-drg: sage-drg` v0.9, 2019. `https://github.com/jaanos/sage-drg/`, `10.5281/zenodo.1418409`.

[24] Eric W. Weisstein. M_22 graph. `https://mathworld.wolfram.com/M22Graph.html`.

[25] R. A. Wilson, R. A. Parker, S. Nickerson, J. N. Bray, and T. Breuer. AtlasRep, a gap interface to the atlas of group representations, Version 2.1.0. `http://www.math.rwth-aachen.de/~Thomas.Breuer/atlasrep`, May 2019. Refereed GAP package.

A Code

Since we have a strict page limit, we can't include all code, hence below we include: a testing script; sample outputs; the constructions described; some other examples.

A.1 Main function

```
def distance_regular_graph( list arr, existence=False, check=True ):
    import drg
    from drg import InfeasibleError

    def result(G):
        if check:
            array = intersection_array_from_graph(G)
            if array != arr:
                raise RuntimeError("Sage built the wrong
         distance-regular graph; expected {}, result
         {}".format(arr,array))
        return G

    def is_iterable(obj):
        try:
            iter(obj)
            return True
        except TypeError:
            return False

    n = len(arr)
    d = n // 2
    #check that arr makes sense:
    try:
        parameters = drg.DRGParameters(arr[:d],arr[d:])
    except (AssertionError, InfeasibleError, TypeError) as err:
        if existence: return False
        raise EmptySetError(
            "No distance-regular graphs with parameters {} exists;
         reason: {}".format(arr,err))

    #handle diameter < 3
    if d == 1 and arr[1] == 1:
        if existence: return True
        return result(GraphGenerators.CompleteGraph(arr[0]+1))
    if d == 2:
        k = arr[0]
        mu = arr[3]
        l = k -arr[1]-1 #a1 = k - b1 -c1
        v = number_of_vertices_from_intersection_array(arr)
        if existence: return
         strongly_regular_graph(v,k,l,mu,existence=True)
```

```python
        return result(strongly_regular_graph(v,k,l,mu))

    t = tuple(arr)
    if t in _sporadic_graph_database:
        if existence: return True
        return result(_sporadic_graph_database[t]())

    for (f,g) in _infinite_families:
        t = f(arr)
        if t is not False:
            if existence: return True

            G = g(*t) if is_iterable(t) else g(t)
            return result(G)

    #now try drg feasibility
    try:
        parameters.check_feasible()
    except (InfeasibleError, TypeError, AssertionError) as err:
        if existence: return False
        raise EmptySetError(
            "no distance-regular graph with intersection array {} exists; reason: {}".format(arr,feasible))

    if existence: return Unknown
    raise RuntimeError("No distance-regular graph with intersection array {} known".format(arr))
```

A.2 Testing script

```python
def test_sporadic():
    ok = 0
    fail = 0
    for arr in _sporadic_graph_database:
        start = time()
        G = _sporadic_graph_database[arr]()
        end = time()
        if tuple(intersection_array_from_graph(G)) != arr:
            print("{} {} failed".format(warning,arr))
            fail+= 1
        else:
            print("{} success ({} edges in {})".format(arr,G.size(),end-start))
            ok += 1

    return (ok,fail)

def call_wrapper(f,obj,unpack):
    if unpack:
        return f(*obj)
    else:
        return f(obj)

def is_iterable(obj):
    try:
        iter(obj)
    except TypeError:
        return False
    return True

def test_function(name, graph, params, array, edges=None, fromArray=False):

    ok = 0
    fail = 0
    iterable = True #assume true; doesn't matter

    #default function to compute number of edges
    def edgesDefault(arr):
        try:
            nE = arr[0]*number_of_vertices_from_intersection_array(arr) // 2
        except OverflowError:
            return edgeLimit+1
        return nE

    def edgesWrapper(t):
        try:
            nE = call_wrapper(edges,t,iterable)
        except OverflowError:
            return edgeLimit +1
        return nE

    for t in params():#parmas is a generator
        iterable = is_iterable(t)
        if edges is None:#no specific function; use default
            arr = call_wrapper(array,t,iterable)
            nE = edgesDefault(arr)
            if nE > edgeLimit: continue
        else:
            nE = edgesWrapper(t)
            if nE > edgeLimit: continue
            arr = call_wrapper(array,t,iterable)

        start = time()
        if fromArray:
            G = distance_regular_graph(arr,check=False)
        else:
            G = call_wrapper(graph,t,iterable)
        end = time()

       if intersection_array_from_graph(G) == arr:
            ok += 1
            print("{} with parameters {} success ({} vertices, {} edges in {})".format(name,t,G.order(),G.size(),end-start))
        else:
            fail += 1
            print("{} {} with parameters {} failed".format(warning,name,t))

```

```python
    return (ok,fail)

@fork(timeout=timeLimit,verbose=False)
def timeout_wrapper(f,*t):
    return f(*t)

def test_all(fromArray=False):
    totalOK = 0
    totalFail = 0

    for t in _tests_list:
        name = t[0]
        print("start testing {}".format(name))

        lt = len(t)
        if lt == 2:
            (ok,fail) = t[1]()
        else:
            graph = t[1]
            params = t[2]
            array = t[3]
            edges = None if lt == 4 else t[4]
            res = timeout_wrapper(test_function, name, graph, params, array, edges, fromArray)
            if type(res) == type(""):
                print("timeout..., not counting")
                (ok,fail) = (0,0)
            else:
                (ok,fail) = res

        totalOK += ok
        totalFail += fail
        print("{} terminated: fail {}, ok {}, total {}".format(name,fail,ok,ok+fail))
        print("---------------------------------------------")

    print("All functions tested")
    print("Tests passed {}, failed {}, total {}".format(totalOK, totalFail, totalOK+totalFail))

    def test_random_array(n, drange, maxV , noConstraints=False):
    #we generate n random arrays
    #they should have diameter in drange
    #each entry in the array is <= maxV (use to bound the graph size)
    #size of graph is < maxV * (maxV^d-1)/(maxV-1) / 2
    #if noConstraints is True, then arrays are completely random
    #otherwise we ensure c_1 =1 and c[i] <= c[i+1]
    #and b[i] >= b[i+1] and a_i >=0
    from numpy.random import randint
    D = len(drange)
    ds = randint(0,D,n)

    for v in range(n):
        d = drange[ds[v]]
        if noConstraints:
            arr = list(randint(1,maxV,2*d))
        else:
            bs = [randint(1,maxV-1)]#we will increase b_0 by 1 later
         to ensure b_i < b_0
            for i in range(d-1):
                bs.append(randint(1,bs[i]+1))

            bs[0] += 1
            bs.append(0)#we need b_d = 0 for below
            cs = [1]
            for i in range(d-1):
                cs.append(randint(cs[i],bs[0]-bs[i+2]+1))

            arr = bs[:-1]+cs
        print("testing array {}".format(arr))
        if distance_regular_graph(arr,existence=True) is True:
            print("array is good!")
            n = number_of_vertices_from_intersection_array(arr)
            if n*arr[0] > 2*edgeLimit:
                print("too big")
                continue
            G = distance_regular_graph(arr,check=True)
            print("constructed {}".format(G.name()))

    print("tests terminated")
```

A.3 Sample Outputs

```
sage: from sage.graphs.distance_regular import *
sage: %time alternating_form_graph(6,2)
CPU times: user 6min 17s, sys: 10 s, total: 6min 27s
Wall time: 6min 33s
Alternating form graph on (F_2)^6: Graph on 32768 vertices
sage: G = _
sage: G.size()
10665984

sage: %time G = symplectic_cover(16,2,8)
CPU times: user 2.8 s, sys: 79.2 ms, total: 2.88 s
Wall time: 2.91 s
sage: G.size()
261120
sage: G.order()
2048
sage: %time H = fold_graph(G)
CPU times: user 14.1 s, sys: 212 ms, total: 14.3 s
Wall time: 14.4 s
sage: H.order()
256
sage: H.size()
32640
sage: H.is_isomorphic(graphs.CompleteGraph(256))
```

```
True

sage: %time distance_regular_graph([57,56,56,8,1,1,49,57],existence=True)
CPU times: user 79.2 ms, sys: 3.84 ms, total: 83 ms
Wall time: 65.5 ms
True
sage: %time G = distance_regular_graph([57,56,56,8,1,1,49,57])
CPU times: user 57.1 s, sys: 674 ms, total: 57.8 s
Wall time: 58 s
sage: %time H = distance_regular_graph([57,56,56,8,1,1,49,57],check=False)
CPU times: user 39.5 s, sys: 653 ms, total: 40.2 s
Wall time: 40.3 s
sage: G.size()
211185
sage: G.is_isomorphic(H)
True


#run of the testing script on 2 families (rounded times to nearest ms)
start testing double Grassmann
double Grassmann with parameters (2, 1) success (14 vertices, 21 edges in
        0.086)
double Grassmann with parameters (3, 1) success (26 vertices, 52 edges in
        0.043)
double Grassmann with parameters (4, 1) success (42 vertices, 105 edges
        in 0.228)
double Grassmann with parameters (5, 1) success (62 vertices, 186 edges
        in 0.137)
double Grassmann with parameters (7, 1) success (114 vertices, 456 edges
        in 0.388)
double Grassmann with parameters (2, 2) success (310 vertices, 1085 edges
        in 1.370)
double Grassmann with parameters (3, 2) success (2420 vertices, 15730
        edges in 17.820)
double Grassmann with parameters (4, 2) success (11594 vertices, 121737
        edges in 281.452)
double Grassmann with parameters (2, 3) success (23622 vertices, 177165
        edges in 380.385)
double Grassmann terminated: fail 0, ok 9, total 9
-------------------------------------------------------
start testing hermitian cover
hermitian cover with parameters (2, 3) success (27 vertices, 108 edges in
        0.456)
hermitian cover with parameters (3, 2) success (56 vertices, 756 edges in
        0.012)
hermitian cover with parameters (4, 3) success (195 vertices, 6240 edges
        in 0.075)
hermitian cover with parameters (4, 5) success (325 vertices, 10400 edges
        in 0.130)
hermitian cover with parameters (5, 3) success (378 vertices, 23625 edges
        in 0.305)
hermitian cover with parameters (7, 2) success (688 vertices, 117992
        edges in 2.395)
hermitian cover with parameters (7, 3) success (1032 vertices, 176988
         edges in 4.870)
hermitian cover with parameters (7, 4) success (1376 vertices, 235984
         edges in 19.940)
hermitian cover with parameters (8, 3) success (1539 vertices, 393984
         edges in 17.466)
hermitian cover terminated: fail 0, ok 9, total 9
-------------------------------------------------------
All functions tested
Tests passed 18, failed 0, total 18
```

A.4 Double, Half and Fold

```python
def bipartite_double_graph(G):
    r"""
    Return the bipartite double of G
    """
    edges = []
    for (u,v) in G.edges(sort=False,labels=False):
        sig_check()
        u1 = (0,u)
        u2 = (1,u)
        v1 = (0,v)
        v2 = (1,v)

        edges.append((u1,v2))
        edges.append((u2,v1))

    H = Graph(edges, format='list_of_edges')
    H.name("Bipartite Double of %s"%(G.name()))

    return H


def extended_biparitite_double_graph(G):
    r"""
    Return extended bipartite double of G
    """
    H = bipartite_double_graph(G)
    for u in G.vertices():
        sig_check()
        u1 = (0,u)
        u2 = (1,u)

        H.add_edge((u1,u2))

    H.name("Extended %s"%(H.name()))
    return H


def halve_graph(G) :
    r"""
    Return the half graph of the graph given
    """
    H = GraphGenerators.EmptyGraph()
    queue = [G.vertices()[0]] # queue of vertices to follow
    H.add_vertex(G.vertices()[0])
    while queue:
        v = queue.pop(0)
        #compute all neighbours of the neighbours
        candidate = set([ x for c in G.neighbors(v,sort=False) for x
         in G.neighbors(c,sort=False) ])
        for w in candidate:
            if not G.has_edge(v,w):#then d(v,w)==2
                if w not in H:
                    queue.append(w)
                    H.add_vertex(w)
                H.add_edge(v,w)

    H.name("Halved %s" % G.name() )
    return H


def fold_graph(G, d=None):
    r"""
    Return the fold of G.

    If d is not None, then we assume that G.diameter() == d
    """
    distance = G.distance_all_pairs()

    if d is None:
        d= G.diameter()

    #go through vertices
    #if d(u,v) == d, then they are in a clique
    vertices = set(G.vertices(sort=False))
    cliques = []
    while vertices:
        v = vertices.pop()
        clique = [v]
        for u in vertices:
            if distance[v][u] == d:
                clique.append(u)

        vertices = vertices.difference(clique)
        cliques.append(frozenset(clique))

    N = len(cliques)
    edges = []
    for i in range(N):
        cl1 = cliques[i]
        for j in range(i+1,N):
            cl2 = cliques[j]

            #look for edge connecting cliques
            edge=False
            for u in cl1:
                for v in cl2:
                    if G.has_edge((u,v)):
                        edge=True
                        break
                if edge:
                    break

            if edge:
                edges.append((i,j))

    H = Graph(edges,format="list_of_edges")
    H.name("Fold of %s" % (G.name()) )
    return H
```

A.5 Dual Polar graphs

```
def dual_polar_orthogonal(const int e, const int  d, const int q):
    r"""
    Return dual polar graph on $GO^e(n,q)$ of diameter d

    n is determined by d and e
    """

    def hashable(v):
        v.set_immutable()
        return v

    if e not in {0,1,-1}:
        raise ValueError("e must by 0,+1 or -1")

    m = 2*d + 1 - e

    group = libgap.GeneralOrthogonalGroup(e,m,q)
    M = Matrix(libgap.InvariantQuadraticForm(group)["matrix"])
    #Q(x) = xMx is our quadratic form

    #we need to find a totally isotropic subspace of dimension d
    #attempt 1 (consider kernel)
    K = M.kernel()
    isotropicBasis = list(K.basis())

    #extend K to a maximal isotropic subspace
    if K.dimension() < d:
        V = VectorSpace(GF(q),m)
        candidates = set(map(hashable,[P.basis()[0] for P in V.subspaces(1)]))#all projective points
        hashableK = map(hashable, [P.basis()[0] for P in K.subspaces(1)])
        candidates = candidates.difference(hashableK) #remove all points already in K
        nonZeroScalars = [ x for x in GF(q) if not  x.is_zero() ]
        while K.dimension() < d:
            found = False#found vector to extend K?
            while not found:
                v = candidates.pop()
                if v*M*v == 0:
                    #found another isotropic point
                    #check if we can add it to K
                    found  = True
                    for w in isotropicBasis:
                        if w*M*v+v*M*w != 0:
                            found  = False
                            break
            #end while found
            isotropicBasis.append(v)
            #remove new points of K
            newVectors = map(hashable,[ k+l*v for k in K for l in
         nonZeroScalars ])
            candidates.difference(newVectors)
            K = V.span(isotropicBasis)
        #end while K.dimension
        isotropicBasis = list(K.basis())

    W = libgap.FullRowSpace(libgap.GF(q), m) #W is GAP version of V
    isoS = libgap.Subspace(W,isotropicBasis) #isoS is GAP version of K

    allIsoPoints =
         libgap.Orbit(group,isotropicBasis[0],libgap.OnLines) #all
         isotropic projective points
    permutation = libgap.Action(group, allIsoPoints,libgap.OnLines)
    #this is the permutation group generated by GO^e(n,q) acting on
         projective isotropic points

    #translate K(=isoS) to int for the permutation group
    isoSPoints = [libgap.Elements(libgap.Basis(x))[0] for x in
         libgap.Elements(isoS.Subspaces(1))]
    isoSPointsInt = libgap.Set([libgap.Position(allIsoPoints, x) for
         x in isoSPoints])

    allIsoSubspaces = libgap.Orbit(permutation,isoSPointsInt,
         libgap.OnSets)#our vertices
    intersection_size = (q**(d-1) - 1) / (q-1) #number of projective
         points in a d-1 subspace

    edges = []
    n = len(allIsoSubspaces)
    for i in range(n):
        seti = allIsoSubspaces[i]
        for j in range(i+1,n):
            setj = allIsoSubspaces[j]
            if libgap.Size(libgap.Intersection(seti,setj)) ==
         intersection_size:
                edges.append( (i,j) )

    G = Graph(edges, format="list_of_edges")
    G.name("Dual Polar Graph on Orthogonal group (%d,%d,%d)"%(e,m,q))
    return G
```

A.6 Double Odd graph

```
def doubled_odd_graph( const int n ):
    r"""
    Double odd graph on 2*n+1 points

    Input: n
    """
    if n < 1:
        raise ValueError("n must be >= 1")

    # a binary vector of size 2n+1 represents a set
    edges = []
    for s1 in IntegerVectors(n, k=2*n +1, max_part=1):
        #s1 is a list
        #iterate through it and create s2
        for i in range(2*n+1):
            sig_check()
            if s1[i] == 0:
                s2 = list(s1)
                s2[i] = 1
                #now s2 is a n+1-set containing s1
                edges.append((tuple(s1),tuple(s2)))

    G = Graph(edges, format='list_of_edges')
    G.name("Bipartite double of Odd graph on a set of %d
         elements"%(2*n+1))
    return G
```

A.7 Bilinear, Alternating and Hermitian form graphs

```
def bilinear_form_graph(const int d, const int e, const int q):
    r"""
    Return the bilinear form graph on $d \times e$ matrices over $\mathbb{F}_q$.
    """
    matricesOverq = MatrixSpace( GF(q), d, e,
         implementation='meataxe' )

    rank1Matrices = []
    for m in matricesOverq:
        sig_check()
        if m.rank() == 1:
            rank1Matrices.append(m)

    edges = []
    for m1 in matricesOverq:
        m1.set_immutable()
        for m2 in rank1Matrices:
            sig_check()
            m3 = m1+m2
            m3.set_immutable()
            edges.append( ( m1, m3) )

    G = Graph(edges, format='list_of_edges')
    G.name("Bilinear form graph over F_%d with parameters (%d,%d)"
         %(q,d,e) )
    return G


def alternating_form_graph(const int n, const int q):
    r"""
    Return the alternating form graph on $n \times n$ matrices on $\mathbb{F}_q$
    """
    def symmetry(x): return -x
    def diagonal(x): return 0

    matrices = MatrixSpace(GF(q), n, n, implementation="meataxe")
    skewSymmetricMatrices = matrices.symmetric_generator(symmetry,
         diagonal)

    rank2Matrices = []
    for mat in skewSymmetricMatrices:
        sig_check()
        # check if mat is a rank2 matrix
        if mat.rank() == 2:
            rank2Matrices.append(mat)

    skewSymmetricMatrices = matrices.symmetric_generator(symmetry,
         diagonal)

    # now we have all matrices of rank 2
    edges = []
    for m1 in skewSymmetricMatrices:
        m1.set_immutable()
        for m2 in rank2Matrices:
            sig_check() # check for interrupts
            m3 = m1+m2
            m3.set_immutable()
            edges.append(( m1, m3 ))

    G = Graph(edges, format='list_of_edges')
    G.name("Alternating form graph on (F_%d)^%d" %(q,n) )
    return G


def hermitian_form_graph(const int n, const int q):
    r"""
    Return the Hermitian from graph of $n \times n$ matrices on $\mathbb{F}_q$

    q must be the square of a prime power
    """
    MS = MatrixSpace(GF(q), n, n, implementation="meataxe")

    (b,k) = is_prime_power(q, get_data=True)
    if k == 0 or k % 2 != 0:
        raise ValueError("We need q=r^2 where r is a prime power")

    # here we have b^k = q, b is prime and k is even
    r = b**(k//2)
    # so r^2 = b^k = q

    def symmetry(x): return x**r

    hermitianMatrices = MS.symmetric_generator(symmetry)

    rank1Matrices = []
    for mat in hermitianMatrices:
        sig_check()
        if mat.rank() == 1: rank1Matrices.append(mat)

    #refresh generator
    hermitianMatrices = MS.symmetric_generator(symmetry)
    edges = []
    for mat in hermitianMatrices:
        mat.set_immutable()
        for mat2 in rank1Matrices:
            sig_check()

            mat3 = mat + mat2
            mat3.set_immutable()
            edges.append( (mat, mat3) )

    G = Graph(edges, format='list_of_edges')
    G.name("Hermitian form graph on (F_%d)^%d" %(q,n) )
    return G
```

A.8 Halved Cube

```
def halved_cube( int n ):
    r"""
    Return the graph $\frac{1}{2}H(n,2)$.
    """
    def hamming_distance( v, w ):
        assert( len(v) == len(w),
         "Can't compute Hamming distance of 2 vectors of different
        size!")

        counter = 0
        for i in range(len(v)):
            if ( v[i] != w[i] ):
                counter = counter + 1

        return counter

    if n <= 2:
        raise ValueError("we need n > 2")

    #construct the half cube graph 1/2 H(n,2) ( = H(n,2)_{1-or-2} )
    G = GraphGenerators.CubeGraph(n-1)
    # we use the fact that the vertices are strings and their
        distance is their hamming_distance
    for i in G.vertices():
        for j in G.vertices():
            sig_check()
            if hamming_distance(i, j) == 2 :
                G.add_edge(i,j)

    G.name("Halved %d Cube"%n)
    return G
```

A.9 Grassmann and Double Grassmann graphs

```
def Grassmann_graph( const int q, const int n, const int input_e ):
    r"""
    Return a Grassmann graph $J_q(n,e)$

    $J_q(n,e) \cong J_q(n,n-e)$, so if n< 2e, then we compute $J_q(n,n-e)$
    """
    if n <= input_e + 1:
        raise ValueError(
            "Impossible parameters n <= e+1 (%d <= %d)" %(n,input_e) )

    e = input_e
    if n < 2*input_e:
        e = n - input_e

    PG = Sage_Designs.ProjectiveGeometryDesign(n-1, e-1, q)
    #we want the intersection graph
    #the size of the intersection must be (q^{e-1} - 1) / (q-1)
    size = (q**(e-1) -  1)/(q-1)
    G = PG.intersection_graph([size])
    G.name("Grassmann graph J_%d(%d,%d)"%(q,n,e))
    return G


def double_Grassmann_graph(const int q, const int e):
    r"""
    Return the double Grassmann graph $DJ_q(2e+1,e)$
    """
    n = 2*e+1
    V = VectorSpace(GF(q),n)

    edges = []
    for W in V.subspaces(e+1):
        Wbasis = frozenset(W.basis())
        for U in W.subspaces(e):
            Ubasis = frozenset(U.basis())
            edges.append( ( Wbasis, Ubasis ))

    G = Graph(edges,format='list_of_edges')
    G.name("Double Grassmann graph (%d,%d,%d)"%(n,e,q))
    return G
```

A.10 Generalised Polygons graphs

```
def generalised_dodecagon(const int s, const int t):
    q = 0
    orderType = 0

    if s == 1: #(1,q)
        q = t
    elif t == 1: # (q,1)
        q = s
        orderType = 1
    else:
        raise ValueError("No known dodecagon with given input")

    if not is_prime_power(q):
        raise ValueError("No known dodecagon with given input")

    if orderType == 0:
        #incidence graph of hexagon (q,q)

        H = generalised_hexagon(q,q)
        lines = extract_lines(H)
        points = list(H.vertices())

        edges = []
        for p in points:
            for l in lines:
                sig_check()
                if p in l:
                    edges.append( (p,l) )

        G = Graph(edges, format='list_of_edges')
        G.name("Generalised dodecagon of order (1,%d)"%q)
        return G

    else: #orderType == 1
        #dual
        H = generalised_dodecagon(t,s)
        G = line_graph_generalised_polygon(H)
        G.name("Generalised dodecagon of order (%s,%d)"%(s,t))
        return G


def generalised_octagon(const int s, const int t):
    cdef int q = 0
    cdef int orderType = 0
    if s == 1:# (1,q)
        q = t
    elif t == 1:# (q,1)
        q = s
        orderType = 1
    elif s**2 ==  t:# (q,q^2)
        q = s
        (p,k) = is_prime_power(q,get_data=True)
        if p != 2 or k%2 != 1:
            raise ValueError("generalised octagon (q,q^2) only for q odd powers of 2")
        orderType = 2
    elif t**2 == s: #(q^2,q)
        q = t
        orderType = 1
    else:
        raise ValueError("No known octagon with input")

    if not is_prime_power(q):
        raise ValueError("No known octagon with input")

    if orderType == 0:
        H = strongly_regular_graph((q+1)*(q*q+1), q*(q+1), q-1, q+1, check=False)
        # above is pointgraph of generalised quadrangle (q,q)
        lines = extract_lines(H)
        points = list(H.vertices())
        #points and lines make the quadrangle

        edges = []
        for p in points:
            for l in lines:
                sig_check()
                if p in l:
                    edges.append( (p,l) )

        G = Graph(edges, format='list_of_edges')
        G.name("Generalised octagon of order (1,%d)"%q)
        return G

    elif orderType == 1:
        #dual
        H = generalised_octagon(t,s)
        G = line_graph_generalised_polygon(H)
        G.name("Generalised octagon of order(%d,%d)"%(s,t))
        return G
    else:
        if q == 2:
            g = libgap.AtlasGroup("2F4(2)", libgap.NrMovedPoints, 1755)
            G = Graph( g.Orbit( [1,73], libgap.OnSets), format='list_of_edges')
            G.name("Generalised octagon of order (2,4)")
            return G
        else:
            raise NotImplementedError("graph would be too big")


def generalised_hexagon( const int s, const int t):
    r"""
    to use libgap.AtlasGroup we need to do
    sage -i gap_packages
    """
    cdef int q = 0
    cdef int orderType = 0
    if s == 1: # (1,q)
        q = t
    elif t == 1:# (q,1)
        q = s
        orderType = 1
    elif s == t:# (q,q)
        q = s
        orderType = 2
    elif s**3 == t:# (q,q^3)
        q = s
        orderType = 3
    elif t**3 == s: # (q^3, q)
        q = t
        orderType = 1
    else:
        raise ValueError("invalid input")

    if not is_prime_power(q):
        raise ValueError("invalid input")

    if orderType == 0:
        #incident graph of generalised 3-gon of order (q,q)
        PG2 = Sage_Designs.ProjectiveGeometryDesign(2,1,q)

        edges = []
        for l in PG2.blocks():
            for p in l:
                sig_check()
                edges.append( (p, frozenset(l)) )

        G = Graph(edges, format='list_of_edges')
        G.name("Generalised hexagon of order (1,%d)"%q)
        return G

    elif orderType == 1:
        # "dual" graph
        H = generalised_hexagon(t,s)
        G = line_graph_generalised_polygon(H)
        G.name("Generalised hexagon of order(%d,%d)"%(s,t))
        return G

    elif orderType == 2:
        # we use the group G2(q)
        # if q == 2, then G2(2) is isomorphic to U3(3).2
        if q == 2:
            group = libgap.AtlasGroup("U3(3).2",
         libgap.NrMovedPoints, 63)
            G = Graph( group.Orbit([1,19], libgap.OnSets),
         format='list_of_edges')
            G.name("Generalised hexagon of order (%d,%d)"%(q,q))
            return G
        elif q == 3: #we don't have permutation rep
            matrixRep = libgap.AtlasGroup("G2(3)", libgap.Position,7)
            e1 = vector(GF(3), [1,0,0,0,0,0,0])
            orb = matrixRep.Orbit(e1, libgap.OnLines)
            group = libgap.Action(matrixRep,orb,libgap.OnLines)
            #now group is our permutation rep
            G = Graph(group.Orbit([1,52], libgap.OnSets), format='list_of_edges')
            G.name("Generealised hexagon of order (%d,%d)"%(q,q))
            return G
        elif q <= 5:
            arr = intersection_array_2d_gon(3,s,t)
            n = number_of_vertices_from_intersection_array(arr)
            G = graph_from_permutation_group( libgap.AtlasGroup("G2(%d)"%q, libgap.NrMovedPoints, n), arr[0])
            G.name("Generalised hexagon of order (%d,%d)"%(q,q))
            return G
        else:
            raise NotImplementedError("graph would be too big")

    elif orderType == 3:
        if q> 3: raise ValueError("graph would be too big")
        movedPoints = 819 if q==2 else 26572
        group = libgap.AtlasGroup("3D4(%d)"%q, libgap.NrMovedPoints, movedPoints)
        G = Graph(group.Orbit([1,2],libgap.OnSets), format='list_of_edges')
        G.name("Generalised hexagon of order (%d,%d)"%(q,q**3))
        return G


def extract_lines( G ):
    r"""
    Return the singular lines of the graph G
    """
    lines = []
    edges = set(G.edges(labels=False,sort=False))

    while edges :
        (x,y) = edges.pop()

        #compute line
        bottomX = set(G.neighbors(x,closed=True))
        bottomY = set(G.neighbors(y,closed=True))
        bottom1 = bottomX.intersection(bottomY)

        b = bottom1.pop()
        bottom2 = frozenset(G.neighbors(b,closed=True))
        for v in bottom1:
            sig_check()
            s = frozenset(G.neighbors(v,closed=True))
            bottom2 = bottom2.intersection(s)

        #now bottom2 is a line
        lines.append(tuple(bottom2))#we need tuple or GAP will complain

        #remove pointless edges
        for u in bottom2:
            for v in bottom2:
                try :
                    edges.remove( (u,v) )
                except KeyError:
                    pass #ignore this

    #end while edges
    return lines


def line_graph_generalised_polygon(H):
    r"""
    Given the point graph of a generalised polygon, it computes its
         line graph
    """
    lines = extract_lines(H)

    #get a map point -> all lines incident to point
    vToLines = { v : [] for v in H.vertices(sort=False) }
    for l in lines:
        for p in l:
            sig_check()
            vToLines[p].append(l)

    k = len(vToLines[lines[0][0]])

    edges = []
    for v in vToLines:
        lines = vToLines[v]
        for i,l in enumerate(lines):
            for j in range(i+1,k):
                sig_check()
                edges.append( (l,lines[j]) )

    G = Graph(edges,format="list_of_edges")
    return G


def graph_from_permutation_group( group, const int order ):
    r"""
    Construct graph looking at the orbit of group on an edge (1,x)
    where x is picked from an obit of length order from the
         stabilizer of 1
    """
    h = group.Stabilizer(1)
    orbitIndex = 0
    orbitLenghts = h.OrbitLengths()

    # if we can't find the correct orbit, we raise out of bound error
    while orbitLenghts[orbitIndex] != order:
        orbitIndex += 1

    #now we found the correct orbit
    v = h.Orbits()[orbitIndex][0] #pick an element of the orbit

    G = Graph( group.Orbit( [1,v], libgap.OnSets),
         format='list_of_edges')

    return G
```

A.11 Generalised Quadrangle graph

```
def GQ_spread_graph(GQ, S):
    r"""
    Point graph of the generalised quadrangle GQ without its spread S
    """
    k = len(GQ.blocks()[0])
    edges = []
    for b in GQ.blocks():
        if b in S: continue
        for i in range(k):
            p1 = b[i]
            for j in range(i+1,k):
                sig_check()
                p2 = b[j]
                edges.append( (p1,p2) )

    G = Graph(edges, format="list_of_edges")
    return G


def generalised_quadrangle_hermitian(const int q):
    r"""
    Construct the generalised quadrangle H(3,q^2) with an ovoid
    The GQ has order (q^2,q)
    """
    GU = libgap.GU(4,q)
    H = libgap.InvariantSesquilinearForm(GU)["matrix"]
    Fq = libgap.GF(q*q)
    zero = libgap.Zero(Fq)
    one = libgap.One(Fq)
    V = libgap.FullRowSpace(Fq,4)

    e1 = [one,zero,zero,zero] #isotropic point

    points = list(libgap.Orbit(GU,e1,libgap.OnLines)) #all isotropic
         points
    pointInt = { x:(i+1) for i,x in enumerate(points) } #+1 because
         GAP starts at 1
    #points is the hermitian variety

    GUp = libgap.Action(GU, points, libgap.OnLines)#GU as permutation
         group of points

    e2 = [zero,one,zero,zero]

    line = V.Subspace([e1,e2])#a totally isotropic line
    lineAsPoints = [libgap.Elements(libgap.Basis(b))[0] for b in
         libgap.Elements(line.Subspaces(1)) ]
    line = libgap.Set([ pointInt[p] for p in lineAsPoints ])

    lines = libgap.Orbit(GUp, line, libgap.OnSets)#all isotropic lines

    #to find ovoid, we embed H(3,q^2) in H(4,q^2)
    #then embedding is (a,b,c,d) -> (a,b,0,c,d) [so we preserve
         isotropicity]
    W = libgap.FullRowSpace(Fq,5)
    J = [ [0,0,0,0,1], [0,0,0,1,0], [0,0,1,0,0], [0,1,0,0,0],
         [1,0,0,0,0]]
    J = libgap(J)
    if q%2 == 1:
        (p,k) = is_prime_power(q,get_data=True)
        a = (p-1)// 2
        aGap = zero
        for i in range(a): aGap += one
        p = [zero,one,one,aGap,zero]
    else:
        a = libgap.PrimitiveRoot(Fq)**(q-1)
        p = [zero,one,a+one,a,zero]

    #now p is a point of H(4,q^2) not in H(3,q^2)

    #p' is collinear to p iff p'Jp^q = 0
    #note that p'Jp^q = bx^q + c where p' =(a,b,0,c,d) and
         p=(0,1,1,x,0)
    ovoid = []
    xq = p[3]**q
    for p2 in points:
        if p2[1]*xq+p2[2] == zero:
            ovoid.append(libgap(pointInt[p2]))

    D = IncidenceStructure(lines)
    return (D,ovoid)
```

A.12 Unitary Nonisotropic graph

```
def unitary_nonisotropic_graph(const int q):
    r"""
    Return graph on nonisotropic points for a Hermitian form on (𝔽_{q²})³
    """
    if q < 3:
        raise ValueError("q must be greater than 2")
    if not is_prime_power(q):
        raise ValueError("q must be a prime power")

    GU = libgap.GU(3,q)
    Fr = libgap.GF(q*q)
    one = libgap.One(Fr)
    zero = libgap.Zero(Fr)
    ev = [one,one,zero]
    w = [zero,one,-one]

    vertices = libgap.Orbit(GU,ev,libgap.OnLines)
    PGU = libgap.Action(GU,vertices,libgap.OnLines)

    evPos = -1
    wPos = -1
    for i,v in enumerate(vertices):
        if v == ev:
            evPos = i+1
        if v == w:
            wPos = i+1

        if evPos != -1 and wPos != -1:
            break

    edges = libgap.Orbit(PGU, libgap.Set([evPos,wPos]), libgap.OnSets)

    G = Graph(edges,format="list_of_edges")
    G.name("Unitary nonisotropic graph on (F_%d)^3"%(q*q))
    return G
```

A.13 Hermitian Cover

```
def hermitian_cover(const int q, const int r):
    r"""
    Return the Hermitian $r$-antipodal cover of $K_{q^3+1}$
    """
    if not is_prime_power(q):
        raise ValueError("invalid input: q must be prime power")

    if not( (r%2 == 1 and (q-1)%r == 0) or
            (q%2 == 0 and (q+1)%r == 0) or
            (q%2 == 1 and ((q+1)//2)%r == 0)):
        raise ValueError("invalid input")

    Fq2 = libgap.GF(q*q)
    one = libgap.One(Fq2)
    zero = libgap.Zero(Fq2)
    gen = libgap.Z(q*q)

    Kreps = [ gen**i for i in range(r) ]#representatives of quotient
        $\mathbb{F}_{q^2}/K$

    #vertices are Kv for isotropic v
    GU = libgap.GU(3,q)
    e1 = [one,zero,zero]
    iso_points = libgap.Orbit(GU,e1,libgap.OnLines)

    vertices = [ k*v for k in Kreps for v in iso_points ]

    #create global variable for function
    libgap.set_global("zero",zero)
    libgap.set_global("r",r)
    libgap.set_global("gen",gen)

    #we need to define the action of GU on (k,v)
    func = """OnKLines := function(v,M)
        local w, i, b, k;

        w := ShallowCopy(v*M);

        i := 1;
        while i < 4 do
            if w[i] <> zero then
                break;
            fi;
            i := i+1;
        od;
        b := w[i];

        i := 1;
        while i < 4 do
             w[i] := w[i]/b;
             i := i+1;
        od;

        k := LogFFE(b,gen);
        i := k mod r;
        b := gen^i;

        return b*w;
    end;"""

    gapOnKLines = libgap.eval(func)
    GUAction = libgap.Action(GU,vertices,gapOnKLines)

    e3 = [zero,zero,one]#other isotropic, with H(e3,e1) = 1
    e1pos = libgap.Position(vertices,e1)
    e3pos = libgap.Position(vertices,e3)

    #now we have that
    #(e1pos, e11pos) is an edge
    edges = libgap.Orbit(GUAction,[e1pos,e3pos], libgap.OnSets)
    G = Graph(edges, format="list_of_edges")
    return G
```

A.14 Pasechnik and Brouwer-Pasechnik graphs

```
def Brouwer_Pasechnik_graph(const int q):
    r"""
    Return the Brouwer-Pasechnik graph on $\mathbb{F}_q$
    """
    Fq = GF(q)

    def cross(v,w):
        z = [ v[1]*w[2]-v[2]*w[1], v[2]*w[0]-v[0]*w[2],
         v[0]*w[1]-v[1]*w[0] ]
        return vector(Fq,z)

    V = list(VectorSpace(Fq,3))
    for v in V:
        v.set_immutable()

    edges = []
    for u in V:
        for v in V:
            for v2 in V:
                sig_check()
                if v2 == v: continue #otherwise cross(v,v2) == 0 and
     u2 == u
                u2 = u+ cross(v,v2)
                u2.set_immutable()
                edges.append(( (u,v),(u2,v2) ))

    G = Graph(edges,format="list_of_edges")
    G.name("Brouwer-Pasechnik graph on GF(%d)"%q)
    return G


def Pasechnik_graph(const int q):
    r"""
    Return Pasechnik graph on $\mathbb{F}_q$.
    """
    H = Brouwer_Pasechnik_graph(q)
    G = extended_biparitite_double_graph(H)
    G.name("Pasechnik graph on D_4(%d)"%q)
    return G
```

A.15 TD graphs

```
def graph_from_TD(const int m, const int u):
    r"""
    Return the incidence graph of a symmetric transversal design with
         parameters m, u.
    """
    SN = Sage_Designs.symmetric_net(m,u)
    return SN.incidence_graph()


def prime_power_and_2_difference_matrix(q):
    r"""
    Return a (q,2q,2) difference matrix where q is a prime power.
    """
    if q % 2 == 0:
        (p,i) = is_prime_power(q,get_data=True)
        return prime_power_difference_matrix(2,i,1)

    Fq = FiniteField(q)
    elems = [x for x in Fq]
    l = len(elems)
    n = Fq.primitive_element() #we only need a non-square, but this
         should do

    D = [ [0]*(2*q) for i in range(2*q) ]
    for i in range(1,5):
        for x in range(l):
            for y in range(l):
                if i == 1:
                    d = elems[x]*elems[y] + (elems[x]**2 / 4)
                elif i == 2:
                    d = elems[x]*elems[y] + (n*elems[x]**2 / 4)
                elif i == 3:
                    d = elems[x]*elems[y] - elems[y]**2 - (elems[x]**2 / 4)
                elif i == 4:
                    d = (elems[x]*elems[y] - elems[y]**2 - elems[x]**2 / 4) / n

                rowshift = q if i > 2 else 0
                colshift = q if i%2 == 0 else 0
                D[x + rowshift][y + colshift] = d

    return (Fq,D)


def prime_power_difference_matrix(p,i,j):
    r"""
    Return a (p^i, p^(i+j), p^j) difference matrix where p is a prime.
    """
    from sage.modules.free_module_element import vector

    G = FiniteField(p**(i+j))
    elemsG = [x for x in G]
    K = [ [ x*y for y in elemsG] for x in elemsG]

    #we need to map G to (Z_p)^(i+j)
    x = G.gen()
    Fp = FiniteField(p)

    basis = [x**l for l in range(i+j) ]
    iso = {}
    for v in VectorSpace(Fp,i+j):
        y = 0
        for l in range(i+j):
            y += v[l]*(x**l)
        iso[y] = v

    H = [ [ tuple(iso[x][:i]) for x in row] for row in K ]

    #So H is Over (Z_p)^i
    V = VectorSpace(Fp,i)

    return (V,H)


def subgroup_construction(g,k,lmbda,existence=False):
    r"""
    Return a (g,k,\lambda) difference matrix using a subgroup construction
    """
    #here we assume (g,k,lmbda) = (g2/s,k,lmbda2*s)
    #and try to construct (g2,k,lmbda2)
```

```python

    possibleS = divisors(lmbda)
    possibleS = possibleS[1:] #remove s=1

    for s in possibleS:
        g2=g*s
        lmbda2 = lmbda//s
        exists = difference_matrix(g2,k,lmbda2,existence=True)
        if exists is not True:
            continue

        (G,M) = difference_matrix(g2,k,lmbda2)

        if G in FiniteFields:
            if existence: return True
            #then G is essentially a vectorspace
            (G,fr,to) = G.vector_space(map=True)

            #map elements of M to the vector space
            for i in range(lmbda2*g2):
                for j in range(k):
                    M[i][j] = to(M[i][j])

        #now we need to find (if it exists) a normal subgroup of G of order s
        if G in VectorSpaces:
            if existence: return True
            Fp = G.base_field()
            p = Fp.characteristic()
            n = 1
            m = p
            while m < s:
                m *= p
                n += 1
            #so n = dimension of H

            n = G.dimension() - n #dimension of G/H
            GH = VectorSpace(Fp,n)

            #now map all elements of M into G/H
            for i in range(lmbda2*g2):
                for j in range(k):
                    M[i][j] = tuple(M[i][j][:n]) #truncate vector

            return GH,M
        else:
            #we don't handle this at this moment
            continue

    if existence:
        return False

    raise EmptySetError("no subgroup construction found")
```

A.16 BIBD graphs

```
def graph_from_square_BIBD(const int v, const int k):
    r"""
    Returns the incidence graph of a symmetric (or square) BIBD with
         v points and block size k
    """
    if v == 1 or (k*(k-1))%(v-1) != 0:
        raise ValueError("no square BIBD exists with v={},
         k={}".format(v,k))
    lmbd = (k*(k-1))//(v-1)
    D = Sage_Designs.balanced_incomplete_block_design(v, k, lmbd=lmbd)
    return D.incidence_graph()
```

A.17 Taylor graph

```
def Taylor_graph(const int n, const int l):
    r"""
    Return the Taylor graph related to the two-graph with parameters
         n, l
    """
    D = two_graph(n,l,regular=True,check=False)
    G = graph_from_two_graph(D)
    return G


def graph_from_two_graph( D ):
    r"""
    Given a two graph (block design) it builds the graph associated
         with it.
    """
    edges = []

    inf = D.ground_set()[0]

    #first we do all coherent edges
    S = set() #set of coherent pairs
    for b in D.blocks():
        sig_check()
        if b[0] == inf: x=b[1]; y=b[2]
        elif b[1] == inf: x=b[0]; y=b[2]
        elif b[2] == inf: x=b[0]; y=b[1]
        else: continue
        #now x,y,inf are coherent
        S.add( frozenset([x,y]) )
        edges.append( ((x,0),(y,0)) )
        edges.append( ((x,1),(y,1)) )

    #inf is coherent with any other vertex!
    for x in D.ground_set()[1:]:#we don't want edge inf inf
        sig_check()
        edges.append( ((x,0),(inf,0)) )
        edges.append( ((x,1),(inf,1)) )
        S.add( frozenset([x,inf]) )

    #now we can handle the non-coherent ones
    l = D.num_points()
    for i in range(l):
        x = D.ground_set()[i]
        for j in range(i+1,l):#go through all ordered pairt
            sig_check()
            y = D.ground_set()[j]
            if frozenset([x,y]) in S: continue#x,y,inf coherent
            #otherwise add edge
            edges.append( ((x,0),(y,1)) )
            edges.append( ((x,1),(y,0)) )

    G = Graph(edges,format="list_of_edges")
    return G
```

A.18 Denniston graph

```
def graph_from_Denniston_arc(const int n):
    r"""
    Returns the distance regular graph related to the Denniston n-arc
        on 𝔽_{n²}
    """
    (p,k) = is_prime_power(n,get_data=True)
    if p != 2:
        raise ValueError("input must be a power of 2")

    q = n*n
    Fq = GF(q)
    Fn = GF(n)
    elemsFq = [ x for x in Fq]

    #ensure elemsFq[0] == 0
    if not elemsFq[0].is_zero():
        for i,x in enumerate(elemsFq):
            sig_check()
            if x.is_zero():
                y = elemsFq[0]
                elemsFq[0] = x
                elemsFq[i] = y
                break

    #find irreducible quadratic
    candidates = set(Fq)
    for x in elemsFq[1:]:#we rely on the first element to be 0
        sig_check()
        a = x + (1/x)
        candidates = candidates.difference({a})

    irrCoef = candidates.pop()
    def Q(x,y):
        return x*x+irrCoef*x*y+y*y

    PG = Sage_Designs.ProjectiveGeometryDesign(2,1,q) #projective
        plane PG(2,q)
    #the points are represented as vectors with homogeneous
        coordinates (first non-zero entry is 1)

    arc = set() #complete arc
    for x in elemsFq:
        for y in elemsFq:
            sig_check()
            if Q(x,y) in Fn:
                arc.add(vector(Fq,[1,x,y],immutable=True))

    #pick all lines intersecting arc in n points (so any line
        intersecting the arc)
    #remove all points in arc
    lines = []
    for b in PG.blocks():
        sb = Set(b)
        for p in b:
            sig_check()
            if p in arc:
                newLine = sb.difference(arc)
                lines.append(newLine)
                break

    #now we have a list of all lines of the complement
    edges = []
    for b in lines:
        bs = frozenset(b)
        for p in b:
            sig_check()
            edges.append( (p,bs) )

    G = Graph(edges,format="list_of_edges")
    G.name("Incidence graph of the complement of a complete %d-arc in
        PG(2,%d)"%(n,q))
    return G
```

A.19 Association Schemes graph

```
def association_scheme_graph(scheme, inf="inf"):
    r"""
    Return the graph related to the given association scheme.

    We need inf not to be a point of the scheme
    """
    if inf in scheme.ground_set():
        raise ValueError("inf must not be in the association scheme")

    r = scheme.num_classes()
    X = scheme.ground_set()
    I = list(range(1,r+1))

    edges = []
    for x in X:
        for i in I:
            edges.append(( (inf,i),(x,i) ))

    n = scheme.num_points()
    for x in range(n):
        for y in range(x+1,n):
            ij = scheme.matrix()[x][y]
            for i in I:
                j = (ij -i)%r
                if j == 0: j = r
                edges.append(( (X[x],i),(X[y],j) ))
                edges.append(( (X[y],i),(X[x],j) ))

    G = Graph(edges,format="list_of_edges")
    return G


def cyclotomic_scheme(const int q, const int r, check=True):
    r"""
    Return cyclotomic association scheme on q points and r classes
    """
    if r <= 0 or (q-1)%r != 0:
        raise ValueError("we need r to be a (positive) divisor of
         q-1")

    Fq = GF(q)
    X = list(Fq)
    XtoInt = { x: i for i,x in enumerate(X) }

    relations = [ [0 for i in range(q)] for j in range(q)] #qxq matrix

    a = Fq.primitive_element()
    ar = a**r
    m = (q-1)//r
    K = [ ar**i for i in range(m)]
    for i in range(1,r+1):
        ai=a**i
        aiK = [ ai*x for x in K]
        for x in X:
            for z in aiK:
                sig_check()
                y = x+z
                relations[XtoInt[x]][XtoInt[y]] = i

    return AssociationScheme(X, relations, check=check)
```

A.20 Preparata graphs

```
def Preparata_graph(const int t, const int i):
    r"""
    Return Preparata graph on $\mathbb{F}_{2^{2t-1}}$ with subgroup $A$ of size $2^i$
    """
    if i > 2*t-2 or i < 0:
        raise ValueError("i should be between (inclusive) 0 and 2*t-2")

    if t < 1:
        raise ValueError("t should be greater than 1")

    q = 2**(2*t-1)
    Fq= GF(q)

    if i != 0:#then A has some meaning
        (Fqvec,fromV,toV) = Fq.vector_space(map=True)
        n = Fqvec.dimension()
        A = []
        for j in range(i):
            v = [0]*n
            v[j] = 1
            v = vector(Fqvec.base_field(), v)
            A.append(v)

        #now A represents a basis for a vector space of dim i
        A = Fqvec.span(A)
        A = [ fromV(x) for x in A]
        #now A is a subgroup of Fq of size 2^i
        Q = set()
        toQ = {}
        Qrep = {}
        for x in Fq:
            sig_check()
            xA = frozenset( [x+a for a in A])
            toQ[x] = xA
            Qrep[xA] = x
        for k in Qrep:
            Q.add(Qrep[k])

    else:
        Q = Fq

    edges  = []
    for x in Fq:
        x2 = x*x
        x3 = x2*x
        for y in Fq:
            y2 = y*y
            r = x2*y+x*y2
            x3py3 = x3+y2*y
            for a in Q:
                sig_check()
                if x != y or r != 0:
                    b = r+a
                    if i != 0: b = Qrep[toQ[b]]
                    edges.append(( (x,0,a),(y,0,b) ))
                    edges.append(( (x,1,a),(y,1,b) ))
                b = r + x3py3 +a
                if i != 0: b = Qrep[toQ[b]]
                edges.append(( (x,0,a),(y,1,b) ))
                edges.append(( (x,1,a),(y,0,b) ))

    G = Graph(edges, format="list_of_edges")
    G.name("Preparata graph on 2^(2%d-1)"%t)
    return G
```

A.21 Symplectic Cover

```
def symplectic_cover(const int q, const int n, const int r):
    r"""
    Returns an r-antipodal cover of $K_{q^n}$ using a symplectic form over
        $\mathbb{F}_q$
    with a subgroup of index r
    """
    if n <= 0:
        raise ValueError("n must be positive")
    if n%2 == 1:
        raise ValueError("n must be even")
    if q%r != 0:
        raise ValueError("r must be a factor of q")

    def ei(i,m):
        v = [0]*m
        v[i] = 1
        return v

    Fq = GF(q)
    V = VectorSpace(Fq,n)

    if r != q:
        #we need A to be a subgroup of the additive group of Fq
        #so we make Fq a vectorspace and A is a subspace
        (Fqvec,fromVec,toVec) = Fq.vector_space(map=True)
        (p,k) = is_prime_power(r,get_data=True)
        Adim = Fqvec.dimension() -k #|A| = q / r
        A = Fqvec.span([ei(i,Fqvec.dimension()) for i in range(Adim)])
        A = [ fromVec(x) for x in A ]

        Q = set()
        toQ = {}# map a -> a+A
        Qrep = {}#map a+A -> a (unique representative for a+A)
        for x in Fq:
            sig_check()
            xA = frozenset([x+a for a in A])
            Q.add(xA)
            toQ[x] = xA
            Qrep[xA] = x
        Q = set([ Qrep[xA] for xA in Q])

    else:
        Q = Fq

    #symplectic form has matrix
    #  0 I
    # -I 0
    M = []
    n2 = n//2
    for i in range(n):
        sig_check()
        row = [0]*n
        if i < n2:
            row[n2+i] = 1
        else:
            row[i-n2] = -1
        M.append(row)
    M = Matrix(Fq,M)

    vectors = list(V)
    for v in vectors:
        v.set_immutable()

    edges = []
    k = len(vectors)
    for i in range(k):
        x = vectors[i]
        for j in range(i+1,k):
            y = vectors[j]
            Bxy = x*M*y
            Byx = - Bxy
            for b in Q:
                sig_check()
                a = b + Bxy
                a2 = b + Byx
                if r != q:
                    a = Qrep[toQ[a]]
                    a2 = Qrep[toQ[a2]]

                edges.append( ( (a,x),(b,y) ) )
                edges.append( ( (a2,y),(b,x) ) )

    G = Graph(edges, format="list_of_edges")
    G.name("Symplectic antipodal %d cover of K_{%d^%d}"%(r,q,n))
    return G
```

A.22 Coset Graph of Linear Code

```
def coset_graph(const int q, C_basis, U_basis=None, n=None :
    r"""
    Computes the coset graph Γ(C) where C = Span(C_basis)

    The elements of C_basis are vectors over (F_q)^n.
    U_basis must span the complement of C

    If n or U_basis are not given, they will be deduced from C_basis.
    So if n or U_basis are not given, then C_basis should not be
        empty.
    """
    if n == None:
        n = len(C_basis[0])# dim V
    F = GF(q) #base field
    lambdas = [ x for x in F if x != 0 ]#non-zero elements of F

    def e(const int i):
        v = [0]*n
        v[i-1] = 1
        return vector(F,v,immutable=True)

    V = VectorSpace(F,n)

    if U_basis is None:
        C = V.span(C_basis)
        Q = V.quotient(C)
        lift = Q.lift_map()#Q -> V
        U_basis = [ lift(v) for v in Q.basis()]

    U = V.span(U_basis)
    vertices = list(U)

    # build our matrix A
    A = U_basis.copy()
    for c in C_basis:
        A.append(c)

    A = Matrix(F,A)
    A = A.transpose()
    Ainv = A.inverse()

    Pei = [] #list of P(e_i)
    for i in range(n+1):
        ei = e(i)
        if ei in U:
            Pei.append(ei)
        else:
            a = Ainv * ei
            # get zero vector and sum a[i]u_i to it
            v = vector(F,[0]*n)
            for i in range(len(U_basis)):
                v += a[i]*U_basis[i]
            v.set_immutable()
            Pei.append(v)

    lPei = [ l*u for l in lambdas for u in Pei]

    edges = []
    for v in vertices:
        for u in lPei:
            w = v + u
            edges.append( (v, w) )

    G = Graph(edges, format='list_of_edges')
    return G
```

A.23 Kasami codes

```
def extended_Kasami_code(const int s, const int t):
    r"""
    Returns the extended Kasami code with parameters (s,t)
    """
    F2 = GF(2)
    V = VectorSpace(F2, s)
    elemsFs = [x for x in GF(s)]

    #we ensure that 0 is the first element of elemsFs
    if not elemsFs[0].is_zero():
        for i in range(s):
            if elemsFs[i].is_zero:
                a = elemsFs[0]
                elemsFs[0] = elemsFs[i]
                elemsFs[i] = a
                break

    FsToInt = { x : i for i,x in enumerate(elemsFs)}
    elemsFsT = [x**(t+1) for x in elemsFs]
    FsTToInt = { x: i for i,x in enumerate(elemsFsT)}

    e1 = [0]*s
    e1[0] = 1
    e1 = vector(F2,e1,immutable=True)

    W1_basis = []
    for i in range(s-1):
        v = [0]*s
        v[i] = 1
        v[s-1] = 1
        W1_basis.append(v)
    W1 = V.span(W1_basis) #W1 satisfies \sum v[i] = 0

    W2_basis = set([e1])#not really a basis...
    for i in range(1,s):#avoid x = 0
        x = elemsFs[i]
        for j in range(i+1,s):
            y = elemsFs[j]
            v = [0]*s
            v[i] = 1
            v[j] = 1
            v[ FsToInt[(x+y)] ] = 1
            v = vector(F2,v,immutable=True)
            W2_basis.add(v)
    W2 = V.span(W2_basis) #W2 satisfies \sum v[i]elemsFs[i] = 0

    W3_basis = set([e1]) #again not really a basis
    for i in range(1,s): #avoid x = 0^(t+1) = 0
        x = elemsFsT[i]
        for j in range(i+1,s):
            y = elemsFsT[j]
            v = [0]*s
            v[i] = 1
            v[j] = 1
            v[ FsTToInt[(x+y)] ] = 1
            v=vector(F2,v,immutable=True)
            W3_basis.add(v)
    W3 = V.span(W3_basis)

    W = W2.intersection(W3)
    codebook = W.intersection(W1)
    return codebook


def Kasami_code(const int s, const int t):
    r"""
    Return the Kasami code with parameters (s,t)
    """
    C = extended_Kasami_code(s,t)
    codebook = [v[1:] for v in C.basis()]
    V = VectorSpace(GF(2),s-1)

    return V.span(codebook)
```

A.24 AB graph

```
def AB_graph(const int n):
    r"""
    Graph using almost bent functions on $(\mathbb{F}_q)^n$

    At the moment only odd n are implemented
    """
    if n%2 == 0:
        raise ValueError("no known AB function for even n")

    Fq = GF(2**n)
    f = { x : x**3 for x in Fq }#AB function

    vectors = [x for x in Fq]
    edges = []
    for i,x in enumerate(vectors):
        for y in vectors[i+1:]:
            for a in vectors:
                sig_check()
                b = a + f[x+y]
                edges.append(( (x,a),(y,b) ))
                edges.append(( (y,a),(x,b) ))

    G = Graph(edges,format="list_of_edges")
    return G
```

A.25 Example of sporadic graph

```python
def cocliques_HoffmanSingleton():
    D = GraphGenerators.HoffmanSingletonGraph()
    DC = D.complement()

    cocliques = DC.cliques_maximum()#100 of this

    edges = []
    for i in range(100):
        sC = frozenset(cocliques[i])
        for j in range(i+1,100):
            if len(sC.intersection(cocliques[j])) == 8:
                sC2 = frozenset(cocliques[j])
                edges.append( (sC,sC2) )

    G = Graph(edges,format="list_of_edges")
    return G
```

A.26 Example of selection function

```python
def is_from_square_BIBD(list arr):
    r"""
    Returns (v,k) s.t. graph_from_BIBD(v,k) has the correct
    intersection array; False if such pair doesn't exist
    """
    if len(arr) != 6: return False
    k = arr[0]
    l = arr[4]
    if l == 0 or (k*(k-1))%l != 0: return False
    v = (k*(k-1))//l +1

    if k <= 2: return False #trivial cases
    if v == k: return False #diameter 2
    #this will force v >= 4 as there is no BIBD with v<k

    if arr != [k, k-1, k-l, 1,l,k]:
        return False

    if Sage_Designs.balanced_incomplete_block_design(v, k, lmbd=l,
         existence=True) is not True:
        return False

    return (v,k)
```