Ollscoil na Gaillimhe
University of Galway

Using Ant Colony Optimisation with Symmetry-Breaking on the Maximum Clique Problem

Sinéad Buggy
22322881
BSc Mathematics and Computing

Neesha Duggan
22477542
BSc Mathematics and Computing

Under the Supervision of Dr Jesse Lansdown
School of Mathematical and Statistical Sciences
University of Galway

13 March 2026

Contents

Declaration of Authorship

We declare that the work presented in this thesis is our own, unless otherwise stated. We maintain that all sources used have been properly acknowledged and cited.

Division of Labour

The majority of this project has been a combined effort between both students. The technique of symmetry-breaking was extensively discussed in the supervisor meetings. The implementation and testing of the SageMath code, as well as obtaining the results from the Ant-Clique algorithm, were performed by both parties.

The research and writing of sections 2.3, 4.1 - 4.5 were a joint effort. The remainder of the sections were researched and written as follows:

- Sinéad: sections 2.1, 3, 5, 6.
- Neesha: sections 1, 2.2, 4.6 - 4.8.

Acknowledgements

We would like to thank Dr Jesse Lansdown for his continued guidance and assistance over the course of this project. We thank Caoimhe Downey and Aoife Doyle for their support and encouragement. We also wish to acknowledge the work of Christine Solnon and Serge Fenet, whose open-source software, Ant-Clique, we used for this project.

1 Introduction

Graph theory plays an important role in many areas of mathematics and computer science. It has applications that range from network analysis to optimisation problems. To ensure clarity, we begin by introducing some definitions that are integral to this project. These are provided from Grinberg (2025) [1] and Wilson (1996) [2] Introduction to Graph theory books. A **graph** $\Gamma = (V, E)$, has a finite set of vertices $V(\Gamma)$ and set $E \subseteq V \times V$ of edges between vertices. This project only discusses **simple** and **undirected** graphs. That is, graphs where there are no loops or multiple edges between vertices, and each edge is bi-directional, meaning if there exists an edge $\{u, v\}$ between vertices u and v, then this edge is equivalent to $\{v, u\}$.

A classic problem in graph theory is to identify cliques within graphs, where a **clique** is a subset C of the set of vertices V in a graph Γ, such that each pair of distinct vertices in C are **adjacent** i.e. they are connected by an edge. A clique of size 1 is simply a vertex, whereas a clique of size 2 is two vertices that have an edge between them. Some examples of cliques in symmetric and random graphs are shown below in purple.

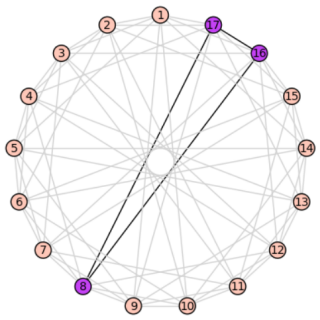

Figure 1: Clique of size 3 in the Paley graph

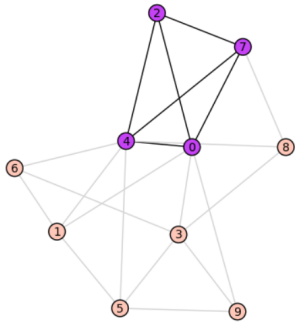

Figure 2: Clique of size 4 in a random graph

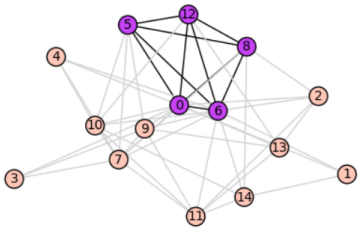

Figure 3: Clique of size 5 in a random graph

This definition naturally leads to two related concepts, maximal and maximum cliques. A clique is **maximal** if it cannot be extended into a larger clique by adding another adjacent vertex i.e. it is not a subset of a larger clique, and **maximum** if it is the largest possible clique in Γ. These concepts are explained in the House graph examples below.

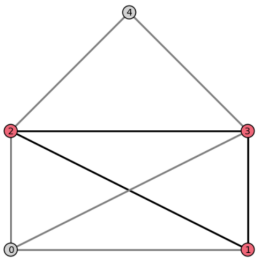

Figure 4: Non-Maximal Clique

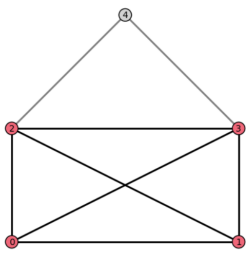

Figure 5: Maximal and Maximum Clique

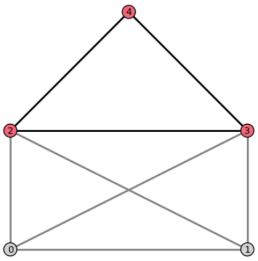

Figure 6: Maximal Clique

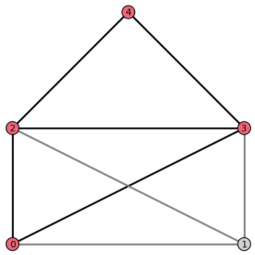

Figure 7: Not a Clique

The primary focus of this project is the **Maximum Clique Problem** (MCP) which is one of the most heavily studied problems in combinatorial optimisation, where the objective is to find a clique containing the largest possible number of vertices in a graph. The size of this largest possible clique is called the clique number of the graph, and is denoted by $\omega(\Gamma)$. Applications of the MCP are discussed in section 2.1, which include the

maximum coclique problem and the minimum vertex cover problem, as well as applications in coding theory and biological data analysis. The MCP is an extremely difficult task as it is NP-Hard, meaning that there is no known polynomial-time algorithm that can solve it. This complexity is analysed further in section 2.2.

A maximum clique must be maximal, though the converse is not necessarily true. Therefore, finding all maximal cliques of a graph can help to find a maximum clique. However, any graph $\Gamma = (V, E)$ can contain as many as $3^{|V|/3}$ maximal cliques [3]. Consequently, enumerating all maximal cliques should be avoided, as it quickly becomes computationally expensive for larger graphs. Various exact methods are known to "solve" the MCP, such as branch-and-bound and recursive backtracking algorithms [4]. These algorithms have an exponential time complexity and are therefore not well suited for larger graphs due to increased runtime and memory issues. In section 2.3, a brute force and greedy algorithm for finding cliques in graphs are examined but ultimately are found to be inefficient or unreliable, respectively. For these reasons, many approaches that aim to estimate the MCP in larger graphs rely on heuristic or metaheuristic methods to avoid an exhaustive search. These algorithms aim to balance computational efficiency with the ability to find high quality solutions, as explored in section 3.

The attention of this project is directed towards the Ant Colony Optimisation (ACO) metaheuristic, which is the topic of section 3.2. ACO is a Population-based metaheuristic, which works by keeping multiple candidate solutions. So, for symmetric graphs, many of these solutions may overlap or be equivalent to each other. The existence of these multiple suboptimal solutions slows the process of finding a maximum clique, because "from a computational point of view symmetries represent a difficulty for optimizations algorithms, because of the presence of a high number of equivalent suboptimal solutions, which tend to hide the optimum." [5].

This project utilises an ACO-based software for the MCP called Ant-Clique and introduces a symmetry-breaking algorithm to improve its performance on graphs with high symmetry. In section 4, the proposed symmetry-breaking algorithm is described. The original graph is split into a set of subgraphs in such a way that no structural information of the graph is lost, by only removing redundant symmetric solutions. This is performed using a concept called "starter sets".

Section 5 examines the Hamming graphs, a family of symmetric graphs that have applications in error correcting codes. These graphs are used as a case study to evaluate the performance of our symmetry-breaking algorithm in combination with the Ant-Clique algorithm, to estimate the maximum coclique in each graph, as the cocliques in the Hamming graphs are much larger than the cliques.

Finally, section 6 presents the conclusion of this project and discusses our results further. It summarises the main findings of the project and outlines potential directions for future research.

2 Maximum cliques of graphs

This section motivates why the MCP was chosen as the central problem of this project. It discusses related graph theory problems and how finding the maximum clique can be useful in other fields. It also illustrates why this problem is so difficult to solve, by examining the theory of NP-Hardness. We motivate the use of metaheuristics for this project by analysing the brute force and greedy algorithms and their limitations.

2.1 Applications of the MCP

The problem of finding the maximum clique in a graph has many applications, both in graph theory and in other fields.

2.1.1 Analogous Graph theory problems

The problem of finding the maximum coclique of a graph is very closely related to the MCP. A **coclique**, also known as an **independent set**, of Γ is a subset I of the set of vertices V in Γ, where all pairs of I are not adjacent to each other, i.e. no vertices in the subset are connected by an edge. The coclique number of a graph, denoted $\alpha(\Gamma)$, is the size of the maximum coclique in Γ.

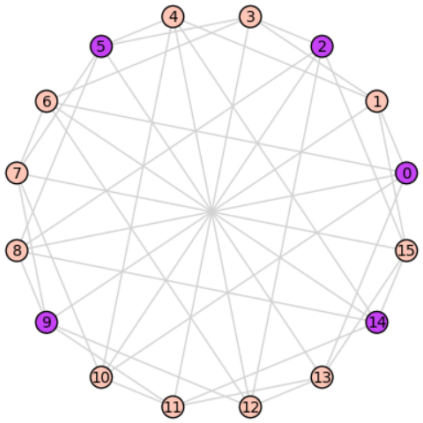

Figure 8: Maximum coclique in the Clebsch graph

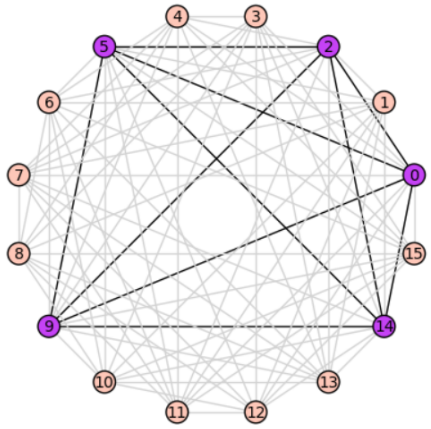

Figure 9: Maximum clique in the complement of the Clebsch graph

The **complement graph** of Γ, denoted $\overline{\Gamma}$, has the same set of vertices V, however two vertices are adjacent in $\overline{\Gamma}$ if and only if they are not adjacent in Γ. Note that a set of vertices that form a coclique in a graph Γ form a clique in its complement graph $\overline{\Gamma}$. Similarly, the maximum coclique of Γ is equivalent to the maximum clique of $\overline{\Gamma}$, as demonstrated in figures 8 and 9. This fact is used in our algorithm in section 4 to find both cliques and cocliques, since an algorithm designed to find cliques can be used to find cocliques by simply inputting the complement graph. When searching for cocliques, our implementation constructs the complement of the graph and computes the cliques of the graph $\overline{\Gamma}$.

Another application of the Maximum Coclique problem is the task of finding the Minimum Vertex Cover. A **vertex cover** of a graph Γ is a subset V' of the set of vertices in the graph such that every edge of the graph contains at least one vertex in V' [1]. The minimum Vertex Cover of a graph is the smallest possible such set V'.

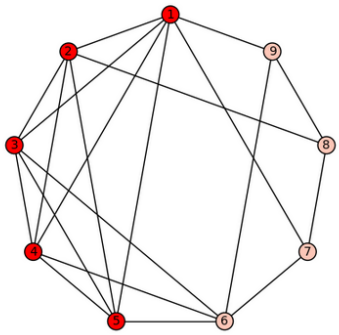

Figure 10: Clique in Γ

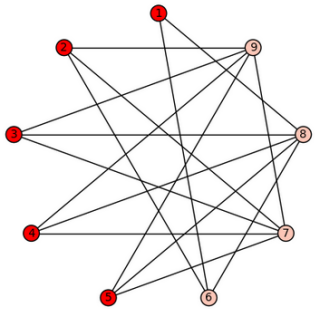

Figure 11: Clique in $\overline{\Gamma}$ (Equivalent to Coclique in Γ)

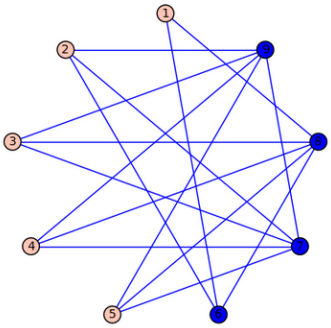

Figure 12: Vertex cover of $\overline{\Gamma}$

Proposition: If V' is a vertex cover, then $\Gamma \setminus V'$ is a coclique.
Proof by Contradiction: Assume $\Gamma \setminus V'$ is not a coclique. This implies that there are two vertices v_1 and v_2 in $\Gamma \setminus V'$ that are adjacent. Therefore, (v_1, v_2) is an edge with neither vertex included in V'. Hence V' is not a vertex cover. This a contradiction, so $\Gamma \setminus V'$ must be a coclique [4].

Hence, the MCP, the Maximum Coclique problem and the Minimum Vertex Cover problem are essentially the same problem, as they can be reduced into each other and are all NP-Hard. So, any algorithm that approximates one of these three problems, can be applied to the other two. Many problems in the field of computer science, such as job scheduling algorithms, can be reduced to the Maximum Coclique problem, so it is useful as a tool for simplifying other algorithms, and as a method of proving their complexity [4].

2.1.2 Applications in other fields

There are many disciplines where it is useful to represent data as an interconnected network, for example in the processing of biological data. Biological components such as genes, proteins or other types of molecules can be represented as vertices, with edges between these vertices indicating a relationship between two molecules [6]. The problem of finding cliques can also be applied to the analysis of social networks, where the vertices of a graph are individuals, and edges between two vertices indicate a relationship between those two people. Finding cliques in these graphs allows communities to be identified, which can, for example, be used by social media algorithms to tailor content to individuals with similar interests [7].

Cliques and cocliques have applications in other areas of mathematics, for example in coding theory. Error correcting codes are ways of encoding messages that are sent over an unreliable channel such that errors in transmission, such as deletion or transposition of bits, can be automatically corrected. For a vector u (a sequence of bits representing the message to be encoded), $F_e(u)$ denotes the set of vectors that can arise from an error e in the transmission of u (where e is a specific error, for example two transpositions or one deletion). A set of codewords is an e-correcting code if all e errors can be identified and corrected. This can only happen if there is no intersection between $F_e(u)$ and $F_e(v)$ for the distinct codewords u and v. Therefore, by mapping each codeword to a vertex in a graph, and adding an edge between two codewords u and v if and only if $F_e(u) \cap F_e(v) \neq \emptyset$, the problem of finding the largest e-correcting code can be reduced to the problem of finding the largest coclique in this graph [8].

2.2 History of the MCP and how it is NP-Hard

The graph theory term "clique" is believed to have been first formally used by Luce and Perry, 1949 [9]. Their motivation was to use matrices to model social cliques, which are groups of individuals that interact with one another and share common characteristics. As graph theory progressed, many researchers began implementing and studying algorithms to try and find a clique of a particular size or larger in a graph. This is framed as a decision problem (i.e. a problem that has a yes or no answer). Is there a clique of size J or larger in a graph $\Gamma = (V, E)$, where $J \leq |V|$? This problem is NP-Complete [10]. This is closely related to but not technically the same as the Maximum Clique Problem which we are investigating. While the decision version is NP-Complete, the optimisation version of the MCP is NP-Hard. Some background information is necessary to understand these terms.

There are two classes P and NP that are critical to the theory of NP-Completeness. The class P consists of all decision problems that can be solved in polynomial-time. A polynomial-time algorithm is one where the number of computational steps is bounded above by a polynomial function dependent on the size of the input, i.e. polynomial-time algorithms can be solved "quickly". The class NP consists of all decision problems that can be "verified" in polynomial-time [10] [4]. NP is short for non-deterministic polynomial-time, this is the class of algorithms where given a solution its accuracy can be verified in polynomial-time but an answer cannot necessarily be found in polynomial-time [4]. So, clearly $P \subseteq NP$ and a key unsolved computing problem is that of "P vs NP" with the goal to prove either $P = NP$ or $P \neq NP$.

A problem is NP-Hard if it is at least as hard as any problem in the class NP. Note that the problem does not have to be in the NP class itself. A problem is NP-Complete if it is both NP-Hard and also in the class NP. Thus our problem of finding an algorithm to solve the maximum clique of a graph is NP-Hard as there is no polynomial-time algorithm known to compute the maximum clique unless $P = NP$.

The concept of NP-Completeness was first established by Cook in 1971 [11]. He introduced the term "reduced". He said that for two problems where one can be reduced into the other, that there exists a transformation that maps instances of the first problem into corresponding instances in the second. Consequently, we can use this transformation to convert any algorithm that computes one problem into one that computes the other. This means that finding algorithms to solve these problems are using some of the same techniques, just modelled differently [10]. Cook also showed that one of the NP-Complete problems called Satisfiability is actually the mother of all NP-Complete problems and that every problem in NP can be reduced in polynomial-time to the Satisfiability problem, see Fig 13. Note that satisfiability can be reduced to the vertex cover problem mentioned in section 2.1.1, which as noted before can be reduced into the clique problem. This is the theory behind how solving for the maximum clique and maximum coclique are essentially the same problem as they can be reduced to each other in polynomial-time.

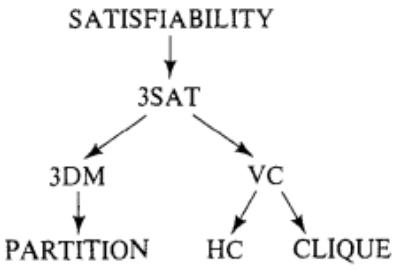

Figure 13: Satisfiability is the mother of all NP-Complete Problems [10]

Algorithm 1: Reducing finding the maximum coclique of a graph Γ into the maximum clique problem on $\overline{\Gamma}$ [4]

Data: Γ: Graph
n: size of coclique to find

Initialisation;
Coclique(Γ, n) :
  $\overline{\Gamma} \leftarrow$ complement of Γ;
  return *Clique(* $\overline{\Gamma}$*, n)*

The Second DIMACS Implementation Challenge, 1992-1993, introduced a list of common benchmark graphs. These benchmark graphs are still widely used today to evaluate performance and fairly compare MCP algorithms. This challenge was set to explore the NP-Hard problems of the MCP, Graph Colouring and Satisfiability. The graphs were chosen to be difficult for the algorithms to solve, to fully test their abilities [12]. However, the majority of this DIMACS challenge set are not symmetric, and so are not relevant for this project. The only DIMACS challenge set graphs included are Hamming10_4 and keller6, the results of which are included in section 6.2.

2.3 Basic approaches for finding the maximum clique

The two simplest and most intuitive methods to attempt to find the maximum clique in a graph are a brute force and greedy approach.

2.3.1 Brute force approach

Brute force is an exhaustive search with backtracking that will return all cliques in a graph so that the maximum clique can be extracted. Using brute force to find the maximum clique of a graph is extremely inefficient. For small graphs it works well, but as graphs get larger in size, it becomes infeasible and the algorithm simply will not terminate. Below is the pseudocode for the Brute Force algorithm with backtracking that returns one of the maximum cliques.

Algorithm 2: Brute Force algorithm

Data: Γ: Graph
Result: *maxClique* : maximum clique found by the algorithm

Initialisation;
stack $\leftarrow$ all vertices of Γ as individual lists, will store all cliques of Γ;
maxClique $\leftarrow$ empty list to store the maximum clique;
while *stack is not empty* **do**
  $s \leftarrow pop(stack)$;
  if $|s| > |maxClique|$ **then**
    $maxClique \leftarrow s$;
  end
  $neighbours \leftarrow$ adjacent vertices to all vertices in s;
  for x *in* $neighbours$ **do**
    $stack \leftarrow stack + s \cup \{x\}$;
  end
end
return *maxClique*

2.3.2 Greedy approach

Greedy algorithms start with one vertex and check the surrounding vertices, adding to the clique the "best" vertex that is connected to every vertex already included in the clique. The "best" vertex is quantified using some criterion, e.g. the vertex of highest degree. The starting point can either be random, or the best option according to the specified criterion. Choosing a random vertex means there is a lot of variation in the results, there is always a possibility of finding the maximum clique, but it is also possible to get very poor results if the starting point is in a very small clique. On the contrary, always starting with the vertex of highest degree creates a deterministic algorithm which returns the largest clique that includes this vertex, which may not necessarily be the maximum clique in the overall graph. Below is the pseudocode for the greedy algorithm.

Algorithm 3: Greedy algorithm

```
Data: Γ: Graph
Result: maxClique : maximum clique found by the algorithm
Initialisation;
vertices ← list of vertices of Γ;
start ← vertex of highest degree in Γ chosen as startpoint;
maxClique ← list to store the maximum clique;
best ← variable to store vertex of highest degree;
add start to maxClique;
while common neighbourhood of maxClique is not empty do
    for vertex in neighbourhood of maxClique do
        if deg(vertex) > deg(best) then
            best ← vertex
        end
    end
    maxClique ← maxClique + best
end
return maxClique
```

2.3.3 Finding a middle ground

The greedy algorithm has no backtracking, meaning it is highly dependent on the choice of starting vertex, but it is fast, because it only checks vertices in the common neighbourhood of the current clique, rather than constructing every possible clique like brute force does. For large graphs, brute force is accurate but too slow and computationally inefficient, whereas the greedy algorithm is fast but may not always return the maximum clique. This is our motivation for using metaheuristics, because they are a family of algorithms that seek to strike a balance between the comprehensive search of brute force and the efficiency of the greedy algorithm.

3 Metaheuristic algorithms

3.1 General form and types

Metaheuristics is a general term applied to algorithms that are used to find solutions to hard problems where an exhaustive search is not viable because of the size of the search space, and there is no obvious way to search the space efficiently [13]. The term search space means the set of all possible solutions to the problem at hand. So, in this project the search space refers to the graph we aim to find the maximum clique of. Metaheuristics seek to find optimal or near optimal solutions to these kinds of problems by incorporating randomness with local search techniques, to find a balance between efficiency and comprehensiveness. It is essentially trying to find a middle ground between the greedy algorithm and the brute force algorithm, where enough of the search space is explored to make it likely to find the optimal solution, but the algorithm is not wasting resources searching places which are unlikely to yield optimal solutions. For this reason, implementing metaheuristic algorithms requires a lot of experimentation, because the optimal balance between these opposing goals of speed and accuracy will differ for every problem. This is why metaheuristics should only be used for problems where the search space is too large to be searched systematically to yield the optimal solution every time [13].

Therefore, metaheuristics are a good option for the MCP, because for large graphs, the search space is too extensive, and there is no exact method for finding the maximum clique in a graph that can run in a feasible time frame. (In optimisation problems, "exact" methods are algorithms that find the optimal solution every time).

Metaheuristics is a term used to describe a broad family of algorithms that seem very different in structure, but have certain traits in common. They start with one or more candidate solutions, where a candidate solution is simply a solution that could possibly be the optimal solution, so in this context each candidate solution is a clique in the graph. Metaheuristic algorithms iteratively improve these solutions until the optimal solution is found. The mechanism by which this improvement is done is dependent on the specific algorithm, and is based on which problem the algorithm aims to solve. In order to know how to improve solutions, the algorithm must also include an evaluation of the quality of each solution at regular intervals, and some procedure to select the optimal solution. This regular evaluation helps guide the algorithm toward more promising areas of the search space.

Most metaheuristics also include some mechanism that introduces randomness to the candidate solutions, which prevents them getting stuck in local optima, which may prevent them from finding the global solution to the problem [13]. For the MCP, "getting stuck in local optima", would mean continually searching the neighbourhood of one previously found large clique, while not exploring other areas of the graph that may include larger cliques. So, the algorithm sometimes moves away from seemingly more promising areas of the search space, and randomly jumps to another part of the search space and searches this area for a while. This is another aspect of metaheuristics that means they require a lot of experimentation. The algorithm must find a balance between enough randomness to ensure sufficient exploration of the search space, while not adding too much that the algorithm is constantly moving away from promising candidate solutions.

Metaheuristics can be split into Single-state methods, which start with one candidate solution and repeatedly improve it, and Population methods, which collect multiple candidates simultaneously, which are improved by exchanging information with each other. Examples of Single-state methods are Hill Climbing, Simulated Annealing and Tabu Search. Examples of Population based methods are Genetic Algorithms, Particle

Swarm Optimisation and Ant Colony Optimisation [13].

3.2 Ant Colony Optimisation

In this project, ACO (Ant Colony Optimisation) was identified as the most suitable metaheuristic. The use of a Population method is preferred as they are more suited to combinatorics problems such as the MCP. Single-state methods like Hill Climbing and Simulated Annealing work better for problems where there is some kind of linear ordering, which is not the case for the MCP. It is better to use a Population method that has multiple candidate solutions exploring different areas of the graph simultaneously. ACO is a well-explored metaheuristic for the MCP, and has obtained reasonably good results [14].

Many metaheuristics are based on processes from nature. ACO is inspired by the way that ants often work together as a population to find food. Each ant forms its own trail by walking randomly and distributing "pheromone" along the trail. The ants that find shorter trails will return faster, and so the pheromone will be spread more densely along these shorter paths. The level of pheromone on a path signals to the rest of the ants how promising the path is. Figure 14 illustrates this idea, where the ants reinforce the shorter trail using pheromones until every ant takes the shortest path to find food.

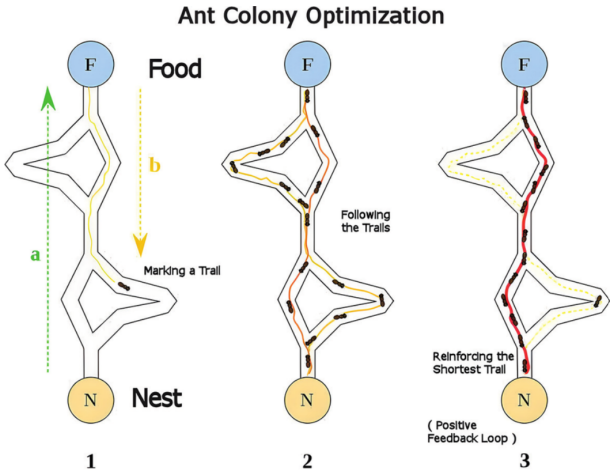

Figure 14: Illustration of ACO [15]

This concept can be applied to the MCP by thinking of the candidate solutions as the ants, and the vertices or edges they are including as trails that they are exploring. The pheromone is a mechanism to store which areas of the graph contain larger cliques. Most ACO algorithms also have a concept of pheromone evaporation, meaning that the pheromone applied to the trail will dissipate after a certain amount of time. This helps ensure that the ants continue exploring different parts of the graph, and so are not getting stuck in local optima at the expense of finding the global optimal solution, which, as discussed, is a core concept of metaheuristics [13].

3.3 Ant-Clique

Because ACO is a well explored metaheuristic for the MCP, there are open source implementations available online. We are using a software called Ant-Clique, which is written in the programming language C [14]. This software implements two versions of ACO, one where the pheromone is laid on the vertices of the graph, called "Vertex-AC" and one

where the pheromone is laid on the edges, called "Edge-AC". This means that "Vertex-AC" keeps a measure of the likelihood of improving the current solution by including each vertex, whereas with "Edge-AC" the inclusion or exclusion of a vertex is dependent on the amount of pheromone on the edge connecting it to the current clique [14] . In this project, the "Edge-AC" version of the algorithm is used for testing, because although it takes longer to find cliques than "Vertex-AC", the cliques found are generally larger. Our symmetry-breaking algorithm aims to accelerate the process of finding the maximum clique, so we are choosing the version of the algorithm that finds better cliques in slightly more time, rather than the version that is already quite fast, but does not find the optimal solution as often.

For the purposes of testing the impact of our symmetry-breaking algorithm, finding a clique "faster" is defined as finding that clique in fewer "cycles". Each cycle is an iteration of the Ant-Clique algorithm, where each ant chooses a random vertex and constructs a maximal clique. After each cycle, the pheromone trails are updated and then each ant chooses a new initial vertex and starts the process again [14]. This was chosen as the metric of comparison because other measurements, such as the CPU time, may be different on different machines, and also because the maximum number of cycles allowed can be set at the beginning of the run.

The original paper [14] that describes the Ant-Clique algorithm has results for the maximum cliques found on many of the DIMACS challenge instances. However, most of the DIMACS challenge instances are not symmetric graphs, and so are not relevant for this project. Therefore, as our testing focuses on highly symmetric graphs, the results cannot be directly compared to the results in the original paper.

4 Symmetry-breaking algorithm

As discussed above, our aim is to improve the performance of Ant-Clique on symmetric graphs by "breaking" the symmetry, so this section details this process by introducing a standard technique in computational combinatorics, sometimes known as "symmetry-breaking" or "orbital branching" [16]. It is particularly useful as it accelerates the search for optimal solutions in combinatorial optimisation problems such as the MCP because it eliminates redundant sub-optimal solutions .

SageMath was used to implement this algorithm. It is a Python-based mathematical software that has group and graph theory specific functions. Pseudocode for this algorithm can be found in section 4.5 and the implementation of this algorithm is included in the Appendix A.1. SageMath is also used to generate the images of graphs included in this report.

4.1 Group and graph theory background

Because our symmetry-breaking algorithm is reliant on concepts from group and graph theory, there are some terms used throughout this section that require definition.

A **permutation group** is a **group** consisting of a set of bijections from a set S to itself. This set of bijections is a group because it satisfies the group axioms; the operation (function composition) is associative, there is a bijection that maps every element of S to itself (the identity bijection) and every function in S has an inverse in S (because it is a bijection) [17].

In the context of this project, the set is the set of vertices, V, and the permutation group is the set of bijections that map V to itself by swapping the positions of the vertices in the graph. The identity element of the group is simply the bijection that does not swap any vertices, and the inverse of any bijection is the element in the group that switches the vertices back to their original position before the bijection was applied to the set.

A **graph isomorphism** is a mapping f where two vertices u and v are adjacent if and only if $f(u)$ and $f(v)$ are adjacent, which essentially means that a graph isomorphism must preserve the edge structure of a graph [2]. A **graph automorphism** is a graph isomorphism that maps the set V of vertices of the graph to itself. These automorphisms form a group called the **automorphism group**.

A **group action** of a group G on a set S is a map from $G \times S$ to S, which is denoted by s^g, where $s^1 = s$ and

$$s^{(g_1 g_2)} = (s^{g_1})^{g_2} \qquad \forall \ g_1,\ g_2 \in G \ \ and \ \ s \in S$$

G acts on S if there exists a group action from $G \times S$ to S [18].

In the context of this project, the group G is the automorphism group of the graph, which can be represented as the group of permutations of V to itself that preserve the vertex-edge structure of the graph, and S is the vertex set V of the graph. So, G acting on V refers to permuting the vertices of the graph, while preserving the vertex-edge structure.

For an element $x \in S$ where G is a group acting on S, the **orbit** [18] of x is the equivalence class of the form

$$x^G = \{x^g : g \in G\}.$$

Within this graph theory context, the orbit of a vertex v is the set of vertices that can be obtained by applying all possible permutations in the automorphism group to v.

The **stabiliser** of an element $x \in S$ is the set

$$G_x = \{g \in G : x^g = x\},$$

it is the set of elements in G that fix x [18]. The **stabiliser** of an element v of the vertex set of a graph is the set of elements of the automorphism group that fix v, i.e. permutations where v is not swapped with any other vertex.

The action of G on S is **transitive** when there is only one orbit. So, take two arbitrary elements $x, y \in S$, there must exist $g \in G$ such that $x^g = y$, i.e. G acts transitively on S [17]. A graph Γ is **vertex-transitive** if the automorphism group of Γ acts transitively on the set of vertices V [19], i.e. any vertex can be mapped by an automorphism to any other vertex in the graph. A graph Γ is **edge-transitive** if any edge can be mapped by an automorphism to any other edge [19]. A graph Γ is **distance-transitive** if for all vertices u, v, x, y of Γ, when the distance between u and v is the same as the distance between x and y, then there exists an automorphism f such that $f(u) = x$ and $f(v) = y$. Distance-transitivity implies vertex-transitivity, however, the converse does not hold [19].

An **induced subgraph** is a subgraph of a graph Γ created from a subset of the vertex set of Γ , where an edge is added to the subgraph if both endpoints of the edge are in this subset[1], see Fig 15 and Fig 16 below.

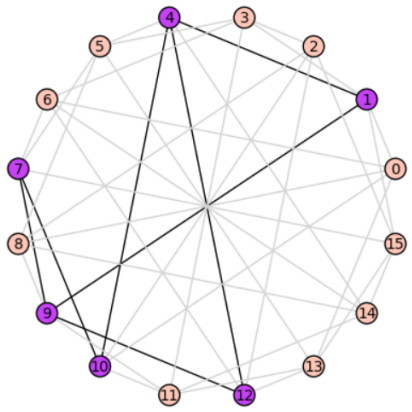

Figure 15: Induced subgraph within Clebsch graph using the subset {1,4,7,9,10,12}

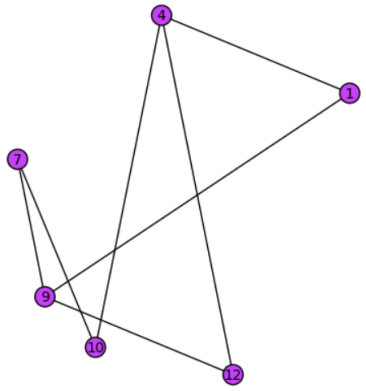

Figure 16: The resulting induced subgraph

The **neighbourhood** of a vertex $x \in V(\Gamma)$ is defined as $\Gamma(x) = \{y \in V(\Gamma) : y \sim x\}$. So, for each vertex x, the set $\Gamma(x)$ consists of all vertices which are adjacent to x. Note that $\Gamma(x)$ excludes x itself as our focus is restricted to simple graphs, i.e. no self loops are present. The **common neighbourhood** of a set $S \subseteq V$ is $\Gamma(S) = \bigcap_{v \in S} \Gamma(v)$. This generalises the concept of neighbourhoods above from a single vertex to a set of vertices. In this project, this will be applied to the common neighbourhood of a starter set. As $v \notin \Gamma(v)$, clearly $v \notin \Gamma(S)$ for $v \in S$, meaning that the common neighbourhood of the starter set excludes the vertices in the starter set. The **subgraph induced by the common neighbourhood** of a set S, that is, the subgraph induced by $\Gamma(S)$ is denoted by Γ_S. Similarly, this concept will be used within the setting of starter sets.

4.2 Equivalency and starter sets

For our symmetry-breaking algorithm, the famous Petersen graph is used, pictured in Fig 17, to demonstrate how our algorithm works. This graph was chosen because it is symmetric and a strongly regular graph. It is large enough to understand the concept of our symmetry-breaking algorithm but sufficiently small to allow us to show each step in detail. Note that usually vertices are labelled $\{1, 2, 3, ..., n\}$ but as SageMath, which is Python-based, was used to generate these graphs they are labelled $\{0, 1, 2, ..., n-1\}$.

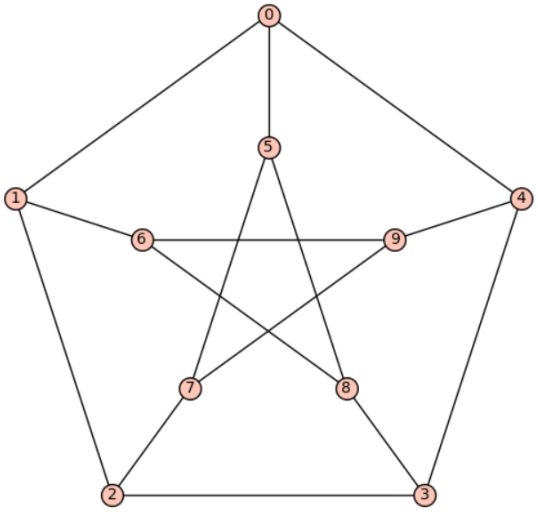

Figure 17: Petersen graph

The aim of our algorithm is to find "starter sets". These are sets of vertices that are cliques or cocliques of a certain size, depending on which is being considered. These are denoted by a set in bold in this section. The objective of finding the starter sets is to find representatives for each clique or coclique in a graph, i.e. only wanting to find starter sets that cannot be permuted into each other under the **automorphism group** of the graph.

Looking at Fig 18 and 19 below, it is clear that the starter set **{0,2,8,9}** and another coclique of size 4, with the set $\{1, 3, 5, 9\}$, are essentially the same graph, just rotated. So, the permutation mapping is as follows $(0, 1, 2, 3, 4)(5, 6, 7, 8, 9)$.

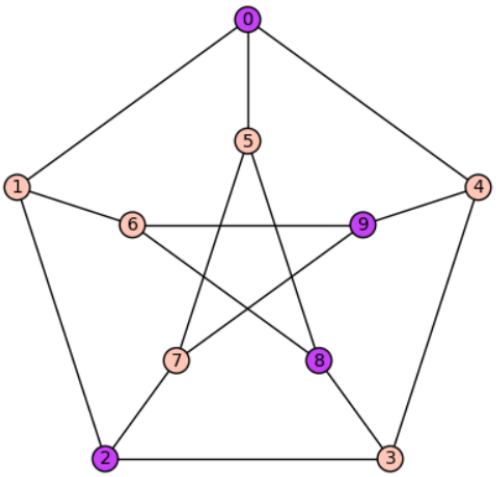

Figure 18: Starter set of size 4

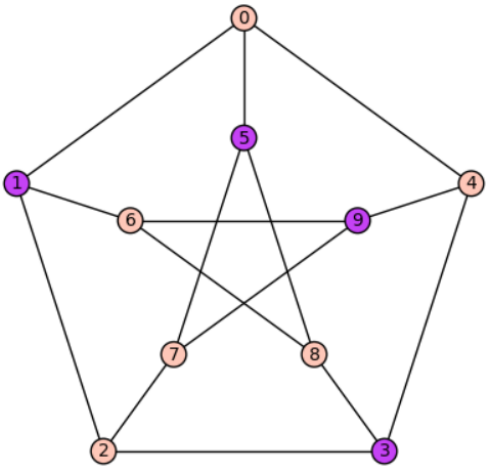

Figure 19: Another coclique of size 4

This equivalency is the idea behind our algorithm as it is redundant to consider both because they are essentially the same. Symmetry-breaking reduces the search space for the MCP. This is achieved by finding the orbits of the stabilisers of different vertices of our graph. Firstly, it starts by finding the automorphism group of the graph, then find the orbits of that group.

4.3 Example: Finding cliques of the Petersen graph

The algorithm starts by finding a representative from each orbit of the automorphism group of the graph. As the Petersen graph is vertex-transitive, all the vertices are in the

same orbit and it does not matter which vertex is chosen as a representative. The smallest labelling is chosen for convenience, so **{0}** is added to the starter set. The vertex 0 is then fixed and the stabiliser of **{0}** is found in the graph. The orbits of this stabiliser are computed, which are $\{0\}, \{1, 4, 5\}, \{2, 3, 6, 7, 8, 9\}$. This can be seen in Fig 20. The orbits are divided in this way because the purple vertices 1, 4 and 5 are all adjacent to 0, the vertex being fixed, so these vertices are equivalent under the group action. The grey vertices 2, 3, 6, 7, 8 and 9 are not adjacent to 0 so these are equivalent under the group action. Therefore, only one representative is chosen from each orbit because the vertices within each orbit are equivalent under the group operation.

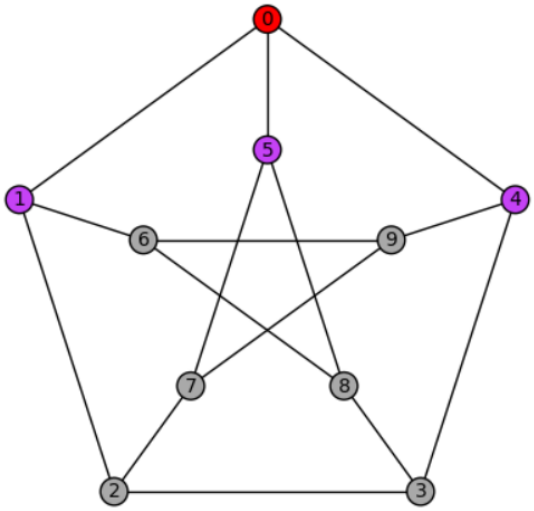

Figure 20: Orbits of the stabiliser of $\{0\}$ in the Petersen graph

When finding cliques, the grey orbit $\{2, 3, 6, 7, 8, 9\}$ is disregarded as none of these vertices are adjacent to 0. Then observe that the vertices in the purple orbit $\{1, 4, 5\}$ are all adjacent to 0. Hence 1, the smallest label, is taken as a representative of this orbit. 1 is added to the starter set, resulting in a current starter set of **{0,1}**. Next, the setwise stabiliser of **{0,1}** is calculated, which is the set of permutations that fix the current starter set, and the orbits of this are computed. This results in the orbits $\{0, 1\}, \{2, 6, 4, 5\}, \{3, 7, 8, 9\}$, which can be seen in Fig 21.

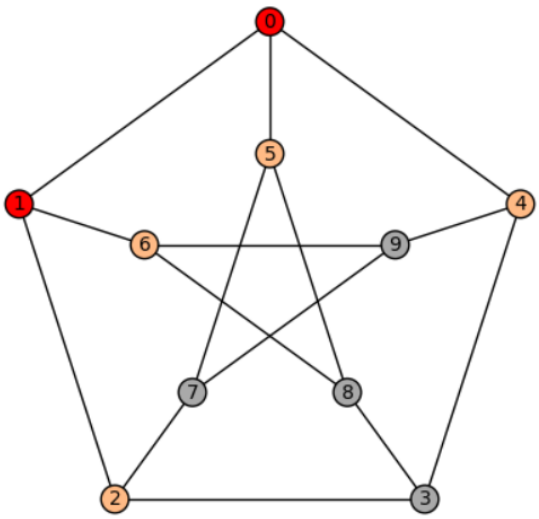

Figure 21: Orbits of the stabiliser of $\{0, 1\}$ in the Petersen graph

The grey orbit contains vertices $\{3, 7, 8, 9\}$. These are not adjacent to the starter set vertices 0 or 1, hence they are ignored. Each element of the orange orbit $\{2, 4, 5, 6\}$ is only adjacent to either 0 or 1, not both, so the clique cannot be extended any further and our algorithm now terminates for this graph. This means that the maximum clique in the Petersen graph is of size 2, i.e. $\omega(\Gamma) = 2$. **{0,1}** is the only starter set for the clique problem for the Petersen graph and so represents all other cliques of size 2 in the graph.

4.4 Example: Finding cocliques of the Petersen graph

The Petersen graph is more interesting for cocliques, since its maximum coclique is larger than its maximum clique, leading to a greater number of steps in our algorithm. Let us return to the orbits when vertex 0 is fixed, $\{0\}, \{1, 4, 5\}, \{2, 3, 6, 7, 8, 9\}$ as seen in Fig 22.

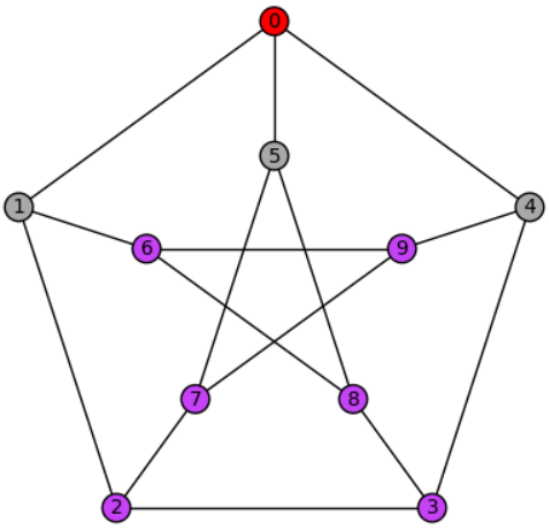

Figure 22: Orbits of the stabiliser of $\{0\}$ in the Petersen graph

This time since the search is for cocliques, the grey orbit $\{1, 4, 5\}$ is overlooked because only the vertices that are not adjacent to 0 are considered. So, the focus is on the purple vertices $\{2, 3, 6, 7, 8, 9\}$. The vertex 2 is chosen as a representative and added to the starter set resulting in **{0,2}**. Then consider the setwise stabiliser of $\{0, 2\}$. Computing the orbits of this setwise stabiliser gives $\{0, 2\}, \{1\}, \{3, 4, 5, 7\}, \{6\}, \{8, 9\}$ as shown below.

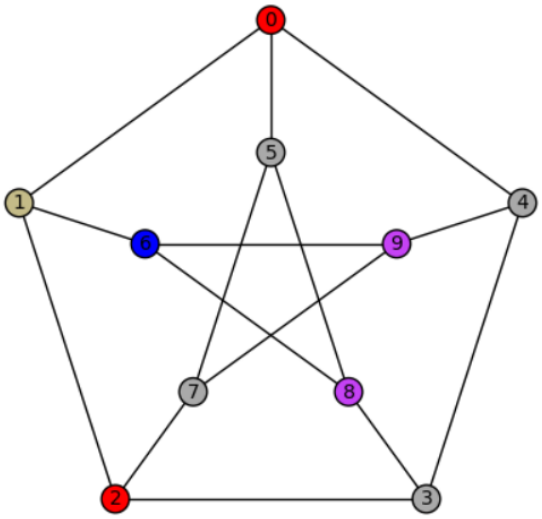

Figure 23: Orbits of the stabiliser of $\{0, 2\}$ in the Petersen graph

As the search is for cocliques, the grey orbit $\{3, 4, 5, 7\}$ is ignored, because each vertex is connected to either 0 or 2. $\{1\}$ is in its own orbit, because it is adjacent to both 0 and 2, so this is ignored as well. The blue orbit $\{6\}$ and the purple orbit $\{8, 9\}$ are the only remaining orbits which are not adjacent to any vertex in the starter set **{0,2}**, so a representative is picked from each of these orbits. As the vertices 8 and 9 are in the same orbit, they are equivalent under the group action, so only one of them, 8, is added to the starter set. 6 is the only vertex in its orbit, so it also taken as a representative. Vertices 8 and 6 are both added individually to the starter set, which creates a list of two starter sets each of size 3. So, the starter sets of size 3 for cocliques are **{0,2,6}** and **{0,2,8}**.

To continue finding starter sets of larger sizes, the algorithm begins with the starter sets of size 3. The setwise stabiliser of **{0,2,6}** is computed and its orbits $\{0, 2, 6\}, \{1\}, \{3, 4, 5, 7, 8, 9\}$ are found. $\{1\}$ is its own orbit, because it is the only vertex adjacent to 0, 2 and 6. This means that it can be neglected when pursuing cocliques. The other orbit $\{3, 4, 5, 7, 8, 9\}$ consists of all vertices connected to either 0, 2 or 6, meaning everything in this orbit must also be ignored. Therefore, the coclique starter set **{0,2,6}** cannot be extended further.

Subsequently, the other starter set of size 3, **{0,2,8}**, is examined. The orbits of the stabiliser of this starter set are $\{0, 2, 8\}, \{1, 3, 5\}, \{4, 6, 7\}, \{9\}$, which are seen in Fig 24. The grey orbit $\{1, 3, 5\}$ and the beige orbit $\{4, 6, 7\}$ are again disregarded for cocliques as they are adjacent to at least one vertex in the starter set. 9 is in its own orbit because it is the only vertex not adjacent to 0, 2 or 8. Therefore, 9 is added to the starter set to get a starter set of size 4, **{0,2,8,9}**. This coclique starter set shown in Fig 25 represents all other cocliques of size 4 in the Petersen graph as there is only one starter set of size 4. Every coclique of size 4 can be permuted under the group operation of the graph so that they are equivalent to this starter set.

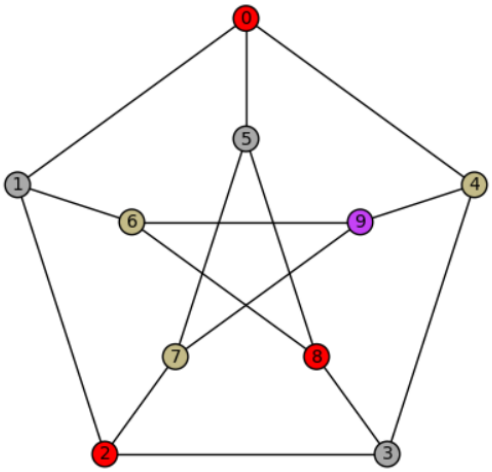

Figure 24: Orbits of the stabiliser of $\{0, 2, 8\}$ in the Petersen graph

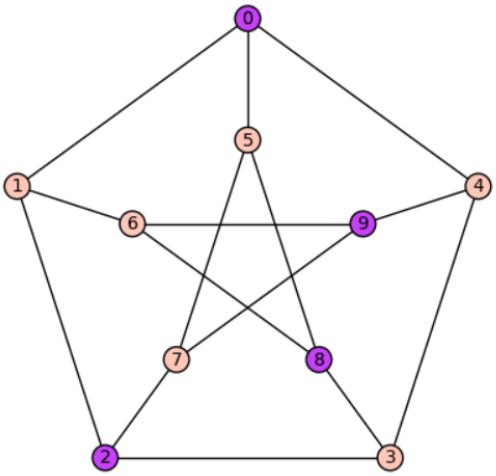

Figure 25: Coclique of size 4 of the Petersen graph

At this point our algorithm will terminate since there are no orbits or vertices that are not adjacent to everything in the starter set **{0,2,8,9}** that could extend the coclique size. Thus, the maximum coclique of the Petersen Graph is of size 4 i.e. $\alpha(\Gamma) = 4$.

4.5 Pseudocode for symmetry-breaking algorithm

Algorithm 4: Symmetry-breaking algorithm

```
Data: Γ: Graph
k : size of starter set
Result: starter_sets : list of starter sets of size k
Initialisation;
automorphism ← Automorphism group of Γ;
orbits ← orbits of automorphism;
stack ← empty list to store incomplete starter sets;
for orbit in orbits do
    representative ← first element of orbit;
    stack ← stack + {representative};
end
while stack is not empty do
    current_ss ← pop off of stack;
    stabiliser ← setwise stabiliser of current_ss on automorphism;
    stab_orbits ← orbits of stabiliser;
    for orbit in stab_orbits do
        representative ← first element of orbit;
        if representative not in current_ss AND length(current_ss) < k then
            if representative adjacent to every element of current_ss then
                new_ss ← current_ss + representative;
                sort new_ss;
                stack ← stack + new_ss;
                if new_ss not in starter_sets AND length(new_ss) == k then
                    starter_sets ← starter_sets + new_ss;
                end
            end
        end
    end
end
return starter_sets
```

4.6 Neighbourhoods of starter sets

Once the list of starter sets of a specific size is constructed to find cliques or cocliques, we can show how they operate in conjunction with the Ant-Clique algorithm using the Schläfli and Hoffman-Singleton graphs as examples.

The subgraphs induced by the common neighbourhood or the non-neighbourhood of the starter sets, for cliques and cocliques respectively, are created. The code to construct these induced subgraphs can be found in the Appendix A.2. These subgraphs are then fed into Solnon and Fenet's Ant-Clique algorithm [14] to heuristically find the maximum clique of each one.

To construct the starter sets, our symmetry-breaking algorithm only removes equivalent cliques or cocliques, so a maximum clique is guaranteed to be in at least one of these induced subgraphs. Consequently, the size of the maximum clique of the original graph is then the largest clique found in the subgraphs plus the size of the starter set used.

4.6.1 The Schläfli graph: Finding cliques

The Schläfli graph is a strongly regular graph that has parameters $(27, 16, 10, 8)$. So, it is a 16-regular graph that has 27 vertices and 216 edges [20]. The Schläfli graph has a maximum coclique of size 3 and therefore is not particularly informative in this context, for example $\{1, 2, 3\}$ is a maximum coclique. Using our algorithm to find a starter set of size two for cliques, only one starter set **{1,12}** is obtained, shown below in blue in Fig 26. Then the common neighbourhood of this starter set, i.e. $\Gamma(\{1, 12\})$, is computed, that is, the vertices that are adjacent to all the vertices in the starter set. The starter set vertices must be all adjacent to each other as they form a clique. Finding the common neighbourhoods identifies candidate vertices to expand the starter set **{1,12}** to a larger clique. The common neighbourhood shown in Fig 27 in purple is then constructed from the starter set. Finally, the associated subgraph induced by this common neighbourhood, $\Gamma_{\{1,12\}}$, is shown in Fig 28. Note that the starter sets are not included in the induced subgraphs as it is known, by construction, that these can be added to extend any clique found in the subgraphs induced by the common neighbourhoods of the starter sets.

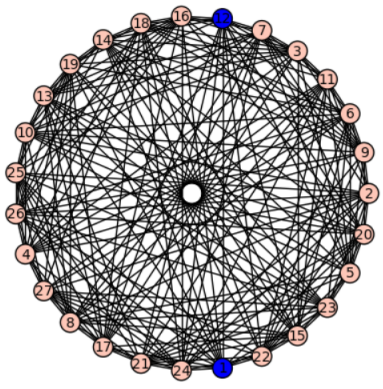

Figure 26: Schläfli graph with starter set of $\{1, 12\}$

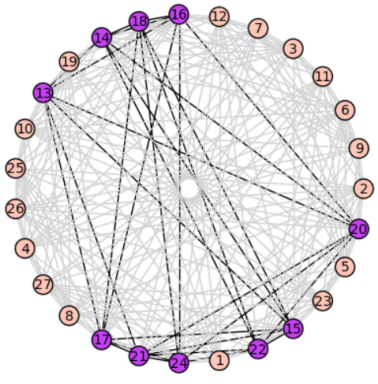

Figure 27: Schläfli graph with $\Gamma(\{1, 12\})$ in purple

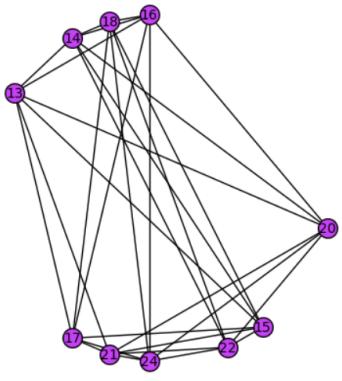

Figure 28: $\Gamma_{\{1,12\}}$

The Schläfli graph has 72 maximum cliques of size six, one of which is given by the set of vertices

$$\{1, 12, 13, 14, 16, 20\}.$$

Using the starter set of **{1,12}**, the last four vertices of the clique can be seen in the induced subgraph of the common neighbourhood of **{1,12}** in Fig 28. This clique of

size 4 is the maximum clique of this induced subgraph. Thus, the maximum clique of the original graph is the union of the starter set used and the maximum clique in the subgraph induced by the common neighbourhood of the starter set.

Now, instead of applying the metaheuristic algorithm Ant-Clique to the entire original graph, it can just be applied to the multiple smaller subgraphs induced by the common neighbourhood, to reduce computational cost and time. In this small example, since it only has one starter set and thus one induced subgraph, the effect of our algorithm can be directly compared to the original graph. Our algorithm reduces the number of edges to check from 216 to 30, resulting in an 84.11% decrease from the original graph. This makes the search for cliques much more efficient.

In larger graphs, multiple subgraphs induced by the common neighbourhood of each distinct starter set are acquired. Consequently, a direct comparison to the original graph cannot be created as done above with the reduction in edges. The benefit of our symmetry-breaking algorithm is not solely the reduction in edges in our subgraphs but the removal of redundant, equivalent starter set cliques or cocliques. These repeated calculations are removed in the search for the maximum clique for more efficiency, resulting in an increased likelihood of finding an optimum clique. In the induced subgraph in Fig 28, there are 5 different maximum cliques. This is substantially reduced from the 72 maximum cliques in the original graph.

4.6.2 The Hoffman-Singleton graph: Finding cocliques

The Hoffman-Singleton graph Γ is a strongly regular graph with parameters $(50, 7, 0, 1)$. It is a 7-regular graph with 50 vertices and 175 edges [20]. As $\lambda = 0$, each pair of adjacent vertices has zero neighbours in common, this implies that the maximum clique is of size 2 and that there are no triangle cliques in the graph. Thus, our search for cocliques in the Hoffman-Singleton graph using our symmetry-breaking algorithm.

Note that our implementation does not directly compute the non-neighbourhood of the starter set as described below, it is provided only for clarity and the visibility of the diagrams as the complement graph can be dense and be visually cluttered. The complement of the Hoffman-Singleton graph has 1050 edges compared to only 175 edges in the original. Our symmetry-breaking algorithm finds the starter sets for cliques in a graph. For determining the maximum coclique our implementation takes the complement of the graph and finds the maximum clique of this graph by creating the subgraphs induced by the common neighbourhood of the clique starter sets of the complement graph.

The starter sets of size 3 obtained by the algorithm are **{1,3,6}** and **{1,3,29}**, the former of which can be seen in Fig 29. When looking for cocliques, the starter set vertices are not adjacent to each other, so the algorithm finds the set of vertices in the original graph that are not adjacent to any vertex in the starter set. This is called the non-neighbourhood of the starter set, which is exactly the common neighbourhood of the starter set in the complement of the graph $\overline{\Gamma}$. The vertices in the non-neighbourhood are potential vertices that could extend the starter set coclique into a larger coclique. This is shown in Fig 30. The associated induced subgraph by the non-neighbourhood of the starter set **{1,3,6}** can be seen in Fig 31.

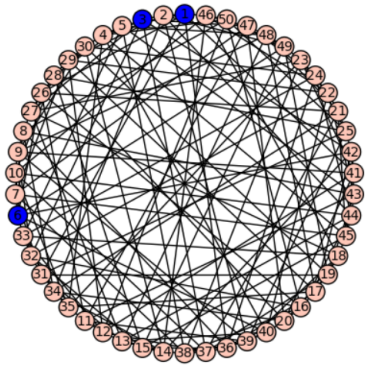

Figure 29: Hoffman-Singleton graph with starter set of $\{1, 3, 6\}$

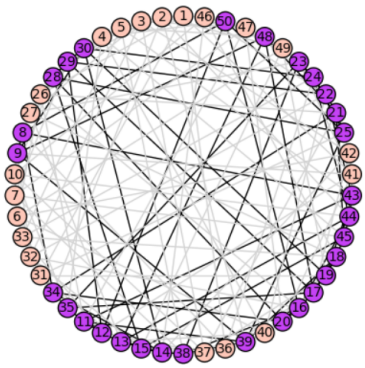

Figure 30: Hoffman-Singleton graph with subgraph induced by the non-neighbourhood of $\{1, 3, 6\}$

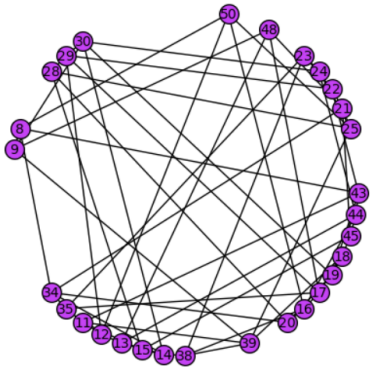

Figure 31: Induced subgraph

Again note that the induced subgraph shown above is of the non-neighbourhood of the original Hoffman-Singleton graph. Our algorithm actually finds the subgraphs induced by the common neighbourhood of the starter sets in the complement graph. So, the set of vertices shown in Fig 31 are correct in the sense that they are the same set of vertices as is in the induced subgraph of the common neighbourhood of the complement graph. However, the edges displayed on this figure do not correspond to the set of edges in the complement induced subgraph. In particular, Fig 31 has 64 edges, whereas, the complement graph induced subgraph for the common neighbourhood of **{1,3,6}** contains 342 edges, so the image above is slightly misleading, while the vertex set is correct the edge set is not.

The Hoffman-Singleton graph contains 100 maximum cocliques of size 15, one of which is

$$\{1, 3, 6, 8, 12, 14, 16, 19, 22, 24, 28, 35, 39, 45, 48\}$$

This coclique lies in the induced subgraph by the non-neighbourhood of the starter set **{1,3,6}** in Fig 31 together with the vertices of the starter set itself. So, the subgraph induced by the common neighbourhood of the starter set **{1,3,6}** has a maximum coclique of size 12, plus the starter set of size 3 to get the original maximum coclique of size 15.

In the induced subgraph in Fig 31, there are four distinct maximum cocliques and similarly, the induced subgraph of the common neighbourhood of **{1,3,29}** has two distinct maximum cocliques of the same size. Although the original graph has 100 maximum cocliques, our algorithm identifies equivalence among many of them and reduces these to six different cocliques.

This subgraph cannot be directly compared to the original graph as there are two subgraphs. To compare the outcome of our algorithm on the original graph and the induced subgraphs, the results section 5.4 introduces a coefficient β that considers the fact that each induced subgraph must be taken into account.

4.7 Advantages of symmetry-breaking

When trying to find cliques and cocliques of highly symmetric graphs, it is somewhat redundant to search the whole graph and find several cliques that are essentially the same clique, just permuted. Therefore, by applying this symmetry-breaking algorithm to highly symmetric graphs before applying the metaheuristic algorithm, a significant amount of the search space can be eliminated. Our algorithm results in a list of starter sets. Instead

of applying the metaheuristic to the original graph itself, the common neighbourhood of each starter set is found, and these induced subgraphs are given to the metaheuristic. This is in addition to the fact that it is predicted that our metaheuristics will work better on graphs that are less symmetric.

Another advantage of our symmetry-breaking algorithm is that the induced subgraphs are completely independent of each other. If the checking of each subgraph is done simultaneously, then in principle the results of the subgraphs could be compared more directly with those from the original graph. This can be achieved by running the Ant-Clique algorithm on each induced subgraph in parallel. This would reduce the overall computation time, or alternatively allowing the run time of the original algorithm to be distributed over several processes.

Although this starter set method can be used to find cliques and cocliques itself, after a certain point the benefits of finding larger and larger starter sets may be outweighed by the cost of repeatedly calculating orbits and stabilisers. Therefore, another element to consider in the project is finding, by trial and error, the point at which to stop calculating starter sets and start using the metaheuristic Ant-Clique algorithm to find the maximum cliques and cocliques using the subgraphs induced by the common neighbourhood of the starter sets.

4.8 Balancing starter set size and metaheuristic starting point

The choice of starter set size greatly affects the number of subgraphs induced by the common neighbourhood and the size of these induced subgraphs. A smaller starter set size normally produces a smaller number of subgraphs induced by the common neighbourhood, however, the number of vertices in these induced subgraphs is larger. As the size of the starter set increases, the number of vertices in the induced subgraphs decreases leading to a simpler subgraph, although the number of actual subgraphs to check can increase. Finding a balance in the starter set size is difficult to establish, but choosing starter sets of size 1, 2 and 3 seem to be the most rewarding. Increasing beyond this for most graphs provides too many induced subgraphs, leading to a substantial increase in the computational cost required to test each subgraph with the Ant-Clique algorithm making it impractical. For this reason, only starter sets up to size 3 were tested with our algorithm.

The figures shown below demonstrate how the number of vertices in the subgraphs induced by the common neighbourhood in the Schläfli graph decreases as the starter set sizes increases. The Schläfli graph is useful for this demonstration as it is vertex-transitive and for all starter set sizes up to size 5 provides only one starter set i.e. only one induced subgraph. There is only one orbit in the orbits generated by the setwise stabilisers of the starter sets, that is adjacent to everything in the starter set, for all starter sets up to size 5. There are two starter sets of size 5 but only one of these has a common neighbourhood that consists of the last vertex to be added to the starter set to get the maximum clique of size 6 in the Schläfli graph.

Induced subgraphs of the common neighbourhood of the Schläfli graph with different sizes of starter sets

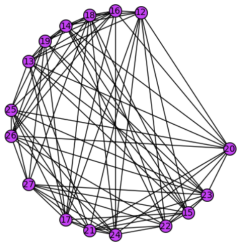

Figure 32: $\Gamma_{\{1\}}$

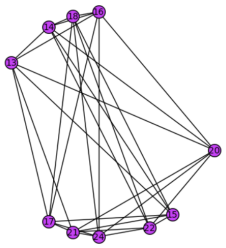

Figure 33: $\Gamma_{\{1,12\}}$

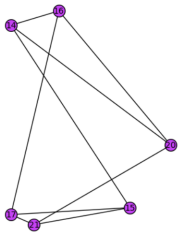

Figure 34: $\Gamma_{\{1,12,13\}}$

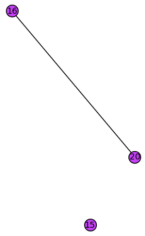

Figure 35: $\Gamma_{\{1,12,13,14\}}$

Figure 36: $\Gamma_{\{1,12,13,14,15\}}$

The Hoffman-Singleton graph is used as an example to demonstrate the effect that the size of our starter sets have on the amount of induced subgraphs created, seen in Table 1 below. Clearly, the number of induced subgraphs can become quite large as the size of the starter set increases. Hence, our reason for avoiding starter sets of sizes greater than 3.

starter set size	*#starter sets*	*#induced subgraphs produced*
1	1	1
2	1	1
3	2	2
4	6	6
5	23	23
6	113	113
7	618	617
8	2444	2444

Table 1: The effect of the starter set size on the number of subgraphs induced by the common neighbourhood of the starter sets produced

5 Hamming graphs

This section is a case analysis of an important family of symmetric graphs called the Hamming graphs. It details their structure, properties and applications that make this a suitable and interesting family of graphs to test the impact of symmetry-breaking on.

5.1 Structure and properties

The Hamming Graphs are a family of graphs with the form $H(d, q)$. The vertex set of $H(d, q)$ is the set of all sequences of length d from the set of integers from $\{1, 2, ...q\}$ (Or $\{0, 1, ...q - 1\}$ depending on construction). It can also be thought of as the Cartesian product of d complete graphs K_q [21], where K_q represents a graph with q vertices where every pair of vertices are adjacent. The vertex set has cardinality q^d, and there is an edge between two vertices if and only if their Hamming distance is 1. The Hamming distance between two sequences is the number of positions in which the sequences differ, e.g. the Hamming distance of 001 and 011 is one. Figure 37 below is the Hamming graph $H(3, 2)$ generated with SageMath. You can see that the vertices are labelled with sequences of length d from the set $\{0, 1\}$, since SageMath indexes from 0 not 1, and vertices that differ in one position only have an edge between them.

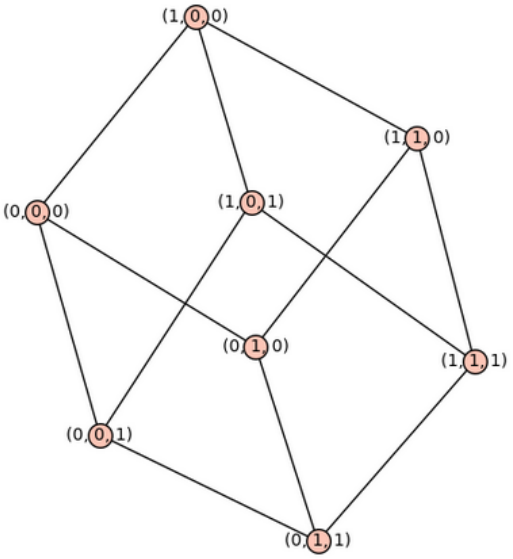

Figure 37: H(3,2)

We have decided to focus on Hamming graphs for our symmetry-breaking algorithm because they are highly symmetric graphs. They are examples of **distance-regular** graphs, meaning that given two vertices x and y that are distance h apart, the number of vertices that are at distance i from x and the number of vertices at distance j from y does not depend on the choice of x and y, only on the choice of h, i and j [22]. The Hamming graphs are also **distance-transitive**. These concepts are related, in that every distance-transitive graph is distance-regular, but the converse does not hold [22]. Distance-transitivity implies vertex-transitivity, so the Hamming graphs are also vertex-transitive. This makes them a good candidate for our symmetry-breaking algorithm, because vertex-transitivity means their automorphism group has only one orbit. The symmetry-breaking algorithm starts with one representative of each orbit, so if there is only one orbit in the initial automorphism group, there are less possibilities for starter sets and less neighbourhoods to check.

5.2 Clique and coclique number

Proposition: The maximum clique of a $H(d, q)$ Hamming graph is of size q.
Proof: Choose a vertex x of $H(d, q)$, with the tuple $x_1x_2...x_d$ as its label. Let x be adjacent to another vertex $y = y_1y_2...y_d$. This implies that x and y differ in exactly one position, i.e. for some $1 \leq t \leq d$, $x_t \neq y_t$ and $x_i = y_i \ \ \forall\ 1 \leq i \leq d$ and $i \neq t$. If x is also adjacent to the vertex $z = z_1z_2...z_d$, x and z must differ in one position, denoted s. If $s \neq t$, then y and z differ in two positions, which implies they are not adjacent, and so x, y and z do not form a clique. Otherwise $s = t$, meaning y and z differ from x in the same position, and so y and z are adjacent, so x, y and z form a clique. Therefore the only cliques in $H(d, q)$ are sets of vertices whose labels all differ from each other in the same position. There are q choices for this position, meaning there are q vertices that are pairwise adjacent, so the largest possible clique is of size q. □

Looking at the below figure for $H(2, 3)$ we can see that the size of the maximum clique is 3, and there are several cliques of size 3, where the vertex labels of the elements of the clique all differ in the same position.

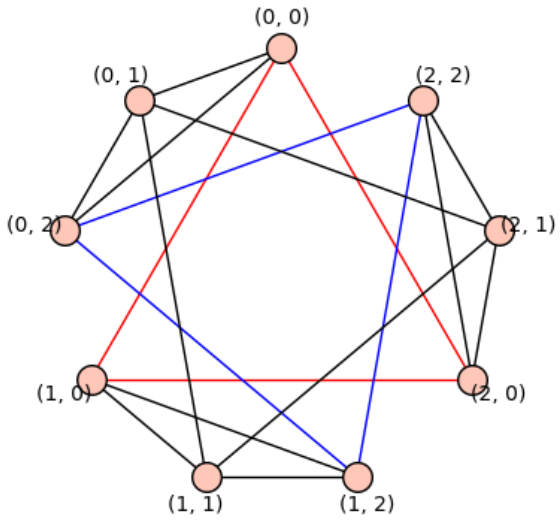

Figure 38: Hamming(2,3) with 2 cliques of size 3 highlighted

Because the Hamming graphs are vertex-transitive, they must satisfy the clique-coclique bound. This states that for a graph Γ, $\alpha(\Gamma)\omega(\Gamma) \leq |V(\Gamma)|$. This means that because the cliques of the Hamming graphs are restricted to q and so are quite small, the cocliques can be quite large. Given that $|V(H(d, q)| = q^d$ and using the previous result $\omega(H(d, q)) = q$, it follows that $\alpha(\Gamma) \leq \frac{|V(\Gamma)|}{\omega(\Gamma)} = \frac{q^d}{q} = q^{d-1}$. This bound is sharp for the Hamming graphs [21], so the size of the maximum coclique of $H(d, q)$ is q^{d-1}. Therefore, the cocliques of the Hamming graphs are a better case study to test our symmetry-breaking on.

5.3 Applications in coding theory

Additionally, the cocliques of the Hamming graphs have an interesting application in coding theory. As discussed in section 2.1.2, error detecting / error correcting codes can be modelled as graphs, with vertices representing codewords. Vertices u and v are adjacent if u could arise from an error e on v, where e is a specific error, e.g. a single transposition or a double deletion of bits. For the $H(d, q)$ graphs we are examining, each vertex is labelled with a block code of length d from the alphabet $\{1, 2, ..q\}$ and two vertices are adjacent if

and only if they differ in exactly one position. This means that the adjacent vertices to a vertex are precisely the set of codewords that could arise from a single transposition error, which is where one digit of a codeword is mistransmitted. Finding a coclique of a $H(d, q)$ graph corresponds to finding a set of codewords where each pair of vertices differ in at least two positions, since any pair of vertices that are adjacent have Hamming distance one. In coding theory, a code, which is a set of codewords, can detect a t-transposition error (i.e. t bits have been changed) if the minimum distance of the code is at least $t + 1$ [23]. The minimum distance of a code is the minimum Hamming distance between any two codewords. For a coclique of $H(d, q)$, the minimum distance is 2, since every pair of vertices in the coclique differ in at least two positions, meaning that a coclique of a $H(d, q)$ graph corresponds to a 1-error detecting code. Since $t + 1 = 2$ means $t = 1$.

To see this, examine the coclique $\{(0, 0), (1, 1), (2, 2)\}$ of $H(2, 3)$. If, for example $(0, 0)$ were mistransmitted as $(0, 1)$, the code is able to detect this error, since $(0, 1)$ is not a codeword of this code. However, the code is not able to correct this error, since it cannot tell whether the mistransmitted $(0, 1)$ was originally $(0, 0)$ or $(1, 1)$. This illustrates that the cocliques of $H(d, q)$ form a 1-error detecting code, but not a 1-error correcting code, because in order for a code to correct a t transposition, its minimum distance must be at least $2t + 1$ [23].

The Hamming graphs we have been examining so far are of the form $H(d, q)$, but graphs of the form $H_q(n, d)$ are also referred to as Hamming graphs by some texts . These graphs have vertex labels as sequences of length n from the alphabet $\{1, 2, ..q\}$, and two vertices are adjacent if their sequences differ in at least d positions. For these types of graphs, the clique number of $H_q(n, d)$ is equal to the maximum number of codewords that can be obtained from sequences of length n from the alphabet $\{1, 2, ..q\}$, while ensuring every pair of codewords differs by at least d [24]. So, these Hamming graphs can correspond to error-correcting codes, in addition to error-detecting codes, since if $d \geq 2t + 1$, the code corresponding to a clique in $H_q(n, d)$ can correct up to t errors.

5.4 Results on the Hamming graphs

To test the effect of our symmetry-breaking algorithm on the Hamming graphs, we compared the maximum clique found using the Ant-Clique algorithm both with and without using symmetry-breaking. The coclique version of the SageMath symmetry-breaking algorithm was used for each graph, which meant that it first calculated the complement graph, and then found the starter sets of the cliques in this complement graph. Our algorithm then found the common neighbourhood of each starter set and wrote the induced subgraphs of these common neighbourhoods to a file in DIMACS format.

In section 4.6, the theory behind symmetry-breaking was described. It stated that the subgraphs induced by the common neighbourhood of each starter set would be given to the Ant-Clique algorithm, and the size of the starter set would be added to the largest clique found in the common neighbourhood graph, to get the size of the maximum clique in the original graph. However, in reality, as can be seen in Appendix A.1, the starter set itself was included in the neighbourhood graph given to Ant-Clique. (This is discussed further in section 6.3). This allows a direct comparison between the results on the original graph and the neighbourhood subgraph results given by Ant-Clique.

Each common neighbourhood induced subgraph was inputted to the Ant-Clique algorithm. The largest clique found in any of the neighbourhood graphs was recorded (which is equivalent to the largest coclique found in the original Hamming graph). Also recorded was the least number of cycles that Ant-Clique took to find this clique. The number of cycles was chosen as our metric of comparison because, unlike other metrics like the CPU

time, it is the same across different machines. This means that we were able to run the same graphs on our respective computers and get the same cycle numbers and results, because Ant-Clique is deterministic when using the same random seed, and the default seed was used each time.

SageMath was used to write the complement graph of the original Hamming graph to a DIMACS file. This graph was given to the Ant-Clique algorithm and ran until it found a clique of the same size as the largest clique recorded in the common neighbourhood subgraphs.

To compare the performance with and without the symmetry-breaking algorithm, we have defined this β coefficient,

$$\beta = \frac{\#cycles\ for\ original\ \Gamma}{\#cycles\ for\ best\ \Gamma_{ss_i} \times \#\Gamma_{ss_i}}$$

where ss_i denotes a starter set of size i.

It compares the minimum amount of work necessary to find the largest coclique in any of the Γ_{ss_i} to the minimum work taken to find a coclique of the same size on the original graph. It incorporates the number of common neighbourhood induced subgraphs that need to be searched, since the largest coclique is not included in each Γ_{ss_i}. $\beta > 1$ indicates that the symmetry-breaking algorithm reduces the minimum amount of work required to find a clique, whereas $\beta < 1$ indicates that symmetry-breaking increases the workload to find a clique of the same size.

The results for the starter sets of size 1 are varied, as seen in Table 2. For some graphs, symmetry-breaking reduced the number of cycles, whereas for some it increased the required number of cycles.

Graph	*Vertices*	*Size of max coclique for* Γ_{ss_1}	*#cycles for best* Γ_{ss_1}	$\#\Gamma_{ss_1}$	*#cycles on original* Γ	β
$H(4,7)$	2401	303	2891	1	3106	1.074
$H(4,8)$	4096	447	1008	1	35	0.034
$H(5,4)$	1024	256	807	1	1025	1.27
$H(10,2)$	1024	512	206	1	235	1.14
$H(11,2)$	2048	1024	295	1	241	0.817
$H(12,2)$	4906	2037	330	1	324	0.98

Table 2: Exploration using Ant-Clique, with starter set of size 1
Γ_{ss_1} = induced common neighbourhood graph for starter sets of size 1

For starter sets of size 1, there is not much symmetry-breaking being performed. Because the Hamming graphs are vertex-transitive, there is only one orbit in their automorphism group, meaning each vertex is equivalent under the permutation operation. This is why there is only one induced common neighbourhood subgraph for each graph. So, essentially, all the algorithm does is choose the first vertex and find its neighbourhood. This testing focused on finding cocliques in the Hamming graphs, and each vertex in $H(d,q)$ has degree d, which means that each vertex has very few neighbours in the original graph. As a result of this, the set of neighbours of a vertex in the complement graph is almost the entire graph, so for the Hamming graphs, the number of vertices in the graph is not reduced by a significant amount. Therefore, it is unsurprising that the starter sets of size 1 do not have too large an impact on the Ant-Clique algorithm,

because the neighbourhood subgraph is quite similar to the original graph. It is possible that calculating starter sets of size 1 may have a bigger impact on the performance for other graphs where each vertex does not have as many neighbours.

It is also important to note that Ant-Clique has an element of randomness built in. The default random seed was used for each run of the algorithm. At each step, a neighbouring vertex to the current clique is randomly chosen based on a probability that is proportional to the level of pheromone on the edges between this vertex and the vertices in the current clique [14]. This means that for both the induced common neighbourhood subgraphs and the original graph, the number of cycles required to find a maximum clique may vary depending on the random seed used. For example, if the first few vertices randomly chosen happen to be in a maximum clique of the graph, Ant-Clique will likely find this maximum clique in fewer cycles. To account for this variation, it would be best to run each graph several times with different random seeds and average their cycle results, to see more clearly how much of an impact the symmetry-breaking has on the algorithm.

Table 3 shows the results in the same format using starter sets of size 2. These results are more wide-ranging. For starter sets of size 2, different numbers of induced subgraphs were generated for different graphs. Observe that for $H(4,7)$, the β coefficient indicates a marginally better result using symmetry-breaking, while $H(4,8)$ found the same coclique in far fewer cycles using symmetry-breaking. For $H(4,8)$, despite the fact that there were three common neighbourhood subgraphs to search, the minimum amount of work required to find the same coclique is significantly lower than for the original graph, with a result of $\beta = 5.719$. This shows that symmetry-breaking, while unpredictable, has the potential to significantly increase the efficiency of the search. For $H(10,2)$ and $H(12,2)$, the β coefficient is much smaller because the extra work required to search 9 and 11 common neighbourhood induced subgraphs outweighed the benefit of finding the coclique in fewer cycles than on the original graph.

Graph	*Vertices*	*Size of max coclique for* Γ_{ss_2}	*#cycles for best* Γ_{ss_2}	$\#\Gamma_{ss_2}$	*#cycles on original* Γ	β
$H(4,7)$	2401	304	993	3	3106	1.043
$H(4,8)$	4096	452	1629	3	27949	5.719
$H(10,2)$	1024	512	175	9	235	0.149
$H(12,2)$	4096	2048	331	11	339	0.093

Table 3: Exploration using Ant-Clique, with starter sets of size 2
Γ_{ss_2} = induced common neighbourhood graph for starter sets of size 2

For starter sets of size 3, $H(4,7)$ showed a sizeable improvement from the starter sets of size 2, from $\beta = 1.043$ to $\beta = 3.65$. This is despite the fact that there are now 21 common neighbourhood subgraphs to search. This indicates that the symmetry-breaking algorithm has a significant impact on the performance of Ant-Clique, as it takes far fewer cycles to find the same coclique on the common neighbourhood subgraph than on the original graph.

Graph	*Vertices*	*Size of max coclique for* Γ_{ss_3}	*#cycles for best* Γ_{ss_3}	$\#\Gamma_{ss_3}$	*#cycles on original* Γ	β
$H(4,7)$	2401	307	149	21	11429	3.65
$H(4,8)$	4096	452	1989	21	27949	0.669

Table 4: Exploration using Ant-Clique, with starter sets of size 3
Γ_{ss_3} = induced common neighbourhood graph for starter sets of size 3

The β coefficient for $H(4,8)$ is worse for starter sets of size 3 than for starter sets of size 2, as the extra work of searching the 21 subgraphs outweighs the benefit of finding the coclique faster on the common neighbourhood subgraph.

The number of cycles on the best common neighbourhood subgraph is significantly lower than on the original graph for almost all of these Hamming graphs tested. Even when the extra work of searching each common neighbourhood subgraph is accounted for, symmetry-breaking results in less work for a significant amount of the Hamming graphs. Therefore, symmetry-breaking seems to be beneficial for some graphs. Note that for the $H(d,q)$ graphs where $q = 2$, Ant-Clique finds a maximum coclique in less than 500 cycles. This may be because for these Hamming graphs where $q = 2$, a maximum coclique consists of half of the vertices of the graph, so Ant-Clique is able to find a maximum coclique in very few cycles without using symmetry-breaking. This explains why there is not a sizeable reduction in the work required to find a coclique after performing symmetry-breaking on these graphs, because Ant-Clique did not struggle to find them in the first place. The impact of symmetry-breaking appears to be more pronounced on $H(d,q)$ with $q > 2$, since Ant-Clique does not easily find a maximum coclique in these graphs. In fact, for $H(4,7)$ and $H(4,8)$, the maximum coclique was not found by either the neighbourhood graphs or the original graph. For consistency and because of time constraints, the default number of cycles, 3000, was used for each common neighbourhood graph, and then the original graph was run for as many cycles as necessary to find a coclique of the same size. This was not a sufficient amount of cycles for Ant-Clique to find the maximum coclique of these harder graphs. Although symmetry-breaking decreased the minimum work required to find some large cocliques, the algorithm was still unable to find the maximum coclique, with this amount of cycles.

6 Further discussion and conclusion

6.1 Starter set size and parallelisation

It can be observed from the above results tables that there is not a clear size of starter set that is always optimal, it is dependent on the structure of the graph. Finding the point where the cost of the extra computations required for larger starter set sizes outweighs the benefit of breaking the symmetry further requires experimentation. This is similar to the experimentation required for metaheuristics to find a good balance between accuracy and efficiency. We decided not to generate starter sets larger than size 3, because the number generated grew too large to be feasible for testing, since each common neighbourhood subgraph was run in sequence, and so going beyond size 3 would have been too time-consuming.

However, because of the fact that each Γ_{ss_i} is independent, these subgraphs could be run in parallel, as mentioned in section 4.7. The overall time it takes to search each graph can be reduced by splitting it up and doing each job simultaneously. Therefore, if parallelisation were implemented, forming larger starter sets and searching each subgraph induced by their common neighbourhoods would be far less time-consuming and more feasible. This would be the natural next step to maximise the benefit of symmetry-breaking and make it practical to test the impact of the algorithm with starter sets of sizes larger than 3.

6.2 Other graphs explored

The nature of this project meant that it required a lot of experimentation, most of which is not included in this final report. Several other families of symmetric graphs were explored before settling on the Hamming graphs, including Johnson graphs, Grassmann graphs and Paley graphs. Some of these were rejected because in our preliminary testing, Ant-Clique performed very well on these graphs without symmetry-breaking, because the cliques in these graphs were very small. However at this stage of the project, the focus was on cliques, not cocliques, and so it is possible that symmetry-breaking would have more of an impact on finding cocliques of these rejected families of graphs.

In addition to this, in an earlier version of the common neighbourhood finding function, available in Appendix A.2, the Γ_{ss_i} were not written to the file correctly, which made our implementation less efficient. Optimising the algorithm by changing how the Γ_{ss_i} were written to the file meant that all previously gathered data had to be discarded. This reduced the amount of results that could be included in this report.

Two of the graphs in the Second DIMACS Implementation Challenge were tested, keller6 and Hamming10_4. Note that this Hamming10_4 graph is not $H(10, 4)$ i.e. it is not one of the $H(d, q)$ Hamming graphs used as the case study for our main testing, rather it is one of the $H_q(n, d)$ graphs described in section 6.3.

Starter set size	*Vertices*	*Size of max coclique for* Γ_{ss_i}	*#cycles on* Γ_{ss_i}	$\#\Gamma_{ss_i}$	*#cycles on original* Γ	β
1	1024	37	730	1	746	1.022
2	1024	40	614	7	763	0.178

Table 5: Hamming10_4 results

Ant-Clique performed well on this graph without symmetry breaking, it found a clique of 40, the size of the maximum clique, in less than 800 cycles, so the impact of symmetry-breaking is minimal.

keller6 is not vertex-transitive, so there were 20 common neighbourhood induced subgraphs created just for starter sets of size 1. Therefore it was not practical for this project to go beyond starter sizes of size 1 for this graph.

Starter set size	*Vertices*	*Size of max coclique for* Γ_{ss_i}	*#cycles on* Γ_{ss_i}	$\#\Gamma_{ss_i}$	*#cycles on original* Γ	β
1	3361	57	2922	20	22720	0.389

Table 6: keller6 results

The β coefficient for this test is less than 1, indicating that the minimum amount of work required to find this clique is higher when using symmetry-breaking. However, comparing directly the number of cycles taken, it took less than $3,000$ cycles to find this clique in one of the neighbourhood graphs, whereas it took over $22,000$ on the original graph. So, although symmetry-breaking does increase the minimum work required to find a maximum clique in the graph, it could make it more practical to find cliques if the neighbourhood graphs were run in parallel. It takes much fewer cycles to find it on one of the common neighbourhood subgraphs than on the original graph, so the amount of time taken to arrive at this result would be lower.

6.3 Conclusion

Overall the results of our project are promising. The Ant-Clique algorithm we used already obtained good results for the Maximum Clique Problem, and we managed to see an improvement from using our symmetry-breaking algorithm on several graphs. The following is a list of directions we would take if we were to delve further into this topic.

Code improvements: We would spend time optimising both the performance of our symmetry-breaking algorithm and the mechanisms of translating between the SageMath graphs and the C code of Ant-Clique. As mentioned in section 6.4, the starter sets themselves were included in the common neighbourhood subgraphs given to Ant-Clique. Upon reflection, although the starter sets only contained 1, 2 or 3 vertices, by definition they are adjacent to every vertex in the induced subgraph, and so by adding them a lot of unnecessary edges were included in the induced subgraphs given to the Ant-Clique algorithm. If we were to continue testing this algorithm, we would change the code to not include the starter sets in the induced subgraph, to improve the overall efficiency.

Further Exploration of Ant-Clique: We would also explore further the parameters of Ant-Clique, since we used the default parameters in our testing. Because of the limited time frame, it was necessary to keep all of the variables constant except the graphs, but there are many different parameters that are adjustable, such as the pheromone coefficients, the number of ants etc. Adjustment of these parameters could allow larger cliques to be found in fewer cycles, but would require extensive experimentation, which as discussed, is a drawback of using metaheuristics in general.

Implementation of Parallelisation: We would implement parallelisation to make it

more feasible to search several common neighbourhood induced subgraphs simultaneously in order to maximise the benefit of performing symmetry-breaking. This would allow us to run the $H(4,7)$ and $H(4,8)$ graphs for enough cycles to potentially find the maximum cocliques that were not found during this project.

Exploration of different graphs: We would particularly like to focus on symmetric graphs for which the clique number is currently unknown, to see if by combining symmetry-breaking and ACO we could find new cliques.

A SageMath Code

A.1 Symmetry-breaking function

```
#Does symmetry breaking on G, and returns starter sets of size k
def symmetry_break(G, k):
    #Holds sets currently being worked on
    queue = []
    #Holds found starter sets of size k
    final_ss = []
    automorph = G.automorphism_group() #automorphism group of G
    orbits = automorph.orbits() #orbits of aut(G)
    for i in range(len(orbits)):#Add a list with a representative of each orbit to the queue
        queue.append([orbits[i][0]])
    if k ==1: #If starter set of size one, just return list of representatives of the orbits
        final_ss = queue
        return final_ss
    else:
        while queue: #while there is something in the queue
            #Pop off top element of the queue and find its setwise stabiliser
            current_ss = queue.pop()
            stabilizer = automorph.stabilizer(current_ss,action= "OnSets")
            stab_orbits = stabilizer.orbits()
            #Iterate through the orbits of the stabiliser of the current starter set
            for i in range(len(stab_orbits)):
                #Check that the rep of the current orbit is not in the starter set, and that the length of the starter set is less than k
                if (stab_orbits[i][0] not in current_ss) and (len(current_ss)<k):
                    #Check if the first element of the current orbit is connected to everything in the current starter set
                    if (all(G.has_edge(stab_orbits[i][0], w) for w in current_ss)):
                        new_ss = current_ss + [stab_orbits[i][0]] #If it is, add this vertex to the starter set
                        #Sort the starter set so that it appears the same as any previous versions in a different order
                        new_ss.sort()
                        #If the newly formed starter set is of length k and not in the list of final starter sets, add it
                        if len(new_ss)==k and (new_ss not in final_ss):
                            final_ss.append(new_ss)
                        queue.append(new_ss) #Add the newly formed starter set to the queue
    return final_ss #Return the list of starter sets of length k
```

A.2 Common neighbourhood finding function

```
from pathlib import Path
import shutil
#Finds the induced subgraphs of common neighbourhoods of each starter set
#of size k and writes each one to a file in DIMACS format
#c has possible values "clique" or "coclique"
def write_neighbourhood_ss(G, k, c):
    #If looking for cocliques, get complement of the graph and find
    ↪  cliques
    if c == "coclique":
        G = G.complement()

    #If user provides c that is not "clique" or "coclique" give message
    elif c not in ("clique", "coclique"):
        print(f"Error {c} is not clique or coclique")
        return

    counter = 0  #Initialise counter for number of neighbourhoods
    #Create folder to store files of nbhds
    folder = Path("Nhbds_of_" + str(G) + "_" + str(k) + "_" + c)

    #Run symmetry breaking algorithm on the Graph to return the
    #starter sets of size k and save this list to the variable l
    l = symmetry_break(G, k)
    if not l: #Check if list is empty
        print(f"{G} has no {c}s of size {k}")
        return
    v = G.vertices() #Get list of vertices of the graph

    for i in range(len(l)): #Iterate through the list of starter sets
        nbhd = [] #Create list to store the neighbours of each starter set
        #Iterate through the vertices of the graph
        for j in range(len(v)):
            #if the vertex is not in the neighbourhood and it is connected
            ↪  to everything in the starter set
            if j not in l[i] and all(G.has_edge(j, s) for s in l[i]):
                #Add it to the neighbourhood
                nbhd.append(j)

        #If there are no common neighbours for the elements of the starter
        ↪  set, print a message
        if not nbhd:
            print(f"no neighbourhood found for {l[i]}")

        else:
            counter +=1 #Increment the neighbourhood counter
            for k in l[i]:
                nbhd.append(k) #Add the starter set to its neighbourhood
            sub = G.subgraph(nbhd) #Create a subgraph of the neighbourhood
            relabel_vertices(sub)#Relabel the vertices of the subgraph

            folder.mkdir(parents=True, exist_ok=True)#Open the folder path
            #Create a file for the current neighbourhood
```

```
            file_path = folder / ("nbhd_" + str(counter) + "_s")
            #Write the subgraph to the file in DIMACS format
            file_path.write_text(convert(sub))

    print(f"Wrote {counter} neighbourhoods to the folder {folder.name}")
```

Glossary

A

Automorphism Group

The automorphism group of a graph is the group of graph automorphisms, with function composition as the group operation. 11–13, 24, 27

C

Clique

A clique is a subset C of the set of vertices V in the graph Γ, such that each pair of distinct vertices in C are adjacent i.e. they are connected by an edge. 1

Coclique

A coclique is a subset I of the set of vertices V in Γ, where all pairs of vertices in I are not adjacent to each other i.e. no vertices in the subset are connected by an edge. 3

Common Neighbourhood

The common neighbourhood of a set $S \subseteq V$ is $\Gamma(S) = \bigcap_{v \in S} \Gamma(v)$. 12

Complement Graph

The complement graph of a graph Γ has the same set of vertices V as Γ, with two vertices being adjacent in $\overline{\Gamma}$ if and only if they are not adjacent in Γ. 3, 20, 26

D

Distance-regular

A graph where for any two arbitrary vertices x and y that are distance h apart, the number of vertices that are at distance i from x and the number of vertices at distance j from y is fixed and depends only on the choice of h, i and j. 24

Distance-transitive

A graph Γ is distance-transitive if for all vertices u, v, x, y of Γ where the distance between u and v is the same as the distance between x and y, then there exists an automorphism f such that $f(u) = x$ and $f(v) = y$. 12, 24

E

Edge-transitive

A graph is edge-transitive if any edge can be mapped by an automorphism to any other edge in the graph. 12

G

Graph

A graph $\Gamma = (V, E)$, has a finite set of vertices $V(\Gamma)$ and set $E \subseteq V \times V$ of edges between vertices. 1

Graph Automorphism

A graph automorphism is a graph isomorphism from a graph onto itself. 11

Graph Isomorphism

Given two graphs Γ_1 and Γ_2, an isomorphism is a bijective mapping $f : V(\Gamma_1) \to V(\Gamma_2)$ such that $f(a)$ and $f(b)$ are adjacent in Γ_2 if and only if a and b are adjacent in $\Gamma_1 \quad \forall \ a, b \in V(\Gamma_1)$. 11

Group

A group is a set G which has an operation * that satisfies the below conditions:
1. The operation * is associative
2. There is an identity element 1 such that

$$1 * a = a = a * 1 \qquad \forall\, a \in G$$

3. Each element $a \in G$ has an inverse $a^{-1} \in G$ such that

$$a * a^{-1} = 1 = a^{-1} * a.$$

11

Group Action

Given a group G and a set S, a group action is a map from $G \times S$ to S, which is denoted by s^g, where $s^1 = s$ and

$$s^{(g_1 g_2)} = (s^{g_1})^{g_2} \qquad \forall \ g_1,\ g_2 \in G \ \textit{and} \ s \in S.$$

11, 14, 16

I

Induced Subgraph

An induced subgraph of a graph Γ is a subgraph created from a subset of the vertex set of Γ, where an edge is added to the subgraph if both endpoints of the edge are in this subset. 12, 19, 22

M

Maximal Clique

A maximal clique is a clique that cannot be extended into a larger clique by adding an adjacent vertex. 1

Maximum Clique

A maximum clique is the clique with the largest number of vertices in the graph. 1

Maximum Clique Problem

The Maximum Clique Problem (MCP) is an optimisation problem where the objective is to find the clique with the largest number of vertices in a graph. 1

N

Neighbourhood

The neighbourhood $\Gamma(x)$ of a vertex x in a graph Γ is the set of all vertices that are adjacent to x. 12

O

Orbit

Given an element $x \in S$ and a group G acting on S, the orbit of x is the equivalence class of the form

$$x^G = \{x^g : g \in G\}.$$

11, 13, 24, 27

P

Permutation Group

A permutation group is a group where each element is a bijection from a set S onto itself. 11

S

Stabiliser

The stabiliser of x in G is the set

$$G_x = \{g \in G : x^g = x\}$$

where G is a group and x is an element of a set S that G acts on. 11–13

Strongly Regular Graph

A graph that has parameters (v,k,λ,μ) where v is the number of vertices, k is the common degree between all vertices, λ is the number of common neighbours between every pair of adjacent vertices and μ is the number of common neighbours between every pair of non-adjacent vertices, where λ, $\mu \geq 0$. 12, 19, 20

T

Transitive

A group G acts transitively on S if for all elements $x, y \in S$, there must exist $g \in G$ such that $x^g = y$. 12

V

Vertex Cover

A vertex cover of a graph Γ is a subset V' of the set of vertices in the graph such that every edge of the graph contains at least one vertex in V'. 3

Vertex-Transitive

A graph is vertex-transitive if any vertex can be mapped by an automorphism to any other vertex in the graph. 12, 27

References

[1] Darij Grinberg. “An introduction to graph theory”. In: (2025). URL: https://www.cip.ifi.lmu.de/~grinberg/t/22s/graphs.pdf.

[2] Robin J. Wilson. *Introduction to Graph Theory*. 4th ed. Addison Wesley Longman Limited, 1996.

[3] J. W. Moon and L. Moser. “On cliques in graphs”. In: *Israel Journal of Mathematics* (1965). DOI: https://doi.org/10.1007/BF02760024.

[4] Steven S. SKiena. *The Algorithm Design Manual Second Edition*. 2nd ed. Springer, 2008, pp. 316, 325–328, 344, 350. DOI: https://doi.org/10.1007/978-1-84800-070-4.

[5] Janos Barta and Roberto Montemanni. “The Maximum Clique Problem for Permutation Hamming Graphs”. In: *Journal of Optimization Theory and Applications* (2022). DOI: https://doi.org/10.1007/s10957-022-02035-w?urlappend=%3Futm_source%3Dresearchgate.net%26utm_medium%3Darticle.

[6] John David Eblen. “The Maximum Clique Problem: Algorithms, Applications, and Implementations.” PhD thesis. University of Tennessee, 2010. URL: https://trace.tennessee.edu/utk_graddiss/793.

[7] Divya Tiwari. “Clique Problem in Social Network Analysis—Community Detection”. In: *medium.com* (2025). URL: https://medium.com/@23bt04029/clique-problem-in-social-network-analysis-community-detection-044c3dd98974.

[8] Butenko Sergiy Pardalos Panos Sergienko Ivan Shylo Vladimir Stetsyuk Petro. “Finding maximum independent sets in graphs arising from coding theory”. In: *Proceedings of the 2002 ACM symposium on Applied computing*. 2002. URL: https://doi.org/10.1145/508791.508897.

[9] R.D. Luce and A.D. Perry. “A method of matrix analysis of group structure”. In: *Psychometrika* (1949), pp. 95–116. DOI: https://doi.org/10.1007/BF02289146.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, 1979.

[11] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proc. Third ACM Symp. Theory of Computing* (1971), pp. 151–158. DOI: https://doi.org/10.1145/800157.805047.

[12] DIMACS. *DIMACS Implementation Challenges*. URL: http://dimacs.rutgers.edu/archive/Challenges/. (accessed: 15.02.2026).

[13] Sean Luke. *Essentials of Metaheuristics*. George Mason University, 2016.

[14] Christine Solnon and Serge Fenet. “Investigating ACO capabilities for solving the Maximum Clique Problem”. In: *University Lyon I* (2004). URL: https://perso.liris.cnrs.fr/christine.solnon/publications/rr-mai04.pdf.

[15] Kapil Choudhary. “Optimization Algorithms - Classics and Recent Advances”. In: IntechOpen, 2024. Chap. Ant Colony Optimization-Based Models for Agriculture Price Forecasting: Innovations, Case Studies, and Future Prospects.

[16] James Ostrowski et al. “Orbital branching”. In: *Math. Program.* (2009). DOI: https://doi.org/10.1007/s10107-009-0273-x.

[17] James S. Milne. *Group Theory (v4.01)*. Available at www.jmilne.org/math/. 2025.

[18] David S Dummit and Richard M Foot. *Abstract Algebra Third Edition*. John Wiley and Sons, Inc, 2004.

[19] Norman Biggs. *Algebraic Graph Theory*. 2nd ed. Cambridge University Press, 1993, p. 118.

[20] Andries E. Brouwer and Hendrik Van Maldeghem. *Strongly regular graphs*. University Printing House, Cambridge, 2022.

[21] M. Saravanana and KM. Kathiresan. "On the Independence Graph of Hamming Graph". In: *Iranian Journal of Mathematical Sciences and Informatics Vol. 20, No. 1 (2025), pp 125-130* (2025). DOI: 10.61186/ijmsi.20.1.125. URL: https://www.researchgate.net/publication/391949236_On_the_Independence_Graph_of_Hamming_Graph.

[22] Koolen J H van Dam E R and Tanaka H. "Distance-regular graphs". In: *Electronic Journal of Combinatorics, vol. Dynamic Surveys, DS22, pp. 1-156* (2016). URL: http://www.combinatorics.org/ojs/index.php/eljc/article/view/DS22.

[23] Raymond Hill. *A First Course in Coding Theory*. Oxford, 1986. URL: https://sites.math.rutgers.edu/~zeilberg/akherim/RHcoding.pdf.

[24] Salim Y. El Rouayheb et al. *Bounds on Codes Based on Graph Theory*. 2008. arXiv: 0806.4979 [cs.IT]. URL: https://arxiv.org/abs/0806.4979.