AMSI SUMMER RESEARCH SCHOLARSHIPS 2023–24

SET YOUR SIGHTS ON RESEARCH THIS SUMMER

Low-diameter networks for applications on high performance computing and communication networks

Matthew Cochran

Supervised by Dr Guillermo Pineda-Villavicencio

Deakin University

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

Abstract

This report presents various techniques for the construction of large graphs with low degree and diameter, in an investigation of a graph-theoretic optimisation problem known as *the degree-diameter problem*. The techniques include both constructions of graphs such as the de Bruijn graph, Kautz graph, and Brown graph, as well as functions of graphs including the Kronecker product, star product, vertex duplication, and voltage assignment. We discuss the theory of these techniques along with their implementations in the mathematical software system SageMath. The implementation of these techniques will be used to programatically construct the currently-known low-diameter networks, and provides computational groundwork to improve on the best known constructions.

1 Introduction

Scientific and AI applications are increasingly demanding more resources; namely processors, memory and accelerators. These applications require multi-node configurations integrated with some type of interconnection network. The performance of these applications is influenced by how the processing nodes are connected, which is defined as the network topology. Key parameters of the network topology are the number of nodes (or *vertices*), and the maximum distance between these nodes. The *degree-diameter problem* is one way of formally characterising the constructions of graphs (or networks) that, for a given node *degree* (number of neighbouring vertices) and diameter (maximum number of edges between any two vertices), maximise the number of nodes in the graphs. This research explores techniques to build low diameter graphs, with emphasis on diameter three, that maximise the degree-diameter problem for degrees that are representative of current and foreseeable network technology. An upper bound for the number of nodes of a network is given by the *Moore bound*, $M(deg, diam)$ (whose value is approximately $(deg-1)^{diam}$); see [3] and [4, p. 5,22]. Our current knowledge on lower bounds for the optimal size $N(deg, diam)$ of a network with maximum degree deg and diameter $diam$ can be found in the form of tables of largest known graphs in [3]. For diameter 2, Brown [4] constructed a graph of maximum degree deg and $1 + (deg-1) + (deg-1)^2$ nodes, for each positive integer deg such that $deg-1$ is a prime power. The *Brown graph* ensures that $N(deg, 2)$ is asymptotically equivalent to the Moore bound. We can get large graphs of diameter two whose number of nodes is more than 95% of the corresponding Moore bound. The *MMS graphs* [4] are another good construction for diameter two. For diameter three, Delorme [4], seemingly inspired by the Brown graphs, produced two important constructions, the *Kronecker product* and the *star product* of graphs. These two constructions already ensure that $N(deg, 3)$ is asymptotically equivalent to the corresponding Moore graph. It was also noted by Delorme that, combining these constructions with the *Paley graph*, we can obtain graphs of diameter three whose number of nodes are more 50% of the Moore bound. Constructions of large graphs have found important applications in the construction of network topologies. See, for instance, the Slimfly network topology of diameter two [1], which is based on the MMS graphs, and the Bundlefly network of diameter three[2], which is a star product of MMS graphs and Payley graphs. In this

project, we will apply the most successful approaches, which are borrowed from geometry and algebra. These include the voltage assignment technique [4], polarity graphs, the Kronecker product [4], and the star product of graphs [4]. A background of the necessary mathematical concepts is provided in the appendix.

2 Statement of Authorship

The algorithms and code included in this report were devised and implemented by Matthew Cochran, based on the techniques and concepts presented in the literature and the guidance of Dr Guillermo Pineda-Villavicencio. The implementations were written using SageMath, and leverage existing SageMath libraries.

3 Techniques for the constructions of large graphs with low degree and diameter

The following definitions are adapted from [5] unless otherwise stated.

3.1 de Bruijn Graph

The *de Bruijn* graph of type (t, k) has its vertex set formed by all sequences of length k, the entries of which are taken from a fixed alphabet A consisting of t distinct letters. Two vertices $u = (u_1, u_2, \ldots, u_k)$ and $v = (v_1, v_2, \ldots, v_k)$ are joined by an edge if either $u_i = v_i + 1$ or $u_i + 1 = v_i$, for $1 \leq i \leq k - 1$.

The generation of a de Bruijn graph with parameters (t, k) can therefore be described according to the following algorithm:

Algorithm 1 de Bruijn Graph

```
alphabet ← {0, 1, 2, ..., t − 1}
V ← {v ∈ alphabet^k}
E ← ∅
for all u = [u_1, u_2, ..., u_k] ∈ V do
    for all v = [v_1, v_2, ..., v_k] ∈ V do
        for all a ∈ alphabet do
            if v = [a, u_1, u_2, ..., u_{k−1}] or v = [u_2, u_3, ..., u_k, a] then
                E ← E ∪ {(u, v)}
            end if
        end for
    end for
end for
```

I have implemented this in SageMath as follows:

```
from itertools import product

def de_bruijn_graph(t: int, k: int) -> Graph:
    alphabet = [i for i in range(t)]
    vertices = product(alphabet, repeat=k)

    g = Graph(loops=True)
    for vertex in list(vertices):
        g.add_vertex(vertex)
    for u in g.vertices():
        for v in g.vertices():
            for letter in alphabet:
                if (letter,) + u[:-1] == v or u[1:] + (
                    letter,
                ) == v:
                    g.add_edge(u, v)

    return g
```

If $t \geq 3$ and $k \geq 3$, the de Bruijn graph of type (t, k) has order t^k, diameter k, and maximum degree $2t$. For any D and even Δ, these graphs give the lower bound

$$N_{\Delta,D} \geq \left(\frac{\Delta}{2}\right)^D$$

Accordingly, I can validate my implementation by executing the following code:

```
t, k = 3, 3
g = de_bruijn_graph(t, k)
assert g.order() == t ** k
assert g.diameter() == k
assert max([g.degree(v) for v in g.vertices()]) == 2 * t
```

Naturally, the above and forthcoming validations hold for all valid input to the graph construction.

SageMath also has a built-in function to generate de Bruijn graphs, which has been parameterised by degree and diameter here.

```
def de_bruijn(degree: int, diameter: int) -> Graph:
    if degree % 2 != 0:
        raise ValueError("Degree must be even.")
    alphabet_size = degree // 2
```

```
    word_length = diameter
    return digraphs.DeBruijn(alphabet_size, word_length).to_undirected()
```

3.2 Kautz Graph

The *Kautz* graph of type (t, k) is an induced subgraph of the de Bruijn graph of type (t, k). The Kautz graph of type (t, k) is obtained by deleting words (represented by vertices) with two consecutive identical letters in the de Bruijn graph of type (t, k).

Since the Kautz graph is an induced subgraph of the de Bruijn graph, the algorithm to generate a Kautz graph of type (t, k) can be described, with reference to the previous algorithm, as follows:

Algorithm 2 Kautz Graph

$G \leftarrow$ de_bruijn(t, k)
for all $v = [v_1, v_2, \ldots, v_k] \in V(G)$ **do**
  if $\exists i \in \{1, 2, \ldots, k-1\}$ such that $v_i = v_{i+1}$ **then**
    $G \leftarrow G - v$
  end if
end for
return G

My resulting SageMath code is as follows:

```
def kautz_graph(t, k):
    g = de_bruijn(t, k)
    for vertex in g.vertices():
        if any([vertex[i] == vertex[i + 1] for i in range(k - 1)]):
            g.delete_vertex(vertex)
return g
```

If $t \geq 3$ and $k \geq 3$, the Kautz graph of type (t, k) has order $t(t-1)^{k-1}$, diameter k, and maximum degree $2t - 2$. Thus, for any D and even Δ, such graphs improve the bound given in (3.2), and consequently, we have

$$N_{\Delta,D} \geq \left(\frac{\Delta}{2}\right)^D + \left(\frac{\Delta}{2}\right)^{D-1}$$

Thus, I can validate my implementation by executing the following code:

```
t, k = 3, 3
g = kautz_graph(t, k)
assert g.order() == t * (t - 1) ** (k - 1)
assert g.diameter() == k
assert max([g.degree(v) for v in g.vertices()]) == 2 * t - 2
```

Again, SageMath has a built-in function to generate Kautz graphs, which I have parameterised by degree and diameter.

```
def kautz(degree: int, diameter: int) -> Graph:
    return digraphs.Kautz(degree // 2, diameter).to_undirected()
```

3.3 Brown Graph

The *Brown* graph is a polarity graph of a projective plane of order $\Delta - 1$, whose polarity has Δ absolute points.

The Brown graph can be constructed as follows: Let V be a three-dimensional vector space over the finite field $\mathbb{F}$ of order $\Delta - 1$. The vertex set of Γ is formed by a non-zero vector from each of the one-dimensional subspaces of V, and the edge set is formed by edges with orthogonal endvertices.

Implemented in SageMath, I have constructed the Brown graph according to the above description as follows:

```
def brown_graph(degree: int) -> Graph:
    f = FiniteField(degree - 1)
    g = Graph()
    vertices = ProjectiveSpace(2, f).rational_points()
    g.add_vertices(vertices)
    for v in g.vertices():
        for u in g.vertices():
            if v == u: continue
            if vector(v) * vector(u) == 0:
                g.add_edge(v, u)
    return g
```

The Brown graph has diameter 2, Δ vertices of degree $\Delta - 1$ (the absolute points of the polarity) and $(\Delta - 1)^2$ vertices of degree Δ.

Therefore, I validate my implementation by executing the following code:

```
degree = 5
g = brown_graph(degree)
assert g.diameter() == 2
assert g.order() == (degree - 1) ** 2 + degree
assert len([v for v in g.vertices() if g.degree(v) == degree - 1]) == degree
assert len([v for v in g.vertices() if g.degree(v) == degree]) == (degree - 1) ** 2
```

3.4 Generalised Polygons

The current best known constructions for $D = 2$, 3 and 5 are polarity graphs of the corresponding generalised polygons. Here, we will focus on the construction of the incidence graph of a generalised polygon, as it is used

later along with the Kronecker product.

To construct the incidence graphs of generalised polygons in SageMath, we leverage GAP (Groups, Algorithms, and Programming), a system for discrete computational algebra, for which SageMath provides an interface. The lines used to construct the generalised polygons were sourced from [7].

```
generalised_polygon = libgap.function_factory('GeneralisedPolygonByBlocks')
incidence_graph = libgap.function_factory('IncidenceGraph')


def generalised_polygon_incidence_graph(blocks: List[List[int]]) -> Graph:
    gen_poly = generalised_polygon(blocks)
    gap_g = incidence_graph(gen_poly).sage()

    vertices = gap_g["representatives"]
    adjacencies = gap_g["adjacencies"]
    adj_dict = {vertices[i]: adjacencies[i] for i in range(len(vertices))}
    g = Graph(adj_dict)
    return g
```

Note that the above depends on the GAP package, `FinInG`, being installed.

3.5 Paley Graph

The *Paley* graph is a graph of diameter 2, used by Delorme in his construction of large graphs of diameter 6. It is constructed as follows: Given the finite field $\mathbb{F}$ of order q such that $q \equiv 1 \pmod 4$, the Paley graph of order q is a graph whose vertex set is formed by the elements of $\mathbb{F}$, and two vertices are adjacent if, and only if, their difference is a non-zero square in $\mathbb{F}$.

The Paley graph is available in SageMath as a function in the `graphs` module, parameterised by order.

```
def paley(order):
    g = graphs.PaleyGraph(order)
    return g
```

3.6 Kronecker Product

The *Kronecker product* is a function of two bipartite graphs Γ, with partite sets A and B and Γ' with partite sets A' and B', denoted by $\Gamma \otimes \Gamma'$. It has vertex set $(A \times A') \cup (B \times B')$ and $(a, a') \sim (b, b')$ if and only if $ab \in E(\Gamma)$ and $a'b' \in E(\Gamma')$.

The algorithm to generate the Kronecker product of two bipartite graphs can therefore be described as follows:

Algorithm 3 Kronecker Product

$\Gamma \leftarrow$ bipartite_graph(A, B, E)
$\Gamma' \leftarrow$ bipartite_graph(A', B', E')
$V \leftarrow (A \times A') \cup (B \times B')$
$E \leftarrow \emptyset$
for all $u = (a, a') \in V$ **do**
  for all $v = (b, b') \in V$ **do**
    if $ab \in E(\Gamma)$ and $a'b' \in E(\Gamma')$ **then**
      $E \leftarrow E \cup \{(u, v)\}$
    end if
  end for
end for

My resulting SageMath code is as follows:

```
from itertools import product

def kronecker(g1: Graph, g2: Graph) -> Graph:
    try:
        g1 = BipartiteGraph(g1)
    except Exception:
        raise ValueError("Graph 1 is not bipartite.")
    try:
        g2 = BipartiteGraph(g2)
    except Exception:
        raise ValueError("Graph 2 is not bipartite.")
    a = list(product(g1.left, g2.left))
    b = list(product(g1.right, g2.right))
    g = Graph()
    for v in (a + b):
        g.add_vertices(a + b)
    for v1 in g.vertices():
        for v2 in g.vertices():
            if g1.has_edge(v1[0], v2[0]) and g2.has_edge(v1[1], v2[1]):
                g.add_edge(v1, v2)

    return g
```

The graph which is created by applying the Kronecker product is bipartite, has diameter $\max\{D(\Gamma), D(\Gamma')\}$,

order $|A||A'| + |B||B'|$, and maximum degree $\max\{\Delta_A\Delta_{A'}, \Delta_B\Delta_{B'}\}$, where Δ_X denotes the maximum degree of the vertices in X.

I can verify this as follows:

```
a = randint(2, 5)
b = randint(2, 5)

g1 = BipartiteGraph(graphs.RandomBipartite(a, b, 1))
g2 = BipartiteGraph(graphs.RandomBipartite(a, b, 1))

g = BipartiteGraph(kronecker(g1, g2))

assert g.diameter() == max(g1.diameter(), g2.diameter())
assert g.order() == len(g1.left) * len(g2.left) + len(g1.right) * len(g2.right)
assert max([g.degree(v) for v in g.vertices()]) == max([len(g1.left) * len(g2.left), le
```

Further, the Kronecker product of the incidence graph of a generalised quadrangle of order $\Delta - 1$, $\Delta - 1$ a prime power, and its *opposite*, has maximum degree Δ^2, diameter 4, and order $2(1+(\Delta-1)+(\Delta-1)^2+(\Delta-1)^3)^2$.

To verify the above, we also define the opposite of a bipartite graph $\Gamma = (A \cup B, E)$ as $(B \cup A, E)$. I have implemented this in SageMath as follows:

```
def opposite(g: BipartiteGraph) -> BipartiteGraph:
    if not isinstance(g, BipartiteGraph):
        try:
            g = BipartiteGraph(g)
        except Exception:
            raise ValueError("Input graph is not bipartite.")

    opposite = BipartiteGraph()
    for edge in g.edges():
        opposite.add_edge((edge[1], edge[0]))
    return opposite
```

With this, we can now verify the above result. Note that in the following code, `generalised_quadrangle` is a function first fetches the blocks of the generalised quadrangle from a file, and then constructs the incidence graph of the generalised quadrangle using the aforementioned `generalised_polygon_incidence_graph` function.

```
delta = 4
gq = generalised_quadrangle(delta - 1)
op = opposite(gq)
```

```
    g = kronecker(gq, op)

    assert maximum_degree(g) == delta ** 2
    assert g.diameter() == 4
    assert g.order() == 2 * (1 + (delta - 1) + (delta - 1) ** 2 + (delta - 1) ** 3)
```

3.7 Star Product

The *star product* of two graphs Λ and Γ, denoted by $\Lambda * \Gamma$, results in graphs of greater order than de Bruijn and Kautz graphs for diameter 4 and 6[8]. They are constructed by first choosing an *orientation* of the graph arbitrarily; that is to say, assigning a direction to each of the undirected edges of Λ. $A(\Lambda)$ then denotes the set of all the arcs of Λ. For each arc $(x, y) \in A(\Lambda)$, choose a bijection $\omega(x, y)$ on the set $V(\Gamma)$. Then $V(\Lambda * \Gamma) = V(\Lambda) \times V(\Gamma)$ and

$$E(\Lambda * \Gamma) = \{(u, x)(v, y) | \text{either } u = v \text{ and } xy \in E(\Gamma) \text{ or } (u, v) \in A(\Lambda) \text{ and } y = \omega(u, v)(x)\}.$$

The resulting graph has diameter at most $D(\Lambda) + D(\Gamma)$, order $|\Lambda| \times |\Gamma|$, and maximum degree $\Delta(\Lambda) + \Delta(\Gamma)$. A good choice of the bijections $\omega(x, y)$ can result in a diameter smaller than $D(\Lambda) + D(\Gamma)$.

The algorithm to generate the star product of two bipartite graphs can therefore be described as follows:

Algorithm 4 Star Product

$\Gamma \leftarrow$ bipartite_graph(A, B, E)
$\Gamma' \leftarrow$ bipartite_graph(A', B', E')
$V \leftarrow (A \times A') \cup (B \times B')$
$E \leftarrow \emptyset$
for all $v = (a, a') \in V$ **do**
  for all $u = (b, b') \in V$ **do**
    if $ab \in E(\Gamma)$ and $a'b' \in E(\Gamma')$ or $a = b$ and $a'b' \in E(\Gamma')$ or $a'b' \in E(\Gamma')$ and $a = b'$ **then**
      $E \leftarrow E \cup \{(u, v)\}$
    end if
  end for
end for

My resulting SageMath code is as follows:

```
def star_product(g1: Graph, g2: Graph, b: Dict) -> Graph:
    g = Graph()
    g.add_vertices(list(product(g1.vertices(), g2.vertices())))

    for u in g.vertices():
```

```
        for v in g.vertices():
            if u == v: continue
                continue
            if (u[0] == v[0] and u[1] in g2.neighbors(v[1])) or (u[0] in g1.neighbors(v[0]
                g.add_edge(u, v)
    return g
```

3.8 Vertex Duplication

Given a (Δ, D)-graph Γ, the technique of *vertex duplication* refers selecting a vertex x of Γ and adding a new vertex x' (the duplicate of x) such that $N(x') = N(x) \cup x$. The resulting graph has maximum degree $\Delta + 1$ and diameter D.

Algorithm 5 Vertex Duplication

$\Gamma \leftarrow$ graph
$v \leftarrow$ vertex
$v' \leftarrow$v.duplicate
for all $u \in N(v)$ **do**
  add_edge(v', u)
end for
add_edge(v', v)
return Γ

My resulting SageMath code is as follows (note: this implementation chooses a random vertex to duplicate by default):

```
def vertex_duplication(g: Graph, v = None) -> Graph:
    if v and v not in g.vertices():
        raise ValueError("Vertex must be in graph.")

    if not v:
        v = g.random_vertex()

    new_vertex = g.add_vertex()
    for u in g.neighbors(v):
        g.add_edge(new_vertex, u)

    g.add_edge(new_vertex ,v)
```

```
    return g
```

3.9 Voltage Assignment

The technique of *voltage assignment* has been used in the construction of graphs with maximum degree $3 \leq Delta \leq 16$ and diameter $2 \leq D \leq 10$ [9]. The following definitions are taken from [9].

Let Γ be a finite, undirected graph, possibly with loops and multiple edges. We also allow *semi-edges*, which are edges with just one end-vertex and with the other end free. To facilitate the description of voltages, we think of the (undirected) edges of Γ that are not semi-edges as pairs of oppositely directed arcs. A semi-edge admits, by definition, just one direction (into its unique end vertex). So, we obtain the digraph Λ. The number of elements in the set $A(\Lambda)$ of all arcs of Λ is therefore twice the number of all edges of Γ minus the number of semi-edges. If e is an arc, then e^{-1} denotes the arc reverse to e; in the case of a semi-edge we set $e^{-1} = e$ by convention.

Let G be a finite group. A mapping $\alpha : A(\Lambda) \to G$ is called a voltage assignment if $\alpha(e^{-1}) = (\alpha(e))^{-1}$ for every arc $e \in A(\Lambda)$. Thus, a voltage assignment sends a pair of mutually reverse arcs onto a pair of mutually inverse elements of the group. Note that if e is a semi-edge, then the voltage condition means that $\alpha(e)$ has order 2 in G. The pair (Λ, α) is the voltage graph, which determines a lift Λ^α of Λ as follows. The vertex set and the arc set of the lift are $V(\Lambda^\alpha) = V(\Lambda) \times G$ and $A(\Lambda^\alpha) = A(\Lambda) \times G$, respectively. In the lift, (e, g) is an arc from the vertex (u, g) to the vertex (v, h) if, and only if, e is an arc from u to v in Λ and $h = g\alpha(e)$. The lift itself is considered to be undirected, since (e, g) and $(e^{-1}, g\alpha(e))$ form a pair of mutually reverse arcs and therefore give rise to an undirected edge of Λ^α.

I have implemented the following code in SageMath to obtain a voltage assignment of a graph given a group:

```
def voltage_assignment(graph, group) -> Dict:
    members = list(group)
    v_a = {}
    for e in graph.edges():
        member = members.pop()
        edge = (e[0], e[1])
        inverse_edge = (e[1], e[0])

        if not v_a.get(edge):
            v_a[edge] = []
        v_a[edge].append(member)

        if member.inverse() in members:
            if not v_a.get(inverse_edge):
                v_a[inverse_edge] = []
```

```
            v_a[inverse_edge].append(member.inverse())
            members.remove(member.inverse())

    return v_a
```

With this mapping, we can apply the voltage assignment to the graph to obtain the voltage graph. I have also created a function which does this in SageMath:

```
def lift(graph: Graph, voltage_assignment: Dict, group) -> Graph:
    voltage_graph = Graph()
    vertices = []
    for v in graph.vertices():
        for h in group:
            vertices.append((v, h))
    voltage_graph.add_vertices(vertices)

    edges = []
    for e in graph.edges():
        for h in group:
            edges.append((e[0], e[1], h))

    for e in edges:
        u, v, g = e
        if v in graph.neighbors(u):
            for h in group:
                for alpha in voltage_assignment[(u, v)]:
                    if h == g * voltage_assignment[(u, v)][0]:
                        voltage_graph.add_edge((u, g), (v, h))
    return voltage_graph
```

With these two functions, we are able to replicate the example in [9] in construction of the Petersen graph using voltage assignment. Note that here, we manually construct the voltage assignment, as the above method will find a different, but also valid, voltage assignment.

```
g = Graph(loops=True, multiedges=True)
g.add_vertices(["u", "v"])
g.add_edge("u", "u")
g.add_edge("v", "v")
g.add_edge("u", "v")
```

```
z5 = CyclicPermutationGroup(5)

voltage_assignment = {
    ("u", "v"): [z5[0]],
    ("v", "u"): [z5[0]],
    ("u", "u"): [z5[1], z5[4]],
    ("v", "v"): [z5[2], z5[3]],
}

voltage_graph = lift(g.to_directed(), voltage_assignment, z5)

assert voltage_graph.is_isomorphic(graphs.PetersenGraph())
```

4 Discussion and Conclusion

The techniques for the construction of large graphs with low degree and diameter have been presented, along with their implementations in SageMath. The techniques include both constructions of graphs such as the de Bruijn graph, Kautz graph, and Brown graph, as well as functions of graphs including the Kronecker product, star product, and vertex duplication. The implementations have been validated by verifying that the constructed graphs satisfy the properties of the respective techniques.

In future, these implementations may be used to recreate and improve on the best known constructions of large graphs with low degree and diameter, and to explore new constructions.

5 Acknowledgements

I would like to express my sincere gratitude for the support and guidance provided by Dr Guillermo Pineda-Villavicencio in the undertaking of this project, and to the Australian Mathematical Sciences Institute for the opportunity to engage in this research.

References

[1] Besta, M, & Hoefler, T 2014, 'Slim Fly: A cost effective low-diameter network topology', *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE Press, pp. 348-359.

[2] Lei, F, Dong, D, Liao, X & Duato, J 2020, 'Bundlefly: A low-diameter topology for multicore fiber', *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, pp. 1-10.

[3] Loz, E, Pérez-Rosés, H & Pineda-Villavicencio, G 2008. *The degree/diameter problem*, Combinatorics Wiki, accessed 10 December 2023. `http://combinatoricswiki.org/wiki/The_Degree/Diameter_Problem`

[4] Miller, M, & Širáň, J 2013. 'Moore graphs and beyond: A survey of the degree/diameter problem', *The Electronic Journal of Combinatorics*, DS14, pp. 1-61.

[5] Pineda-Villavicencio, G 2009, 'Topology of Interconnection Networks with Given Degree and Diameter', PhD thesis, *University of Ballarat, Ballarat, Victoria, Australia.*

[6] Tits, J 1974, *Buildings of Spherical Type and Finite BN-pairs*, 1st edn, Springer-Verlag, Heidelberg, Berlin.

[7] Vidali, J. (2003) *Generalised Polygons of Small Order*, Janoš Vidali, accessed 29 December 2023. `https://jaanos.github.io/tables/moorhouse/pub/genpoly/`

[8] Bermond, J C, Delorme, C & Farhi, G 1982, 'Large graphs with given degree and diameter III', *North-Holland Math. Studies*, vol. 62, pp. 23-31.

[9] Loz, E & Pineda-Villavicencio, G 2010, 'New Benchmarks for Large-Scale Networks with Given Maximum Degree and Diameter', *The Computer Journal*, vol. 53, pp. 1092 - 1105

6 Appendix

The following definitions are taken from [5].

6.1 Groups

Let X be a non-empty set, together with an associative binary operation $\cdot$. Then $\Omega = (X, \cdot)$ is a group if:

1. There exists an element $id \in \Omega$, called the *identity*, such that for all $\omega \in \Omega$, $id \cdot \omega = \omega \cdot id = \omega$.
2. For each $\omega \in \Omega$ there exists an element $\omega^{-1} \in \Omega$, called the inverse of ω, such that $\omega^{-1} \cdot \omega = \omega \cdot \omega^{-1} = id$.

If the operation $\cdot$ is commutative, we say that the group is commutative. Commutative groups are called *abelian*.

The order of a group Ω, denoted by $|\Omega|$, is the number of elements in Ω. The order of an element ω of a finite group Ω is the smallest positive integer n such that $\omega^n = id$, where id is the identity element of Ω. An element of order 2 is called an involution of Ω.

6.2 Graphs

Assume that $[X]_k$ denotes the set of all k-subsets of a set X. A *graph* Γ is a pair (V, E) of sets satisfying $E \subseteq [V]^2$, where $V \neq \emptyset$. The elements of V and E are called the *vertices* and *edges* of the graph Γ, respectively. A graph so defined is called *undirected*, where each edge is an unordered pair of vertices, and hence, has no direction.

The *vertex set* V of a graph Γ is denoted by $V(\Gamma)$, and the *edge set* of Γ by $E(\Gamma)$. In some instances for a vertex (or an edge), we may write $x \in \Gamma$ instead of $x \in V(\Gamma)$ (or $E(\Gamma)$). The number of vertices of Γ represents the *order* of Γ, denoted $|\Gamma|$. An edge $e = \{x, y\}$ may be written $e = xy$, xy, or $x \sim y$. For an edge $e = xy$, we say that x and y are the*end vertices* or *ends* of e, *adjacent* or *neighbours*, and *incident* with e. We also say that e is incident with x and y. Two edges $e \neq f$ are *adjacent* if they share an end vertex. If two vertices x and y are not adjacent, then we write $x \not\sim y$.

We denote by $E_\Gamma(X, Y)$ the set of edges in a graph Γ joining a vertex in X to a vertex in Y. For simplicity, instead of $E_\Gamma(\{x\}, Y)$, we write $E_\Gamma(x, Y)$, and instead of $E_\Gamma(X, X)$, we write $E_\Gamma(X)$. The set of all edges in $E(\Gamma)$ incident with a vertex x is denoted by $E_\Gamma(x)$.

Let Γ be a graph. The set of *neighbours* of a vertex x in Γ is denoted by $N_\Gamma(x)$. The set of neighbours of a vertex set X, $X \subseteq V(\Gamma)$, denoted by $N_\Gamma(X)$, can be defined as follows:

$$N_\Gamma(X) = \bigcup_{x \in X} N_\Gamma(x).$$

The *degree* of a vertex x is the number of edges incident with x, denoted by $d_\Gamma(x)$. The number $\Delta(\Gamma) = \max\{d_\Gamma(x) \,|\, x \in V(\Gamma)\}$ is the maximum degree of Γ, while the minimum degree over all the vertices of Γ is called the minimum degree of Γ.

In any of the aforementioned concepts, as elsewhere, the index referring to the underlying graph is dropped if the reference is clear. For instance, in such cases, we write $N(x)$ instead of $N_\Gamma(x)$.

Let Γ and Γ' be two graphs. If $V(\Gamma') \subseteq V(\Gamma)$ and $E(\Gamma') \subseteq E(\Gamma)$, then Γ' is a *subgraph* of Γ, denoted by $\Gamma' \subseteq \Gamma$. When we have $\Gamma' \subseteq \Gamma$ without equality, we say that Γ' is a *proper subgraph* of Γ, denoted by $\Gamma' \subset \Gamma$. If $\Gamma' \subseteq \Gamma$ and Γ' contains all the edges $xy \in E(\Gamma)$ with $x, y \in V' = V(\Gamma')$, then Γ' is an *induced subgraph* of Γ, denoted by $\Gamma[V']$.

Let Γ and Γ' be two graphs. The graphs Γ and Γ' are isomorphic, denoted by $\Gamma \cong \Gamma'$, if there is a bijection $f : V(\Gamma) \to V(\Gamma')$ such that $xy \in E(\Gamma)$ if, and only if, $f(x)f(y) \in E(\Gamma')$. The function f is called an *isomorphism*, and if $\Gamma = \Gamma'$, it is called an *automorphism*.

The set of automorphisms of a graph Γ forms a group under the operation of composition of functions. This group is called the *automorphism group* of Γ, denoted by $\mathrm{Aut}(\Gamma)$. The group $\mathrm{Aut}(\Gamma)$ is a *permutation group* on $V(\Gamma)$, which helps us to understand the symmetries in Γ. We say that Γ is *vertex-transitive* if $\mathrm{Aut}(\Gamma)$ acts transitively on $V(\Gamma)$, that is, for any two vertices u and v, there is an automorphism $\omega \in \mathrm{Aut}(\Gamma)$ such that $\omega(u) = v$. Analogously, we say that Γ is *edge-transitive* if, for any two edges e and f in $E(\Gamma)$, there is an automorphism $w \in \mathrm{Aut}(\Gamma)$ such that $w(e) = f$, where $w(e = \{x, y\}) = \{w(x), w(y)\}$.

Let Γ be a bipartite graph with partite sets V_1 and V_2. A *polarity* ω on Γ is an involution of $\mathrm{Aut}(\Gamma)$ that interchanges V_1 and V_2, that is, $\omega(V_1) = V_2$ and $\omega(V_2) = V_1$.

6.3 Generalised Polygons

Generalised polygons were introduced by Tits in 1974 [6]. They are related to incidence geometry, and their definition requires elements of geometry, such as the notions of *incidence structures* and *incidence graphs*.

An incidence structure is a triple $\Omega = (P, L, I)$, where $P \neq \emptyset$ is a set of points, $L \neq \emptyset$ is a set of lines, $P \cap L = \emptyset$, and $I \subseteq P \times L$ is a relation, called the incidence relation. Given a point p and a line l, if $(p, l) \in I$ then we say that p and l are incident.

Two points are collinear if they are incident with at least one common line. Two lines are concurrent if they share at least one point. Given an incidence structure Ω, if each line is incident with $s + 1$ points, and each point is incident with $t + 1$ lines, we say that Ω has order (s, t). If $s = t$ then Ω is said to have order s.

An incidence structure $\Omega = (P, L, I)$ is said to be finite if P and L are finite sets.

Let $\Omega = (P, L, I)$ be an incidence structure. The incidence graph Γ of Ω is the graph with vertex set $V(\Gamma) = P \cup L$, and the following adjacency relation: $(x, y) \in E(\Gamma) \leftrightarrow xIy$ or yIx for $x, y \in V(\Gamma)$.

A generalised D-gon is an incidence structure whose incidence graph is a bipartite graph of diameter D and girth $2D$. It is common to use standard names for small polygons, for instance, generalised quadrangle instead of generalised 4-gon.

A generalised triangle of order $s, s > 1$, is a *projective plane* of order s. To date projective planes of order s have been obtained only when s is a prime power.

Let $\Omega = (P, L, I)$ be an incidence structure with a polarity ω (a polarity of the incidence graph). The polarity graph, denoted by Γ_ω, of Ω with respect to ω is the graph with vertex set $V(\Gamma) = P$, and the following adjacency relation: $pp_1 \in E(\Gamma_\omega)$ if $p \neq p_1$ and $(p, \omega(p_1)) \in I$. We call a point p an absolute point of the polarity ω if $(p, \omega(p)) \in I$. The number of absolute points of ω is denoted by N_ω.