

1-1-2011

Cops and robbers on graphs and hypergraphs

William David Baird
Ryerson University

Follow this and additional works at: <http://digitalcommons.ryerson.ca/dissertations>



Part of the [Applied Mathematics Commons](#)

Recommended Citation

Baird, William David, "Cops and robbers on graphs and hypergraphs" (2011). *Theses and dissertations*. Paper 821.

This Thesis is brought to you for free and open access by Digital Commons @ Ryerson. It has been accepted for inclusion in Theses and dissertations by an authorized administrator of Digital Commons @ Ryerson. For more information, please contact bcameron@ryerson.ca.

COPS AND ROBBERS ON GRAPHS AND HYPERGRAPHS

by

William David Baird, B.Sc. Wilfrid Laurier University, 2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Applied Mathematics

Toronto, Ontario, Canada, 2011

©Copyright by William David Baird, 2011

I hereby declare that I am the sole author of this thesis or dissertation.

I authorize Ryerson University to lend this thesis or dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis or dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

ABSTRACT

COPS AND ROBBERS ON GRAPHS AND HYPERGRAPHS

Master of Science, 2011

William Baird

Applied Mathematics

Ryerson University

Cops and Robbers is a vertex-pursuit game played on a graph where a set of cops attempts to capture a robber. Meyniel's Conjecture gives an asymptotic upper bound on the *cop number*, the number of cops required to win on a connected graph. The incidence graphs of affine planes meet this bound from below, they are called *Meyniel extremal*. The new parameters m_k and M_k describe the minimum orders of k -cop-win graphs. The relation of these parameters to Meyniel's Conjecture is discussed. Further, the cop number for all connected graphs of order 10 or less is given. Finally, it is shown that cop win hypergraphs, a generalization of graphs, cannot be characterized in terms of retractions in the same manner as cop win graphs. This thesis presents some small steps towards a solution to Meyniel's Conjecture.

ACKNOWLEDGMENTS

I would like to thank the many people at Ryerson who have helped me in completing this work. The Faculty and Staff in the Department of Mathematics have contributed in innumerable small ways to this end product. In particular I would like to thank Steve Kanellis for his technical guidance and patience. I would also like to thank Dr. Pavel Prałat for his guidance in the area of computer programming. Special thanks also go to Dr. Peter Danziger and Dr. Dejan Delić for being part of my thesis committee and their part in my mathematical development.

My deepest thanks go to my supervisor Dr. Anthony Bonato. From early on in my undergraduate career Dr. Bonato has been a driving force in my pursuit of mathematics. His love of the discipline and his commitment to excellence are an inspiration to me. Without his guidance, wisdom and mentorship this thesis would not have been possible.

I am also deeply indebted to my family and friends. My parents Eric and Rose-Mary, and my sister Emily have all been a constant source of support and inspiration. I also extend my thanks to my girlfriend Shelby, for her ceaseless encouragement and support. Finally, I wish to thank my friends who have encouraged me through this entire process.

Contents

List of Figures	vii
Chapter 1. Introduction	1
1.1. Introduction to Cops and Robbers	1
1.2. Graphs	3
1.3. Incidence Graphs	7
1.4. Asymptotic Notation	9
1.5. Outline of Thesis	10
Chapter 2. Meyniel Extremal Graphs	13
2.1. Steiner 2-designs	14
2.2. Projective Planes	15
2.3. Affine and partial affine planes	19
Chapter 3. Growth Rates and Minimum Orders of k -Cop-Win Graphs	25
3.1. Introduction	25
3.2. Minimum orders and growth rates	26
3.3. Algorithms for computing cop number	29
3.4. The k -cop-win Algorithm	39
3.5. Lower bounds	43
Chapter 4. The Game of Cops and Robbers on Hypergraphs	49

4.1.	Introduction	49
4.2.	Definitions and Notation	50
4.3.	Cops and Robbers on hypergraphs	52
Chapter 5.	Conclusions and Future Work	57
5.1.	Summary	57
5.2.	Open Problems	58
Appendix A.	Source Code	61
A.1.	Cop-win Checker	63
A.2.	Cops and Robbers Game Simulator	70
A.3.	2-cop-win Checker	84
Bibliography		99

List of Figures

1.1	A cop-win graph, G .	2
1.2	A 2 cop-win graph, C_4 .	2
1.3	A graph G .	4
1.4	A subgraph of G .	4
1.5	An induced subgraph of graph G .	4
1.6	A corner in a graph.	6
1.7	An example of a dominating set in a graph.	6
1.8	A hypergraph.	7
1.9	The Fano plane.	8
1.10	The Incidence graph of the Fano plane.	9
2.1	The incidence graph of a projective plane of order q .	17
2.2	The robber has $q + 1$ escape routes.	19
2.3	The $q + 1$ cops can dominate the position of the robber.	20
2.4	The robber again has $q + 1$ escape routes.	21
3.1	The Petersen Graph.	30
3.2	The graph G , a cop-win graph.	35
3.3	A cop-win ordering for the graph G .	36

3.4	The cop computes her second-to-last move, $F_2(a) = b$, and so moves to b .	38
3.5	The graph G , a cop-win graph.	40
3.6	The construction of G' .	44
3.7	The isomorphism $h : G' \rightarrow J'$.	45
4.1	A hypergraph with hyperedges of cardinality 3 and 5.	49
4.2	The vertex a is internal, the vertex b is external.	50
4.3	A 5-regular 2-joined hypergraph.	50
4.4	The graph $G(H)$ for H , the hypergraph in Figure 4.1.	51
4.5	A hyperpath.	51
4.6	A hypercycle.	52
4.7	The deleted hyperedges give the robber an escape route.	56

CHAPTER 1

Introduction

1.1. Introduction to Cops and Robbers

A well-known, anonymous quote is “Philosophy is a game with objectives but no rules. Mathematics is a game with rules but no objectives”. *Cops and Robbers* is a game within Mathematics with both rules and objectives. Cops and Robbers is a *vertex-pursuit* game played on a graph for reasons that will be more clear as we give the rules for the game.

The game is played as follows. First the players, a *cop* and a *robber*, each select a vertex of a graph on which to begin the game. The cop selects the first vertex. In alternating rounds the cop and the robber take turns moving from vertex-to-vertex along edges. The object of the game for the cop is to occupy the same vertex as the robber, while the object for the robber is to prevent this from happening. We say the cop has *captured* the robber if the cop occupies the same vertex as the robber. A graph is said to be *cop-win* if after some finite number of rounds a cop can move so as to capture the robber. Otherwise, the graph is *robber-win*. Cop-win graphs, such as the graph in Figure 1.1, have a well known characterization based on the successive deletion of corners. This characterization will be discussed later in Section 3.3 of Chapter 3.

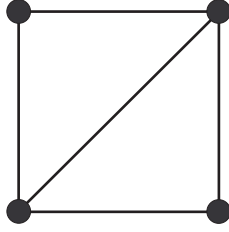


FIGURE 1.1. A cop-win graph, G .

Cops and Robbers can be played with any number of cops. For $k > 0$ an integer, if k cops are sufficient to capture a robber on a given graph, then we say that graph is k -cop-win. For example, the graph in Figure 1.2 is not cop-win. If the game is played on this graph with just one cop, then the robber can evade the cop indefinitely. However, if the game is played by two cops, then the cops win since there is no way for the robber to escape.

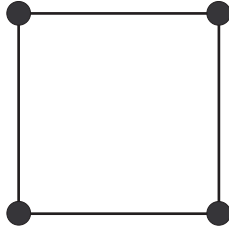


FIGURE 1.2. A 2 cop-win graph, C_4 .

A 1-cop-win graph is simply a cop-win graph. We define the *cop number* of a graph G , written $c(G) = k$, to be the minimum number of cops required for the graph G to be k -cop-win. For example a cop-win graph, like the graph in Figure 1.1, has $c(G) = 1$. The graph in Figure 1.2 satisfies $c(C_4) = 2$. This parameter is well-defined since placing a cop on every vertex will always result in a win (in one move) for

the cops. This fact gives a trivial upper bound on the cop number: $c(G) \leq |V(G)|$.

Cops and Robbers was introduced by Nowakowski and Winkler in [15] and independently by Quilliot in his Ph.D. Thesis [18]. Both sets of authors give a characterization of cop-win graphs. Later on, Aigner and Fromme [1] considered the game played with multiple cops.

Cops and Robbers belongs to a family of combinatorial games called *vertex-pursuit games* or *graph searching games*. These games provide a simplistic model for network security. We can think of the robber as an intruder in a network and cops as network monitors.

1.2. Graphs

The graph on which Cops and Robbers is played is crucial to the outcome of the game. In this section we give preliminary definitions and terminology that will be useful throughout the remaining chapters. A *graph* G consists of a non-empty set of vertices, written $V(G)$, and a set of edges, written $E(G)$. Edges are unordered pairs of vertices. If two vertices are connected by an edge, then we say they are *adjacent*. If u and v are adjacent, then we write $uv \in E(G)$ or $u \sim v$. The *order* of G is the number of vertices, $|V(G)|$. A graph is *reflexive* if all vertices have a loop. We will assume all our graphs are reflexive as this allows the cops and robbers to *pass* or stay on the same vertex, although we will omit reference to loops in figures. All the graphs we consider are of finite order.

A *subgraph* H of a graph G is a graph such that the vertices of H are a subset of the vertices of G and the edges of H are a subset of the

vertices of G . An *induced subgraph* J is a graph with vertices a subset of the vertices of G and containing all edges of G with both ends in the vertex set of J . That is, an edge uv is in the edge set of J if and only if uv is in the edge set of G and both u and v are vertices in J . Figure 1.3 shows a graph G , Figure 1.4 shows a subgraph of G , and Figure 1.5 shows an induced subgraph of G .

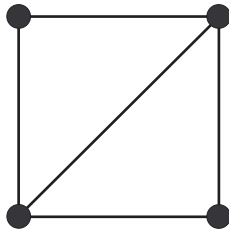


FIGURE 1.3. A graph G .

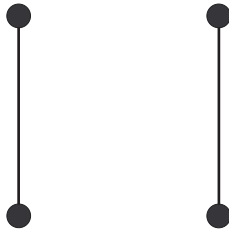


FIGURE 1.4. A subgraph of G .

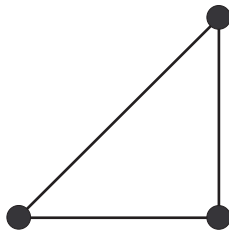


FIGURE 1.5. An induced subgraph of graph G .

The *degree* of a vertex u in a graph G is the number of edges incident with it, and is written $\deg_G(u)$ (we drop the subscript G if it is clear from context). The *maximum degree* of a graph G , denoted $\Delta(G)$, is the degree of a vertex with largest degree in G . Similarly, the *minimum degree* of G , denoted by $\delta(G)$, is the degree of the vertex with the smallest degree in G . A graph is said to be *k-regular* if all vertices in the graph have degree k .

A *path* of order n , written P_n , is a graph with vertices $v_1, v_2, \dots, v_n$, so that v_i is adjacent to v_{i+1} , where $1 \leq i \leq n-1$. A *cycle*, written C_n , is formed by adding the edge v_1v_n in the path P_n . The graph in Figure 1.2 is a cycle. The *girth* of a graph, written $g(G)$, is the order of a smallest cycle contained in G as a subgraph. Aigner and Fromme in [1] give the following lower bound on cop number using girth and minimum degree.

THEOREM 1. *Let G be a graph of minimum degree $\delta(G) \geq n$ with girth $g(G) \geq 5$. Then $c(G) \geq n$.*

A graph is *connected* if for all vertices u and v there exists a path from u to v . Usually when playing the game of Cops and Robbers we consider graphs to be connected.

The *neighbour set* of a vertex v is the set of all vertices adjacent to v , written $N(v)$. The *closed neighbour set* of a vertex v is the union of $N(v)$ and v itself, denoted $N[v]$. A vertex u is a *corner* if there is a vertex v such that $N[u] \subseteq N[v]$. Figure 1.6 shows a corner in a graph. We say that a vertex v *dominates* or *covers* u .

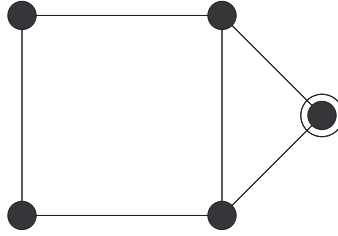


FIGURE 1.6. A corner in a graph.

A *dominating set* S of a graph G is a set of vertices such that all vertices of G are either in S or adjacent to a vertex in S . The *domination number*, written $\gamma(G)$, is the minimum number of vertices required for a dominating set in G . Note that $c(G) \leq \gamma(G)$, as placing a cop on each vertex of a dominating set ensures a win for the cops. The circled vertices in Figure 1.7 show a dominating set in a graph.

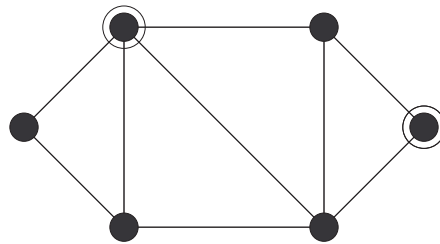


FIGURE 1.7. An example of a dominating set in a graph.

A graph G is called *bipartite* if the vertices of G can be partitioned into two colour classes, say *red* and *blue*, such that no red vertex is adjacent to any other red vertex and no blue vertex is adjacent to any other blue vertex. Alternatively, a bipartite graph is one whose vertex set is partitioned into sets of vertices containing no edges between them. We call such sets of vertices *independent sets*.

Hypergraphs are a generalization of graphs. A *hypergraph* H consists of a non-empty set of vertices, written $V(H)$, and a set of edges (or hyperedges), written $E(H)$. Edges are sets of vertices of any cardinality. The *order* of an edge in a hypergraph is the number of vertices contained in the edge. We will discuss hypergraphs further in Chapter 4.

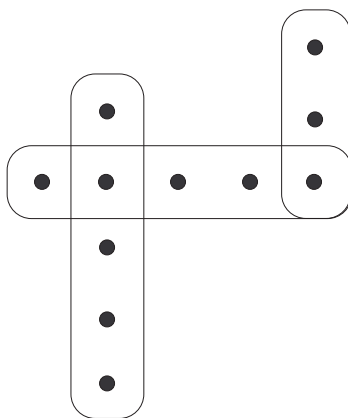


FIGURE 1.8. A hypergraph.

1.3. Incidence Graphs

An *incidence structure* consists of a set of points and lines, denoted P and L respectively, together with an *incidence relation* of pairs of points and lines. Incidence structures are generalizations of familiar geometric structures such as planes.

We will deal with several specific incidence structures. A *Steiner 2-design* or a $2-(v, k, 1)$ *design* is an incidence structure with v -many points, with lines (sometimes called *blocks*) containing k points, such that any pair of points is contained in exactly one block. It is assumed

that $2 < k < v$. A *finite projective plane* of order n is a Steiner 2-design on $n^2 + n + 1$ points, that has blocks of size $n + 1$. Projective planes are then $2-(n^2 + n + 1, n + 1, 1)$ designs. They can also be defined as a set of $n^2 + n + 1$ points with the properties that any two points determine a unique line and any two lines determine a unique point. Note that every point has $n + 1$ lines incident with it, and every line contains $n + 1$ points. The smallest example of a projective plane is the *Fano plane* (see Figure 1.9). It is a projective plane of order $n = 2$. Projective planes are known to exist if the order $n = p^k$, where p is prime, and $k \geq 1$. All known projective planes have such orders. In fact it is conjectured that only the only possible orders for projective planes are prime powers (see [6]).

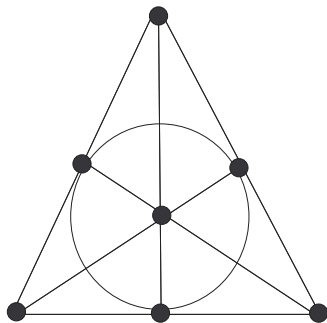


FIGURE 1.9. The Fano plane.

An *affine plane* of order n is a $2-(n^2, n, 1)$ design. An affine plane of order n exists if and only if a projective plane of order n exists. A *partial affine plane* is an incidence structure obtained from an affine plane by the deletion of one or more lines. An *incidence graph* is a graph that is constructed from an incidence structure as follows. The vertices of the graph are the points and lines, and there is an edge between two

vertices if they are incident with each other; that is, there is an edge between a vertex representing a point and a vertex representing a line if and only if the point is on the line. Incidence graphs are bipartite since no points are incident with any other points and no lines are incident with any other lines hence, the graph is the union of two independent sets. Figure 1.10 shows the incidence graph of the Fano plane from Figure 1.9.

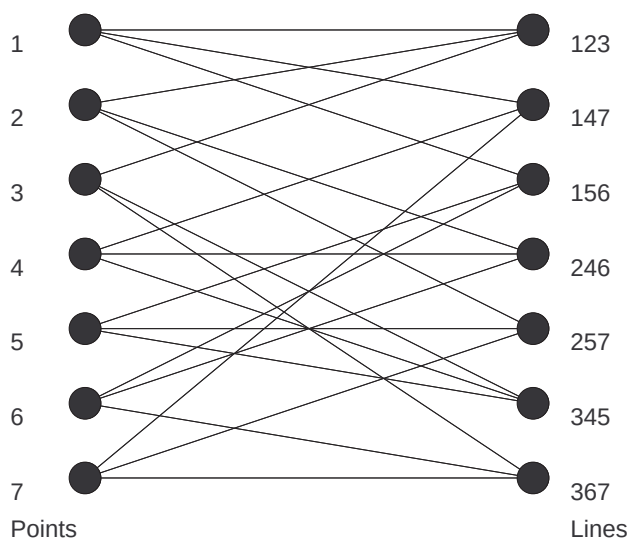


FIGURE 1.10. The Incidence graph of the Fano plane.

1.4. Asymptotic Notation

Asymptotic notation gives a simple and succinct method for describing how a function $f(x)$ behaves for large values of x . This notation is especially useful for describing the running time of an algorithm, though we will also use it to describe the asymptotic behaviour of other functions. Let $f(x)$ and $g(x)$ be real-valued functions. We

write $f(x) \in O(g(x))$ if and only if there exists a constant $c > 0$ and an integer $N > 0$ such that for $x > N$, $f(x) \leq cg(x)$. Equivalently, $f(x) \in O(g(x))$ if

$$\limsup_{x \rightarrow \infty} \frac{f(x)}{g(x)}.$$

exists and is finite. It is a common abuse of notation to write $f(x) = O(g(x))$ or simply $f = O(g)$. If $f = O(g)$, then $g = \Omega(f)$. If $f = O(g)$ and $f = \Omega(g)$, then $f = \Theta(g)$. We say that $f = o(g)$ if

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

If $f = o(g)$, then $g = \omega(f)$. Equivalently, $g = \omega(f)$ if

$$\lim_{x \rightarrow \infty} \frac{g(x)}{f(x)} = \infty.$$

Note that if $f = o(1)$, then

$$\lim_{x \rightarrow \infty} f(x) = 0.$$

We say f and g are asymptotically equivalent, written $f \sim g$ if

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 1.$$

1.5. Outline of Thesis

Perhaps the deepest open problem in the study of Cops and Robbers is *Meyniel's Conjecture*. The conjecture, due to Henri Meyniel [11], is that for a connected graph G ,

$$c(G) = O(\sqrt{|V(G)|}).$$

In Chapter 2, we give some background on Meyniel's Conjecture, discuss some results related to the conjecture and give the cop number of incidence graphs of projective and partial affine planes. We introduce so-called Meyniel extremal graphs which realize the upper bound in the conjecture, and give infinitely many new examples of such families satisfying certain regularity conditions. In Chapter 3 we consider the minimum orders of a k -cop-win graph and their relation to Meyniel's Conjecture. We also present experimental results about the cop number of connected graphs of order 10 or less, as well as the algorithms used to obtain these results. The implementations of the algorithms used in this chapter are included as an appendix. Chapter 4 discusses the game of Cops and Robbers played on hypergraphs and its relationship to the game played on graphs. Our final chapter summarizes open problems in the area and describes some future directions.

CHAPTER 2

Meyniel Extremal Graphs

Difficult mathematical challenges often serve to motivate study in a particular area. For Cops and Robbers, one such problem is Meyniel's Conjecture. The conjecture states that if G is a graph of order n , then

$$c(G) = O(\sqrt{n}).$$

Equivalently, for n sufficiently large there exists a constant $d > 0$ such that

$$c(G) \leq d\sqrt{n}.$$

Although it was first introduced by Frankl in 1987 [11] the problem went largely unstudied for many years. In this initial paper Frankl proved that $c(G) = o(n)$; In particular, he proved that if G is a graph of order n , then

$$c(G) = O\left(n \frac{\log \log n}{\log n}\right).$$

This bound has been improved upon incrementally since the introduction of the topic. The best known upper bound on the cop number is the following, which was recently proven independently by three sets of researchers.

THEOREM 2. [13, 12, 21] For a graph G of order n ,

$$(2.1) \quad c(G) \leq O\left(\frac{n}{2^{(1-o(1))\sqrt{\log_2 n}}}\right).$$

Note that although the bound in (2.1) is currently the best known upper bound for general graphs, it is still far from proving Meyniel's conjecture.

In the search for proof or disproof of the conjecture it is interesting to see if there are graphs that meet the bound predicted by Meyniel's Conjecture from below. These graphs would be, in a sense, the graphs with largest possible cop number relative to their order. We present here a new infinite family of graphs, so-called Meyniel Extremal graphs, that have cop number meeting the Meyniel bound from below. To make this precise, an infinite family of graphs $(G_n : n \geq 0)$ is *Meyniel extremal* if there is a constant $d > 0$ such that for sufficiently large n ,

$$c(G_n) \geq d\sqrt{|V(G_n)|}.$$

2.1. Steiner 2-designs

In order to describe Meyniel extremal families, we need some background from combinatorial designs. If t, v, k and λ are positive integers, then a t -(v, k, λ) *block design* or a *t-design*, is an incidence structure with the following properties. It contains v points, each block contains k points, and every set of t points is contained in exactly λ blocks. A Steiner 2-design is a t -design in which and two points uniquely determine a block. Thus, Steiner 2-designs are 2 -($v, k, 1$) designs. For a

general t -design, with b blocks, it can be shown that (see [6])

$$b \binom{k}{t} = \lambda \binom{v}{t}.$$

In a Steiner 2-design this gives that

$$b \binom{k}{2} = \binom{v}{2}.$$

Rearranging yields

$$(2.2) \quad b = \frac{\binom{v}{2}}{\binom{k}{2}} = \frac{v(v-1)}{k(k-1)}.$$

Thus, the number of blocks in a Steiner 2-design is $\frac{v(v-1)}{k(k-1)}$. Let r be the number of blocks incident with each point, also called the *replication number*. It can be shown that

$$(2.3) \quad r = \frac{v-1}{k-1}.$$

Thus, each point is incident with $\frac{v-1}{k-1}$ blocks.

2.2. Projective Planes

With the notation of the previous section we can describe projective planes. *Projective planes* of order q are Steiner 2-designs with $q^2 + q + 1$ points and $q + 1$ points on a line. Thus, projective planes are $2 - (q^2 + q + 1, q + 1, 1)$ designs. Projective planes are also often defined as a set of $q^2 + q + 1$ points with the following properties.

- (1) There is exactly one line incident with every pair of distinct points.
- (2) There is exactly one point incident with every pair of distinct lines.
- (3) There are four points such that no line is incident with more than two of them.

Note that the second axiom means that any pair of distinct lines intersect and thus, there are no parallel lines in projective planes. From the definition we know that projective planes contain $q^2 + q + 1$ points and that each point is incident with $q + 1$ lines. Using this information with the formulas derived above for general Steiner 2-designs, we find from Formula (2.2) that the number of lines is $q^2 + q + 1$ and from Formula (2.3) that each point is incident with $q + 1$ lines (and so by duality for projective planes, each line contains $q + 1$ points).

Recall that incidence structures such as t -designs and projective planes can be used to construct *incidence graphs*. Given an incidence structure $\mathcal{P}$ we will denote its incidence graph $G(\mathcal{P})$. Also recall that since edges only exist between points and blocks, $G(\mathcal{P})$ is bipartite.

The following lemma is part of folklore, and we include it for completeness.

LEMMA 3. *If $\mathcal{P}$ is a Steiner 2-design, then*

$$g(G(\mathcal{P})) \geq 6.$$

PROOF. Since bipartite graphs cannot contain odd cycles the girth of $G(\mathcal{P})$ must be even. Suppose there exists a cycle of length four in

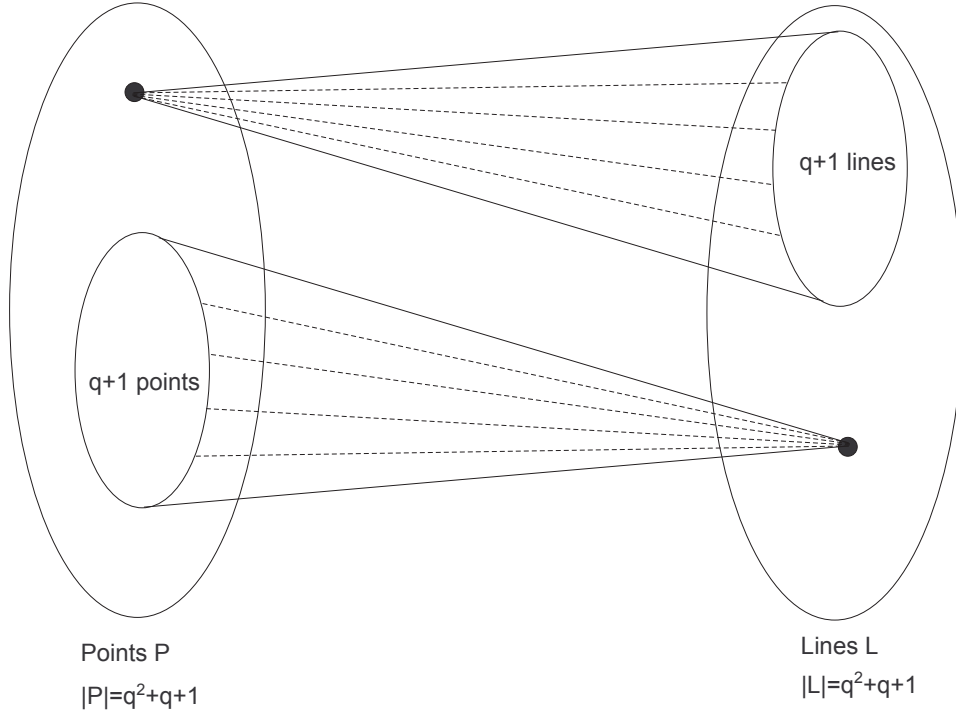


FIGURE 2.1. The incidence graph of a projective plane of order q .

$G(\mathcal{P})$. The vertices of a cycle of length four would be vertices corresponding to two points and two blocks in the design. This implies that two points are incident with two distinct blocks, which is a contradiction since two points determine a unique line. Therefore, $G(\mathcal{P})$ has girth at least 6. $\square$

Lemma 3, along with a theorem due to Aigner and Fromme [1] can be used to give our first explicit family of Meyniel extremal graphs.

THEOREM 4. [16] *If $\mathcal{P}$ is a projective plane of order q , then*

$$c(G(\mathcal{P})) = q + 1.$$

We include a proof of Theorem 4 from [16] for completeness.

PROOF. Since $\delta(G(\mathcal{P})) = q + 1$ and by Lemma 3 $g(G(\mathcal{P})) \geq 6$ the cop number is greater than or equal to $q + 1$ (see [1]). We now prove the upper bound.

We place $q + 1$ cops, say $C_1, C_2, \dots, C_{q+1}$, on a fixed set of points and consider cases for the location of the robber.

Case 1: Robber in L. Suppose the robber R begins in the vertices corresponding to lines in the plane. Call the vertex that R occupies $L(x_1, x_2, \dots, x_{q+1})$, where $x_1, x_2, \dots, x_{q+1}$ represent the points on the line. Since lines are incident with $q + 1$ points, $L(x_1, x_2, \dots, x_{q+1})$ has degree $q + 1$ in the graph $G(\mathcal{P})$. Thus, the robber has $q + 1$ escape routes from which he can leave the line.

Each pair of points $(x_1, C_1), (x_2, C_2), \dots, (x_{q+1}, C_{q+1})$ determines a unique line by properties of projective planes. Let these lines be $L(x_1, C_1), L(x_2, C_2), \dots, L(x_{q+1}, C_{q+1})$, respectively. The cops can then each move to these $q + 1$ lines.

Now on the next time-step the robber may remain at $L(x_1, x_2, \dots, x_{q+1})$ or move to any one of $x_1, x_2, \dots, x_{q+1}$. To capture the robber the cops need only to travel to $x_1, x_2, \dots, x_{q+1}$, respectively and the robber will be caught at this round or in the following round.

Case 2: Robber in P. Suppose the robber R begins in P . We identify the vertex occupied by the robber as R .

Since any two points determine a unique line the point occupied by the robber, R , and the points occupied by the cops $C_1, C_2, \dots, C_{q+1}$,

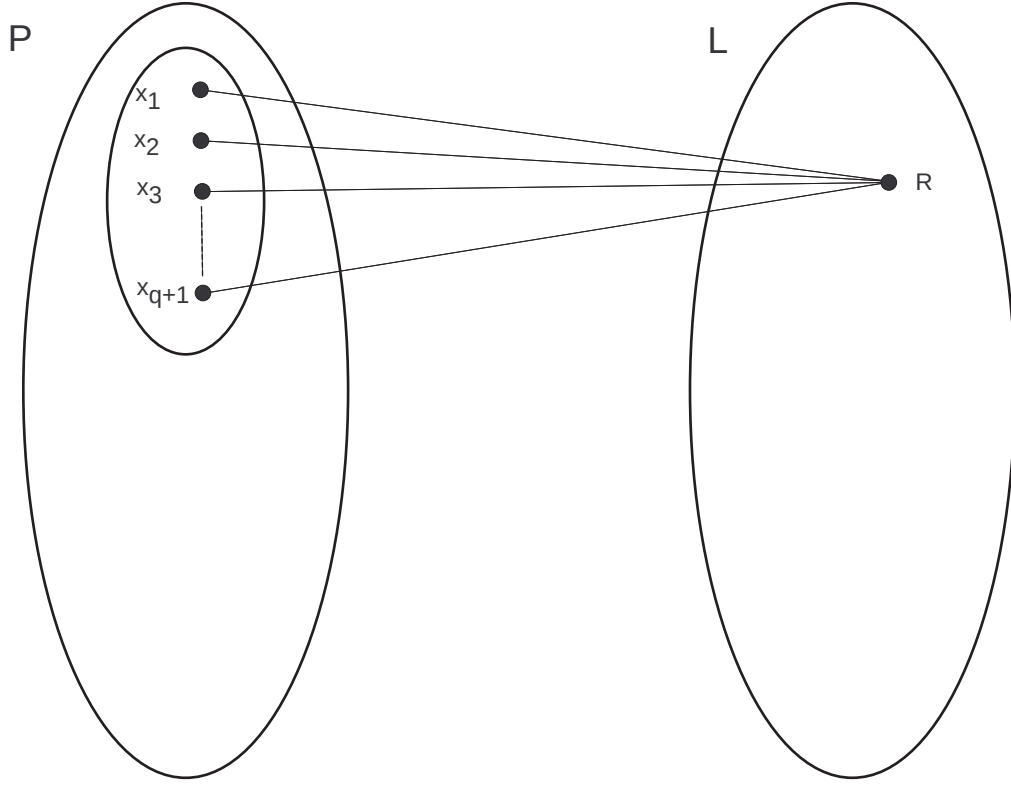


FIGURE 2.2. The robber has $q + 1$ escape routes.

uniquely determine a set of $q + 1$ distinct lines. These lines are denoted $L(R, C_1), L(R, C_2), \dots, L(R, C_{q+1})$.

Since the degree of the vertex occupied by the robber is $q + 1$ this set of $q + 1$ lines dominates the position of the robber. In order to cover the escape routes of the robber, the cops can move from their initial positions $C_1, C_2, \dots, C_{q+1}$ to the lines $L(R, C_1), L(R, C_2), \dots, L(R, C_{q+1})$, respectively. The cops then win in at most two rounds. $\square$

2.3. Affine and partial affine planes

Affine planes are another well-known example of an incidence structures. An *affine plane* of order q is a $2-(q^2, q, 1)$ design. By applying the

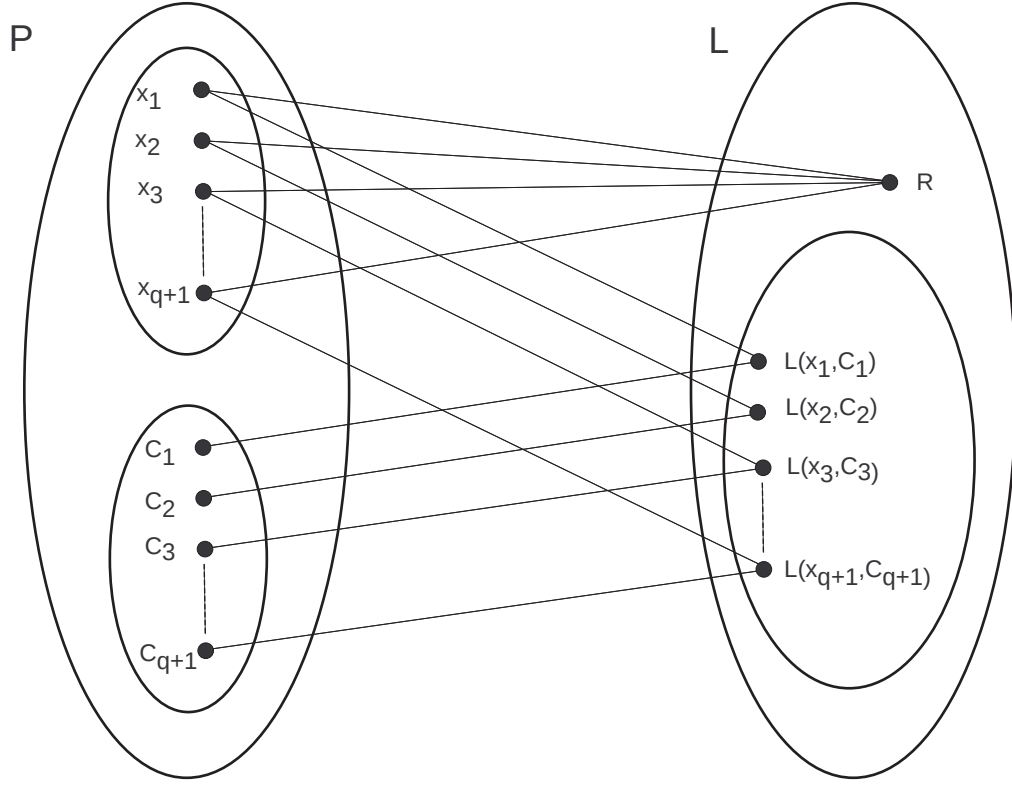


FIGURE 2.3. The $q + 1$ cops can dominate the position of the robber.

formulas (2.2) and (2.3) derived for Steiner 2-designs, it can be shown that in an affine plane, any point lies on $q + 1$ lines and there are $q^2 + q$ lines in total. A projective plane can be obtained from an affine plane by the addition of a point and a line at infinity. Similarly, an affine plane can be obtained from a projective plane by the deletion of any one line and all the points incident with it. In fact, an affine plane of order q exists if and only if a projective plane of order q exists; see [6]. An affine plane can be defined axiomatically in a similar manner to projective planes. An affine plane is a set of q^2 points and $q^2 + q$ lines with the following properties.

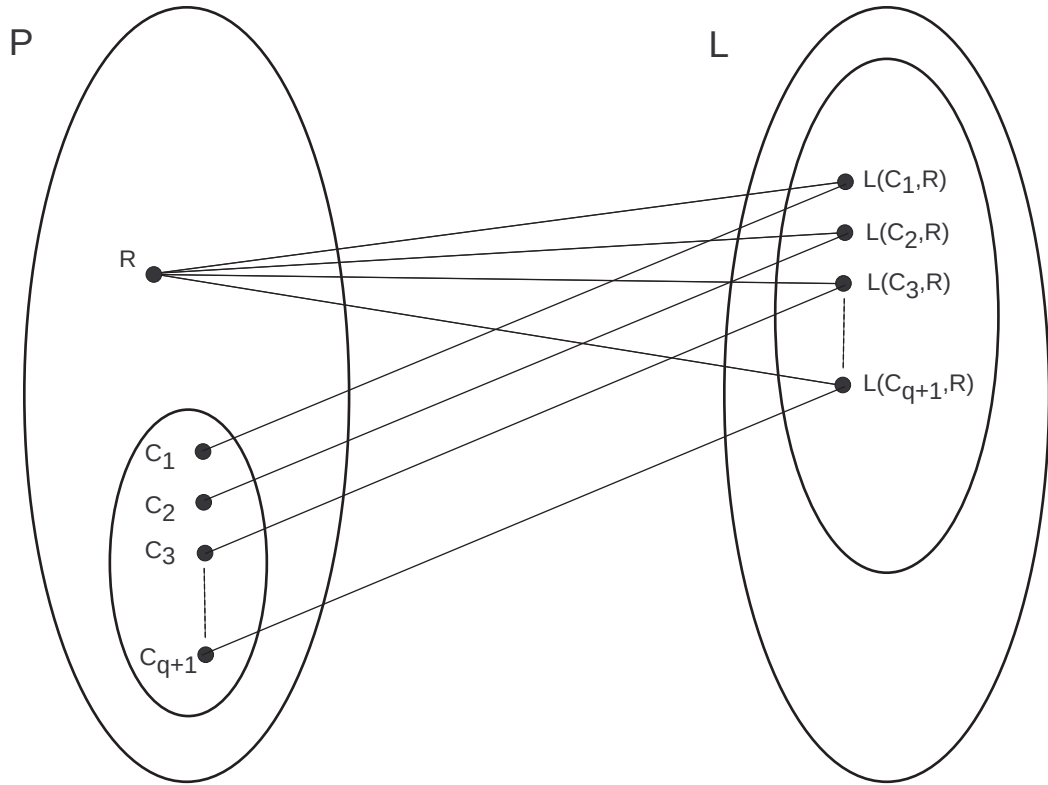


FIGURE 2.4. The robber again has $q + 1$ escape routes.

- (1) There is exactly one line incident with every pair of distinct points.
- (2) Given a point p and a line L , there is a unique line incident with p and containing no point of L .
- (3) There exist three non-collinear points.

Item (2) from the above list is sometimes referred to as *Playfair's Axiom* or *Euclid's Parallel Postulate*. Like the incidence graph for a projective plane, the incidence graphs G of an affine planes is bipartite and has $g(G) \geq 6$. There are $2q^2 + q$ vertices in the incidence graph of an affine

plane of order q . By properties of affine planes the vertices corresponding to points have degree $q + 1$ and vertices corresponding to lines have degree q . The incidence graph is then said to be $(q, q + 1)$ -regular.

Using affine planes, we derive a new family of Meyniel extremal graphs. For this, we need the notion of parallelism in affine planes.

In contrast to projective planes, where all lines intersect one another, lines in affine planes do not necessarily intersect. If two lines in an affine plane are equal or do not intersect, then they are said to be *parallel* to one another. Parallelism is an equivalence relation on the set of lines. *Parallel classes* (or *parallel pencils*) are the equivalence classes that partition the set of lines. Affine planes are said to be *resolvable* since each line can be identified with a unique parallel class. An affine plane of order q contains $(q + 1)$ -many parallel classes, each containing q lines. Note that every point in an affine plane is incident with exactly one line from each parallel class.

A new incidence structure can be constructed from an affine plane $\mathcal{A}$ by deleting the lines in some fixed set of $k > 0$ parallel classes. This structure, a *partial affine plane* $\mathcal{A}^{-k}$ of order q , contains $q^2 + q - kq$ lines and q^2 points. Each point in an affine plane is incident with $q + 1$ lines and each of those lines belongs to a distinct parallel class therefore, every point in $\mathcal{A}^{-k}$ is incident with $q + 1 - k$ lines. Thus, the minimum degree, written $\delta(G)$, of a vertex in the incidence graph is $\min(q, q + 1 - k)$.

The new main result of this section is the following.

THEOREM 5. *If $\mathcal{A}^{-k}$ is a partial affine plane of order q with $0 \leq k < q$ parallel classes deleted, then $G(\mathcal{A}^{-k})$ are $(q + 1 - k, q)$ -regular graphs with cop number between $q + 1 - k$ and q .*

If $k = o(q)$, then the graphs described in Theorem 5 have order $(1 - o(1))q^2$ and cop number $(1 - o(1))q$. In particular, we can set $k = q^{1-\varepsilon}$, for $\varepsilon \in (0, 1)$ and obtain infinitely many distinct Meyniel families.

PROOF OF THEOREM 5. Since

$$\delta(G(\mathcal{A}^{-k})) = \min(q, q + 1 - k)$$

and

$$g(G(\mathcal{A}^{-k})) \geq 6,$$

we know by the result of [1] that the cop number is greater than or equal to $\min(q, q + 1 - k)$.

This proves the lower bound, and so we now show that

$$(2.4) \quad c(G) \leq q.$$

To prove (2.4), we play with q cops. Fix a parallel class which was not deleted, say ℓ , and place one cop on each line of the parallel class. As each point is on some line in ℓ , the robber must move to some line $L \notin \ell$ to avoid being captured in the first round.

Fix a point P of L , and let L' be the line of ℓ which intersects L at P . Move the cop on L' to P . Now the robber cannot remain on L without being captured, and so must move to some point. However, each point not on L' is joined to some cop, so the robber must move

to a point of L' . But the unique point on L' joined to L is P , which is occupied by a cop. $\square$

We note that although the proof of Theorem 5 is elementary, it gives a new family of Meyniel extremal graphs.

CHAPTER 3

Growth Rates and Minimum Orders of k -Cop-Win Graphs

3.1. Introduction

Meyniel's Conjecture is concerned with the cop number of a graph of a given order. The conjecture posits that the asymptotic upper bound on the cop number for a graph of order n is $O(\sqrt{n})$. In this chapter we consider a different problem: How many k -cop-win graphs of order n exist? In this chapter we define two new parameters m_k and M_k that describe the minimum orders of k -cop-win graphs. We continue to define a new function $f_k(n)$ to describe the number of k -cop-win graphs of order n . In this chapter the values of m_k and M_k for $k = 1, 2$, and 3 are given. The values of M_3 and m_3 were not previously known. We also determine the values of $f_k(n)$ for $n \leq 10$. These values were computed by an exhaustive computer search that characterizes the cop number of all connected graphs of order 10 or less. In this chapter we also describe the algorithms used to complete this computer search. We conclude this chapter with a novel lower bound for the function $f_k(n)$.

3.2. Minimum orders and growth rates

For an integer $k \geq 1$, define m_k to be the minimum order of a connected graph satisfying $c(G) \geq k$. Define M_k to be the minimum order of a connected k -cop-win graph. Note that in general these values may not be the same; in fact $m_k \leq M_k$ since it may be the case that there exists a $(k+1)$ -cop-win graph of order n but there do not exist any k -cop-win graphs of order less than or equal to n . Although m_k is monotonically increasing, M_k may not be monotonically increasing. Only the first three values of m_k and M_k are known. It is trivial to see that $m_1 = M_1 = 1$ as a single vertex is cop-win. The cycle of order 4, C_4 , is the minimum order 2-cop-win graph; thus,

$$m_2 = M_2 = 4.$$

Through a computer search we have determined that

$$m_3 = M_3 = 10.$$

Please see Section 3.3 below for the details of the computer search and a complete discussion of our results.

The following theorem gives the asymptotic upper bound of the parameter m_k and relates Meyniel's conjecture to m_k . If Meyniel's conjecture holds, then it also gives the lower asymptotic bound for m_k and thus, the asymptotic behaviour of m_k .

THEOREM 6. *Let $k > 0$ be an integer.*

$$(1) \ m_k = O(k^2).$$

(2) *Meyniel's conjecture is equivalent to the property that*

$$m_k = \Omega(k^2).$$

Hence, if Meyniel's conjecture holds, Theorem 6 tells us that

$$m_k = \Theta(k^2).$$

PROOF. For item (1), note that the incidence graph of a projective plane has order $2(q^2 + q + 1)$ and has cop number $q + 1$, where q is a prime power [16]. This implies that

$$m_{q+1} = O(q^2).$$

Fix k a positive integer. Bertrand's postulate gives us a prime power q such that $k \leq q \leq 2k$ [7]. Hence,

$$m_k \leq m_q \leq m_{k+1} = O(q^2) = O((2k)^2) = O(k^2).$$

Item (1) follows.

For (2), if $m_k = o(k^2)$, then there exists a connected graph G with order $o(k^2)$ and cop number k . This is a contradiction since Meyniel's conjecture implies that $c(G) = o(k)$. Therefore, by Meyniel's conjecture we have that $m_k = \Omega(k^2)$.

For the reverse direction of item (2), suppose for contradiction that $m_k = \Omega(k^2)$ and that Meyniel's conjecture does not hold. We then have that there exists a connected graph G with order n such that

$$c(G) = k = \omega(\sqrt{n}).$$

This implies that $\sqrt{n} = o(k)$, which gives $n = o(k^2)$. Therefore,

$$m_k \leq n = o(k^2)$$

which is a contradiction. $\square$

Determining whether M_k are non-increasing is an open problem. The first few values of m_k and M_k suggests that these parameters are equal, for all $k \geq 1$, but this is also an open problem.

Define $f_k(n)$ to be the number of non-isomorphic connected k -cop-win graphs of order n . Define $g(n)$ to be the number of non-isomorphic (possibly disconnected) graphs of order n and $g_c(n)$ to be the number of connected non-isomorphic graphs of order n . By definition of these functions we have that $f_k(n) \leq g_c(n) \leq g(n)$ for all k and n . The table below presents the values of g , g_c , f_1 , f_2 and f_3 for small orders. The values of g and g_c are from The On-Line Encyclopedia of Integer Sequences [22]. The values for f_1 are the result of applying the algorithm to check whether a given graph is cop-win given in [15] to the data from [14]. To compute the values for f_2 , the algorithm to check whether $c(G)$ is less than or equal to two given in [4] was applied to the same data set. The value for f_2 is then the number of graphs of order n that have cop-number less than or equal to two, minus the number of cop-win graphs of order n . The value for f_3 was computed in a similar manner. The algorithms used to complete these computations are discussed in the next section.

order n	$g(n)$	$g_c(n)$	$f_1(n)$	$f_2(n)$	$f_3(n)$
1	1	1	1	0	0
2	2	1	1	0	0
3	4	2	2	0	0
4	11	6	5	1	0
5	34	21	16	5	0
6	156	112	68	44	0
7	1044	853	403	450	0
8	12,346	11,117	3,791	7,326	0
9	274,668	261,080	65,561	195,519	0
10	12,005,168	11,716,571	2,258,313	9,458,257	1

As a result of the classification of all connected graphs of order 10 or less we determined that $m_3 = M_3 = 10$. Further, the graph that realizes these minimum orders, the Petersen graph, is the unique minimum order 3-cop-win graph. The Petersen graph is a very famous graph that arises often as a counterexample in many areas of graph theory. Similar to the unique minimum order 2-cop-win graph C_4 which is 2-regular and has girth 4, it is 3-regular and has girth 5.

3.3. Algorithms for computing cop number

Cop-win graphs have been well understood since the introduction of the game Cops and Robbers. Both Nowakowski and Winkler [15], and Quilliot [18] give a characterization of cop-win graphs based on the idea of a cop-win ordering or elimination ordering. These characterizations can be used to define cop-win graphs in an algorithmic manner. A

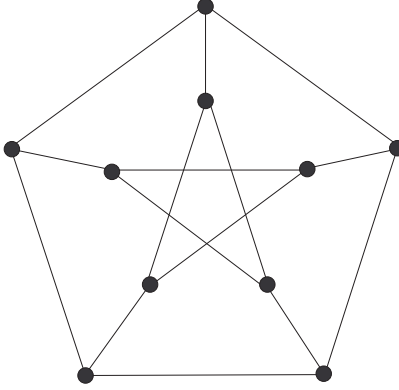


FIGURE 3.1. The Petersen Graph.

cop-win graph can be recognized by the repeated deletion of a vertex with certain adjacency properties called a corner. A *corner* is a vertex whose closed neighbour set is contained in the closed neighbour set of some other vertex in the graph. More succinctly, a *corner* is a vertex v such that $N[v] \subseteq N[u]$, for some $u \in V(G)$. Corners play an important role in the mechanics of the game of Cops and Robbers. If a robber is on a corner, then the cop can dominate the neighbour set of the robber. If the cop is on a vertex that dominates a corner, then no matter where the robber moves the cop can capture the robber on the next time-step. A critical, though elementary, fact we prove now is the following.

LEMMA 7. [5] *A cop-win graph contains a corner.*

PROOF. Consider the last move of the robber before being captured by the cop. If the cop can capture the robber on the next move, then the cop is in a position that is both adjacent to the robber and adjacent to all of the neighbours of the robber. Hence, the robber is on a corner dominated by the vertex occupied by the cop. $\square$

Moreover, this result implies that when a corner is deleted from a cop-win graph the resulting graph contains a corner. In a cop-win graph this process can be repeated until only a single vertex remains. The order in which corners are deleted is the *cop-win* or *elimination ordering*. If a graph can be reduced to a single vertex (that is, to K_1) by the successive deletion of corners, then it is called *dismantlable*. Cop-win graphs are exactly those that are dismantlable.

THEOREM 8. [5] *A graph G is cop-win if and only if it is dismantlable.*

PROOF. By induction on the order of G , we show that if G is a cop-win graph, then G is dismantlable. In the base case G has order 1 and so must be isomorphic to K_1 , which is dismantlable. Suppose G has order $n \geq 1$, where n is some fixed integer. Since G is cop-win, it must contain a corner, say u , by Lemma 7.

Let v dominate u . Let $H = G - u$. We claim H is cop-win. If H is cop-win, then H itself will contain a corner, and this will prove the forward direction by induction. We consider two parallel Cops and Robbers games: one played in G and one in H . The game in H may be considered as being played in G ; since H is an induced subgraph of G . The strategy in G may not be sufficient alone to capture the robber in H . For example, the robber may need to leave H to be captured in G .

Let the cop in H be labeled C' . Let C' move as C does in G ; however, when C moves to u , then C' moves to v . We claim this is a winning strategy for C' in H . Let the cop play in G with R restricted

to H . Suppose that the cop is about to win in G . But then $N[R]$ is contained in $N[C]$ in G . As v dominates u , we must have that $N[R]$ is contained in $N[C']$. Hence, the robber loses in the next round played in H .

Now suppose G is dismantlable. By induction on the order of the graph we show that G is cop-win. As before in the base case G is isomorphic to K_1 and hence, is cop-win. Suppose G has order $n+1$ and is dismantlable for some fixed integer $n \geq 1$. Since G is dismantlable, it contains a corner. Label the corner u and the vertex dominating it v . The graph resulting from the deletion of u , $G - u$ is also dismantlable. Since $G - u$ has order n , it is cop-win by induction hypothesis. We will show that G is cop-win. A single cop has a winning strategy on $G - u$, since it is a cop-win graph. A cop can play this strategy on G with the following modification: if the robber is on the vertex u , the cop plays as if the robber is on v the vertex that dominates u . We say the cop moves to capture the *shadow* of the robber in $G - u$. If the cop is able to capture the shadow of the robber on the vertex v , then the cop can capture the robber on the vertex u in the graph G because u is a corner dominated by v . Since the cop has a winning strategy on $G - u$ the cop has a winning strategy for G . Therefore, if a graph G is dismantlable, then it is cop-win. $\square$

This characterization of a graph in terms of the deletion of corners gives an algorithm for recognizing cop-win graphs. This algorithm is given below in pseudocode. In natural language it can be summarized as follows: While the graph G contains a corner, delete the corner to

obtain G' . Set $G=G'$ and repeat the process. When the graph G no longer contains a corner, and if $G \cong K_1$, then the initial graph G is cop-win; otherwise, G is not cop-win.

Algorithm 1 CHECK COP-WIN

Require: $G = (V, E)$

```

1: while  $|V(G)| > 1$  do
2:   if  $G$  contains a corner,  $x$  then
3:      $V(G) \leftarrow V(G) \setminus \{x\}$ 
4:      $E(G) \leftarrow E(G) \setminus \{(a, b) \in E(G) : a = x \text{ or } b = x\}$ 
5:   else
6:     return FALSE
7:   end if
8: end while
9: return TRUE

```

The process of dismantling the graph by successive deletion of corners gives a linear ordering of the vertices, called a *cop-win (or elimination) ordering*. The cop-win ordering can be used to construct a winning strategy for the cop. The *No-backtrack* strategy, first described by Clarke and Nowakowski [9], defines a strategy for the cop that results in the capture of the robber in at most n moves.

Given a cop-win ordering of G , $[n] = [1, 2, \dots, n]$, define

$$G_i = G \upharpoonright \{n, n-1, \dots, i\}$$

for $i \in [n]$. The graph G_i is the induced subgraph of G restricted to vertices not appearing in the cop-win ordering before i . Note that $G_1 = G$ and G_n is just the vertex n . If u is a corner dominated (or covered) by v in G , then a *retraction* is a mapping $f : G \rightarrow G - u$

defined by

$$f(x) = \begin{cases} v & \text{if } x = u, \\ x & \text{otherwise.} \end{cases}$$

Each deletion of a corner in the graph G corresponds to a retraction. The No-backtrack strategy is defined by the cop-win ordering which can be encoded in the composition of retractions. Let $f_i : G_i \rightarrow G_{i+1}$ be the retraction mapping i onto the vertex that dominates i in G_i , for each $1 \leq i \leq n-1$. Define F_1 to be the identity mapping on G , defined by $F_1(x) = x$ for all $x \in V(G)$. For $2 \leq i \leq n$ define

$$F_i = f_{i-1} \circ \dots \circ f_2 \circ f_1.$$

Since the maps f_i correspond to the deletion of corners, the maps F_i correspond to the successive deletion of all corners the cop-win ordering up to but not including i . Note that for all i , since the retractions f_i either map a vertex to itself or to an adjacent vertex, $F_{i+1}(x)$ and $F_i(x)$ are equal or adjacent because they are composed of the retractions f_i .

The No-backtrack strategy is as follows. The cop begins on G_n , which is just the vertex n . In every round the cop plays a winning strategy on the graph restricted to the vertices not appearing in the cop-win ordering before i , namely G_i . The cop determines the position of the “shadow” of the robber on G_i and captures the “shadow” of the robber in G_i . In the first round the cop plays on G_n , in which there is only one possible position for the robber, n . Note that every vertex in G maps to vertex n under F_n . In the next round the cop computes $F_{n-1}(u)$ where u is the vertex of G currently occupied by the

robber. The vertex determined by the map $F_{n-1}(u)$ is the *shadow* of the robber on G_{n-1} . We think of the shadow as the position of the robber in the restricted graph. The cop then moves to capture the robber on G_{n-1} . In each remaining round the cop repeats this process. First, the cop determines the position of the shadow robber by computing $F_i(u)$, where i is the current round numbered from n to 1 and u is the vertex currently occupied by the robber and then moves to capture the shadow robber. After n rounds the cop is playing on $G_1 = G$ and can capture the robber.

We now consider an example. Consider the application of the cop-win algorithm to the graph G in Figure 3.2 below. We note that the order in which we delete corners is not unique. The graph G contains four corners: a, c, d , and e . We could begin the algorithm by deleting any one of these corners.

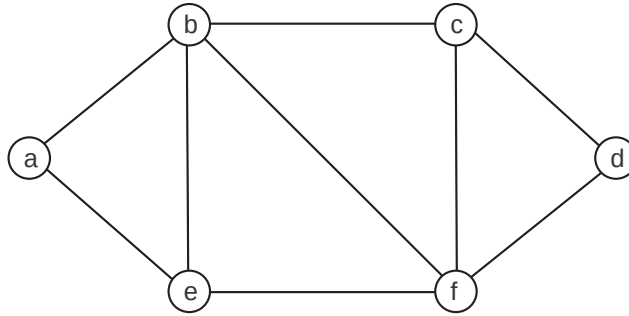


FIGURE 3.2. The graph G , a cop-win graph.

Suppose we begin by deleting a , and note that it is dominated by the vertices b and e . Again we could choose to record either of these as the dominator for a , we will choose b . The function f_1 then becomes:

$$f_1(x) = \begin{cases} b & \text{if } x = a, \\ x & \text{otherwise.} \end{cases}$$

It is useful to note that every retraction f_i maps a corner to a vertex that dominates it and keeps all other vertices fixed. In order to represent the retraction all that is necessary is the corner and dominating vertex. Thus, the first retract could be written as $f_1 : a \rightarrow b$. It can be easily checked that (a, e, b, f, c, d) is a valid cop-win ordering for the graph in Figure 3.2. The cop-win ordering for the graph G is given in Figure 3.3.

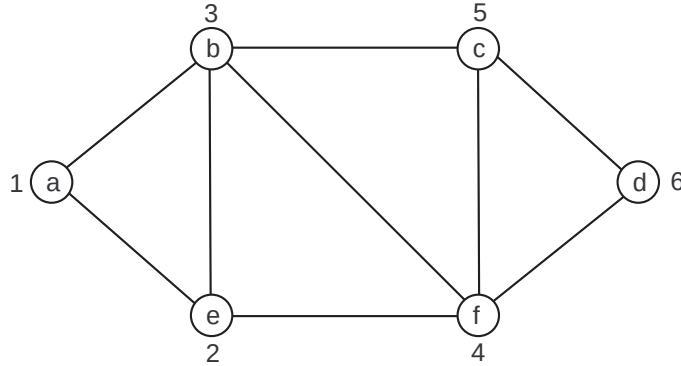


FIGURE 3.3. A cop-win ordering for the graph G .

The following is a list of possible retractions that correspond to the given cop-win ordering:

$$f_1 : a \rightarrow b.$$

$$f_2 : e \rightarrow b.$$

$$f_3 : b \rightarrow f.$$

$$f_4 : f \rightarrow c.$$

$$f_5 : c \rightarrow d.$$

The maps F_i are constructed from last used to first used. The first map computed is F_1 which is the same as f_1 . This function as above maps $a \rightarrow b$ and leaves all other vertices fixed. In each step of constructing the maps F_i we only need to compose the previous F map with the next f map. This can be seen by noting that we can write F_i , the composition of the first $i - 1$ f -maps as $F_i = f_i \circ F_{i-1}$. The next map can then be computed as $F_2 = f_2 \circ F_1$. The retraction f_2 maps $e \rightarrow b$. The function F_2 then maps both vertices a and e to vertex b and leaves all others fixed. Here are the explicit mappings F_i for $1 \leq i \leq 5$.

$$F_1(x) = \begin{cases} b & \text{if } x = a, \\ x & \text{otherwise.} \end{cases}$$

$$F_2(x) = \begin{cases} b & \text{if } x = a \text{ or } e \\ x & \text{otherwise.} \end{cases}$$

$$F_3(x) = \begin{cases} f & \text{if } x = a, b \text{ or } e \\ x & \text{otherwise.} \end{cases}$$

$$F_4(x) = \begin{cases} c & \text{if } x = a, b, e \text{ or } f \\ x & \text{otherwise.} \end{cases}$$

$$F_5(x) = \begin{cases} d & \text{for all } x \in V(G) \end{cases}$$

Now, the maps, F_i can be used to determine the cops strategy. The cop's first move is determined by F_5 , so the cop will always begin on the vertex d . It makes sense that the cops initial decision should not depend on the robber's actions since the cop chooses the start vertex before the robber has been placed in the graph.

Suppose the robber chooses the vertex a , the vertex with the greatest distance from the initial position of the cop. The next move of the cop is then computed by finding $F_4(a)$, which indicates the cop should move to the vertex c . Suppose the robber moves to the vertex e ; since $F_3(e) = f$, the cop moves to f . Now, the robber's only option is to move to the vertex a . The map $F_2(a) = b$, tells the cop to move to the vertex b which dominates the position of the robber thus, the cop wins in the next round regardless of the choice of the robber in the next round.

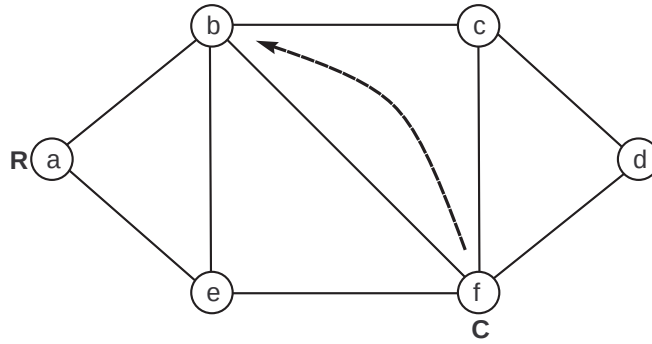


FIGURE 3.4. The cop computes her second-to-last move, $F_2(a) = b$, and so moves to b .

3.4. The k -cop-win Algorithm

The problem of deciding whether a given graph is cop-win is relatively easy computationally speaking. Although it is a more difficult problem, there is an algorithm to check whether a graph has cop number k or less for a fixed k . This algorithm, originally due to Berarducci and Intriglia [2], relies on the idea of a graph product. The *graph product* of two graphs, G and H , is a new graph. The vertices of the product are the Cartesian product of the vertices of the original graphs. That is, each vertex of the product graph is an ordered pair of vertices from the constituent graphs. There are several different types of graph product. The difference between these types of graph product is in how the edges of the product graph are determined. The edges of all graph products are determined by the edges of the original graphs. The most common graph product is the *Cartesian product*, denoted $G \square H$. Suppose (g_1, h_1) and (g_2, h_2) are two vertices in $V(G \square H)$. An edge exists between (g_1, h_1) and (g_2, h_2) if they are adjacent in one component and equal in the other. Note that we write $g \sim h$ if the vertices g and h are adjacent. More precisely, if $g_1 \sim g_2$ and $h_1 = h_2$ or $g_1 = g_2$ and $h_1 \sim h_2$. Another common graph product is *categorical product*, denoted $G \times H$. Edges are adjacent in the categorical product if the components are both adjacent in the original graphs, $g_1 \sim g_2$ and $h_1 \sim h_2$. The graph product used by the algorithm to decide whether a graph has cop number k or less is the *strong product*, denoted $G \boxtimes H$. This notation hints at the relationship it shares with the Cartesian product and the categorical product. The edge set of the strong product can be thought of

as the union of the edge sets of the Cartesian and categorical product. There is an edge between (g_1, h_1) and (g_2, h_2) if and only if: $g_1 \sim g_2$ and $h_1 = h_2$; or $g_1 = g_2$ and $h_1 \sim h_2$; or $g_1 \sim g_2$ and $h_1 \sim h_2$. Vertices in the product are adjacent if they are adjacent or equal in each component. Figure 3.4 below shows the strong product of two graphs in terms of the Cartesian and categorical products.

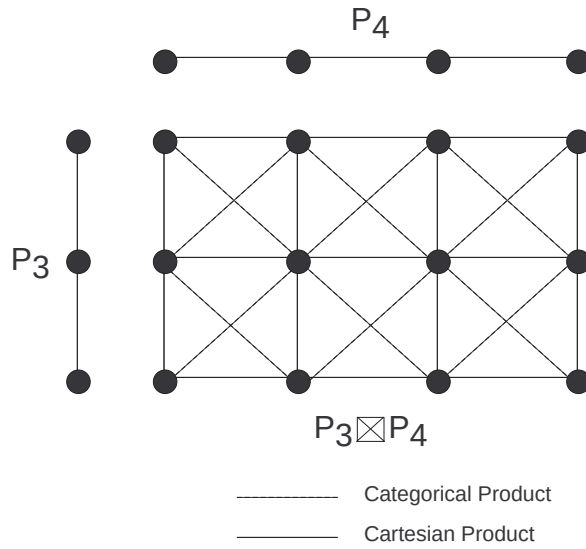


FIGURE 3.5. The graph G , a cop-win graph.

The product of graphs can be generalized to the notion of the exponentiation of a graph. Like the exponentiation of numbers, exponentiation of graphs is just repeated application of products. The k^{th} strong power of a graph is constructed in a similar manner to the product of graphs. A vertex in the k^{th} strong power of a graph G is a k -tuple of vertices of G . A pair of k -tuples are adjacent if they are adjacent or equal in each component. The algorithm to check whether a graph is k -cop-win, for a fixed k uses the k^{th} strong power, written $\boxtimes^k G$, to

simulate the movements of the cops. Each coordinate of the k -tuple represents the position of the k cops in the original graph. Using this approach the algorithm rules out possible positions for the robber in the original graph. The following theorem is the basis of this algorithm. Note that $2^{V(G)}$ is the set of all subsets of $V(G)$.

THEOREM 9. [2] *Suppose $k \geq 1$ is an integer. Then $c(G) > k$ if and only if there is a mapping $f : V(\boxtimes^k G) \rightarrow 2^{V(G)}$ with the following properties.*

(1) *For every $u \in V(\boxtimes^k G)$,*

$$\emptyset \neq f(u) \subseteq V(G) \setminus N_G[u].$$

(2) *For every $uv \in E(\boxtimes^k G)$,*

$$f(u) \subseteq N_G[f(v)].$$

We omit the proof, which can be found in [4].

The value of $f(u)$ is the set of vertices in the graph G such that, if the cops occupy the k vertices in G specified by the vertex u , then the robber has a strategy to escape capture in the next round. If any one of the maps $f(u)$ is empty for some u in $V(\boxtimes^k G)$, then the robber can be captured by the cops from this position. That is, k or fewer cops have a winning strategy on the graph G . Algorithm 2 gives the statement of the algorithm in pseudocode.

The first phase of the algorithm is to initialize the sets $f(u)$ to be $V(G) \setminus N_G[u]$, where $N_G[u]$ contains the k vertices of G that make up u , as well as their neighbours in G . This step rules out the vertices that

Algorithm 2 CHECK K-COP-WIN

Require: $G = (V, E), k \geq 0$

```
1: initialize  $f(u)$  to  $V(G) \setminus N_G[u]$ , for all  $u \in V(\boxtimes^k G)$ 
2: repeat
3:   for all  $(u, v) \in E(\boxtimes^k G)$  do
4:      $f(u) \leftarrow f(u) \cap N_G[f(v)]$ 
5:      $f(v) \leftarrow f(v) \cap N_G[f(u)]$ 
6:   end for
7: until the value of  $f$  is unchanged or  $f(u) = \emptyset$  for some  $u \in V(\boxtimes^k G)$ 
8: if there exists  $u \in V(\boxtimes^k G)$  such that  $f(u) = \emptyset$  then
9:   return  $c(G) \leq k$ 
10: else
11:   return  $c(G) > k$ 
12: end if
```

are dominated by the cops when they are in the position specified by u . If after this initialization phase, we have that $f(u) = \emptyset$, then k or fewer cops can form a dominating set in G . The main phase of the algorithm uses the second rule to rule out possible escape routes for the robber. The second rule of Theorem 9 guarantees that the robber will always have a way to get from one safe set of vertices to another, from $f(u)$ to $f(v)$ or from $f(v)$ to $f(u)$, along the edge uv . Also note that if any $f(u)$ is empty at any point in the algorithm, then by repeated application of the second rule of Theorem 9, all maps will eventually be empty. If $f(u)$ is empty for any u in $V(\boxtimes^k G)$, then there exists a position in the graph from which the robber cannot escape and therefore there is a strategy for k or fewer cops to win in G .

Note that Algorithm 2 has polynomial running time. The worst case running time of this algorithm $O(n^{3k+3})$ (see[4]). The main contribution to the running time of the algorithm comes from lines 2–7.

Since the cardinality of $f(u)$ is reduced by at least one in every iteration of lines 2–7. This section is then executed at most $O(n^{k+1})$ times. In each of these iterations all edges of the product graph $\boxtimes^k G$ are being processed in lines 3–6. Since there are at most $\frac{n(n-1)}{2}$ edges in the original graph, there are at most $(\frac{n(n-1)}{2})^k$ edges in $\boxtimes^k G$. Thus, iterating through the edges of $\boxtimes^k G$ has computational complexity $O(n^{2k})$. Finally, the operation of taking intersections and finding the neighbourhoods of each vertex (lines 4–5) have complexity $O(n)$ and $O(n^2)$, respectively. The total running time is then the product of these complexities as follows:

$$\begin{aligned}
O(n^{k+1})O(n^{2k})(O(n) + O(n^2)) &= O(n^{k+1})O(n^{2k})O(n^2) \\
&= O(n^{k+1})O(n^{2k+2}) \\
&= O(n^{3k+3}).
\end{aligned}$$

3.5. Lower bounds

Recall from the beginning of this chapter that $f_k(n)$ is the number of non-isomorphic connected k -cop-win graphs of order n and that $g(n)$ is the number of non-isomorphic (possibly disconnected) graphs of order n . As previously mentioned, we have a trivial upper bound on $f_k(n)$ in terms of $g(n)$: $f_k(n) \leq g(n)$ for all k and n . In this section we give an original lower bound on $f_k(n)$ in terms of $g(n)$. We begin with a lemma that shows that given a k -cop-win graph, we can construct another k -cop-win graph of arbitrarily large order.

THEOREM 10. *Given graph G of order n and a k -cop-win graph H of order m_k , there exists a k -cop-win graph G' constructed from G and H with order $n + m_k + 1$.*

PROOF. Construct G' as follows. Attach a new vertex, say x , to all vertices of G . The vertex x is universal in the new graph $G \cup \{x\}$. Attach x to a fixed vertex in H , say y , to form G' from H and $G \cup \{x\}$.

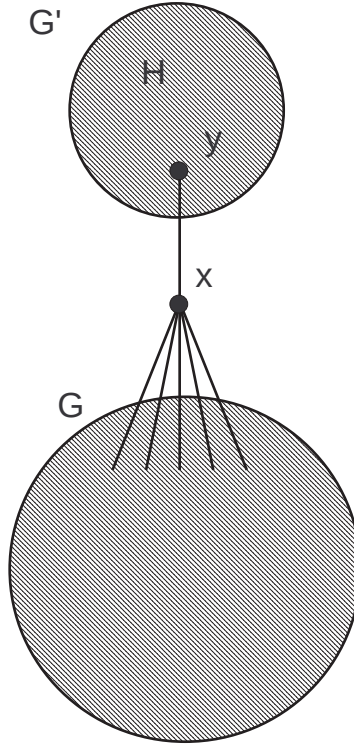


FIGURE 3.6. The construction of G' .

If there are less than k cops, then the robber has a winning strategy. The robber can remain in H since k cops are required to win in H . Thus, $c(G') \geq k$.

Now to show $c(G') \leq k$ we give a winning strategy for k cops in G' . Place one cop on x and the remaining $k - 1$ cops on some vertices of G .

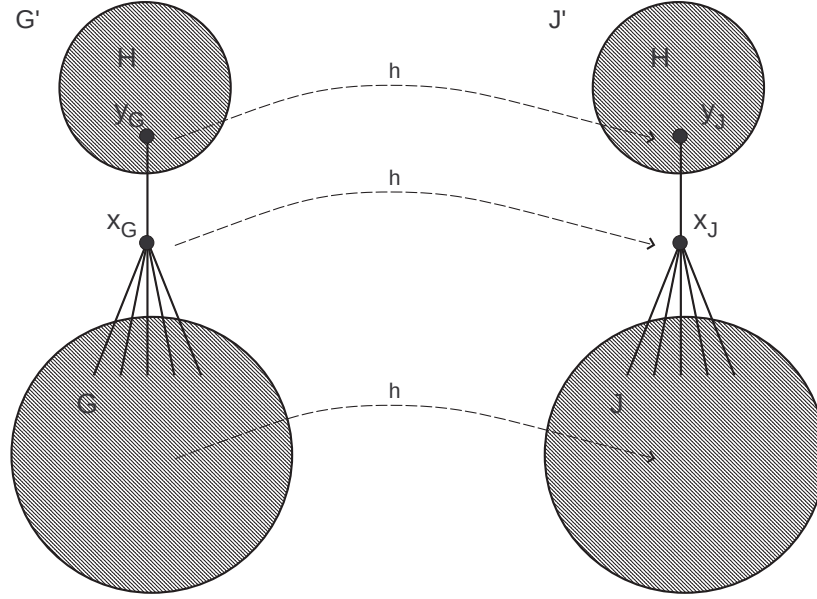


FIGURE 3.7. The isomorphism $h : G' \rightarrow J'$.

The robber cannot start in G since the vertices of G are dominated by x . The robber then begins in H . The cops can then all move to play their winning strategy for k cops on H . If the robber leaves H , then the cops play as if the robber is on y . Eventually either the robber is caught in H or the robber has moved to $G \cup \{x\}$. If the robber has moved to $V(G) \cup \{x\}$, then at least one cop occupies y . One of these cops moves to x which dominates G and the robber is caught in the next round. $\square$

THEOREM 11. *Suppose that $n > m_k$ and let G and J be connected graphs of order n and H be a minimum order connected k -cop-win graph of order less than n . Let G' and J' be the graphs constructed as above. If $G \not\cong J$, then $G' \not\cong J'$.*

PROOF. Suppose for contradiction that $G' \cong J'$, but $G \not\cong J$ and let h be an isomorphism from G' to J' . We claim that the vertex with maximum degree in G' is x_G . The degree of x_G is $n + 1$ since it is connected to all vertices of G and one vertex of H_G . It is the unique vertex with maximum degree since no vertex in H_G can have more than $n - 1$ neighbours in H_G and a vertex in G has at most $n - 1$ neighbours in G . A similar argument shows that the unique vertex with maximum degree in J' is x_J . Since x_G and x_J are the unique vertices with maximum degree in G' and J' , respectively, h must map x_G to x_J . The vertex y_G is the unique vertex of G' that is adjacent to x_G and adjacent to vertices that are not adjacent to x_G . Similarly, y_J is the unique vertex in J' adjacent to x_J and no other neighbours of x_J . Thus, y_G must be mapped to y_J by f . This implies that h maps H_G to H_J and that the restriction of h to G is an isomorphism mapping G to J . This contradicts our assumption that G is not isomorphic to J . Therefore, G' is not isomorphic to J' . $\square$

The following theorem gives us the central result of this section. The theorem gives a lower bound for the number of k -cop-win graphs of a given order in terms of the function $g(n)$.

THEOREM 12. (1) For all $n > 1$,

$$g(n - 1) \leq f_1(n).$$

(2) For $k > 1$ and all $n > 2m_k$,

$$g(n - m_k - 1) \leq f_k(n).$$

PROOF. For item (1), fix a graph G of order $n - 1$. Form G' by adding a universal vertex to G . The graph G' is cop-win since placing a single cop on the universal vertex is a winning strategy for a cop. If $G \not\cong H$, then we show that $G' \not\cong H'$. Suppose for contradiction that there exists an isomorphism $h : G' \rightarrow H'$. Let G_U and H_U be the sets of universal vertices in G' and H' , respectively. The isomorphism h must map G_U to H_U . Fix some vertex in G_U , call it x . If h is restricted to the vertices of G excluding x , then we have that

$$h \upharpoonright G' - x : G' - x \rightarrow H' - h(x)$$

is an isomorphism. This is a contradiction since $G' - x \cong G$ and $H' - h(x) \cong H$, and thus, $G \cong H$. Since for every isomorphism type of order $n - 1$ we can construct a cop-win graph of order n it follows that for all $n > 1$, $g(n - 1) \leq f_1(n)$.

For (2) note that there are $g(n - m_k - 1)$ -many non-isomorphic graphs of order $n - m_k - 1$. By Theorem 10 for each of these $g(n - m_k - 1)$ -many graphs, a k -cop-win graph of order n can be constructed. By Theorem 11 each of these k -cop-win graphs are non-isomorphic so long as

$$n - m_k + 1 \geq m_k.$$

But the latter inequality is equivalent to $n > 2m_k$, which is our hypothesis. Therefore, there are at least $g(n - m_k - 1)$ distinct k -cop-win graphs of order n . $\square$

CHAPTER 4

The Game of Cops and Robbers on Hypergraphs

4.1. Introduction

While the game Cops and Robbers is usually played on a graph, the game can be played on many other mathematical structures. One such structure is a hypergraph. A *hypergraph* is set of vertices and subsets of vertices, called *hyperedges*. Hypergraphs are a generalization of graphs, where hyperedges have cardinality 2. Instead of connecting two vertices like edges, hyperedges can connect some number of vertices. In this chapter we give some definitions and notation to describe hypergraphs and discuss the game of Cops and Robbers played on hypergraph. We conclude the chapter with a discussion of some open problems.

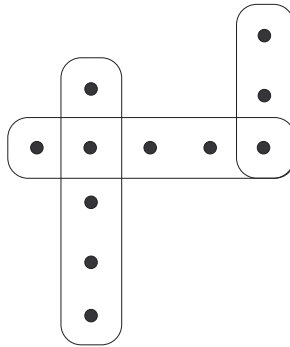


FIGURE 4.1. A hypergraph with hyperedges of cardinality 3 and 5.

4.2. Definitions and Notation

If a vertex is contained in just one hyperedge, then we call this vertex *internal*. We call a vertex that is contained in two or more hyperedges *external*.



FIGURE 4.2. The vertex a is internal, the vertex b is external.

In contrast to the notion of regularity in graphs, in hypergraphs, regularity describes the number of vertices in each edge rather than the number of edges incident with a vertex. A hypergraph is k -regular if every hyperedge contains exactly k vertices. or the remainder of the chapter, all the hypergraphs we consider will be k -regular for some $k > 1$. We say that a hypergraph is t -joined if each intersection of hyperedges contains exactly t vertices.



FIGURE 4.3. A 5-regular 2-joined hypergraph.

Similar to the incidence graphs of designs, we can define an incidence graph of a hypergraph. The *graph* of a hypergraph H , written $G(H)$, is a graph with the vertices of H and an edge between these vertices if they are connected by a hyperedge in H . That is, $V(G(H)) = V(H)$ and $(u, v) \in E(G(H))$ if and only if $\{u, v\} \subseteq e$ for some $e \in E(H)$. Note that a 1-joined, 2-regular hypergraph H is isomorphic to its own underlying graph.

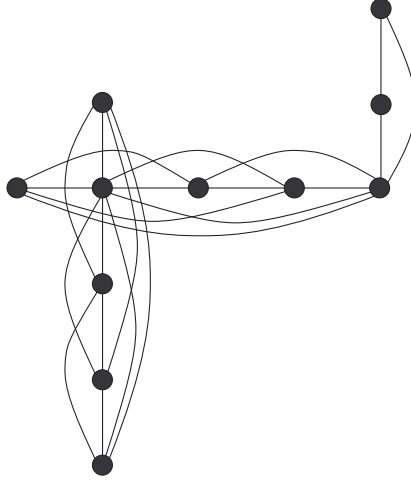


FIGURE 4.4. The graph $G(H)$ for H , the hypergraph in Figure 4.1.

In the subject of graph theory there are many types of special graphs. Paths and cycles are among the most common of these. We define a hypergraph analogue of paths and cycles. A *hyperpath* is a sequence of hyperedges $E_1, E_2, \dots, E_k$, such that E_i and E_{i+1} are t -joined, for some $t > 0$, and for $i = 1, 2, \dots, k-1$ and $E_i \cap E_j = \emptyset$ when $j \neq i+1 \pmod k$.

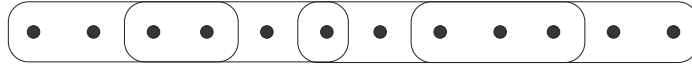


FIGURE 4.5. A hyperpath.

For $k > 2$ an integer, a k -*hypercycle* is a collection of k hyperedges, E_i , with two hyperedges E_i and E_j incident if $i = j + 1 \pmod k$. For more information about hypergraphs we direct the reader to the books by Berge [3] and Voloshin [23].

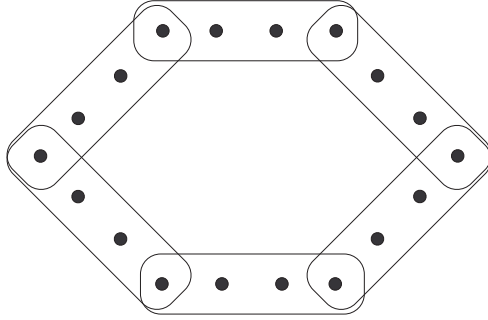


FIGURE 4.6. A hypercycle.

4.3. Cops and Robbers on hypergraphs

The game of Cops and Robbers played on a hypergraph is analogous to the game played on graphs. Some set of cops choose vertices to occupy, and then the robber choose a vertex. The players can move to vertices by moving from their present vertex u to any vertex v such that u and v are in hyperedge. The remaining rules of the game, along with the definition of the cop number of hypergraphs is analogous to the case for graphs.

As before, a hypergraph where one cop has a winning strategy is called *cop-win*. Corners are defined analogously as in graphs. Hyperpaths are cop-win. To see that a hyperpath is cop-win we will consider a strategy for a cop that relies upon the following lemmas.

LEMMA 13. *A cop playing on a connected hypergraph with at least two hyperedges can always move from an external vertex to another distinct external vertex.*

PROOF. As the cops can move from any vertex of a hyperedge to any vertex of an incident hyperedge, the lemma follows. $\square$

LEMMA 14. *The cop can play a winning strategy by remaining on external vertices until, perhaps, the final move of the game.*

PROOF. By Lemma 13, the cop can always travel from external vertex to external vertex. To capture a robber on an internal vertex, the cop needs only to occupy the same hyperedge as the robber. If the robber is on an external vertex, then the cop can follow the robber to the next edge by traveling along external vertices. $\square$

THEOREM 15. *If H_p is a hyperpath, then $c(H_p) = 1$.*

PROOF. The cop begins on an external vertex of the edge E_1 at one end of the path. By Lemma 14 the cop may move from external vertex to external vertex through the path. The cop moves via external vertices from one end to the other. If the cop encounters the robber on an internal vertex, then the cop can capture the robber on the next move. $\square$

THEOREM 16. *If H_c is a hypercycle, then $c(H_c) = 2$.*

PROOF. If the game is played on hypergraph with a single cop, then the robber can always evade the cop. The robber evades capture by moving on external vertices around the cycle. Thus, hypercycles are not cop-win.

However, if the game is played with two cops, the following strategy for the cops is sufficient to win the game. Place the first cop C_1 on an external vertex of edge E_1 , place the second cop C_2 on the edge E_k . Since there are no edges in between the edges E_1 and E_k , the

robber cannot choose to begin between the cops. The cops can then each move away from each other on each successive round, with C_1 , moving to edges with higher index, and C_2 moving to edges with lower index. Eventually, the cops will occupy the same edge and the robber will be caught on the next round. $\square$

Consider a Steiner 2-design, or a $2-(v, k, 1)$ design. Recall from Chapter 2 that $2-(v, k, 1)$ designs are a set of v points, and a set of blocks of size k , such that any two points determine a unique block. We assume always that $v > k$. Steiner 2-designs, and indeed any other design, can be thought of as a hypergraph with the vertices being points and hyperedges defined by the blocks. In the following theorem we show that Steiner 2-designs are in fact cop-win.

THEOREM 17. *$2-(v, k, 1)$ designs are cop-win.*

PROOF. The cop begins on some vertex, and the robber then chooses some vertex. Since any two points uniquely determine a line the positions of the cop and robber uniquely determine a hyperedge. Thus, the cop can capture the robber in the round after the robber chooses a vertex. $\square$

In fact, any position that the cop occupies in a Steiner 2-design dominates the entire graph. That is, the cop is adjacent to every vertex in the hypergraph, a universal vertex in the hypergraph. We define a *corner* in a hypergraph to be a vertex x such that x and all vertices connected to x by a hyperedge are also adjacent via hyperedges to some other vertex, just as in the case of graphs. This leads us to think we

might be able to characterize cop-win hypergraphs in the same way we characterize cop-win graphs. The following theorem shows that this is not the case. The proof of this theorem shows that the result of deleting a corner in a cop-win hypergraph is not necessarily cop-win. Formally, $H - x$ is the result of removing the vertex x and all edges incident with x from the hypergraph H .

THEOREM 18. *There exist cop-win hypergraphs H , such that for all vertices x , the hypergraph $H - x$ is not cop-win.*

PROOF. We choose H to be the hypergraph defined from a fixed Steiner 2-design. Since all vertices can be dominated by any vertex in the hypergraph, all vertices in H are corners. Delete one of them, say x .

Suppose the cop begins at vertex C_1 of $H - x$. In order to avoid the cop in the first round the robber selects a vertex that is on the hyperedge that has been deleted that contained both x and C_1 . This hyperedge must exist and is unique since any two points determine a unique line. Call this point occupied by the robber R_1 .

Now, in order to capture the robber, the cop moves to some point C_2 which is not on the line determined by x and C_1 . To evade the cop the robber must move to some vertex on the deleted hyperedge determined by x and C_2 . If the parameter $k \geq 3$, then such a vertex must exist.

If there exists a vertex R_2 that is not equal to x or C_2 , on the line determined by x and C_2 , then there is a hyperedge connecting R_1 and R_2 . This hyperedge could not have been deleted since it is not

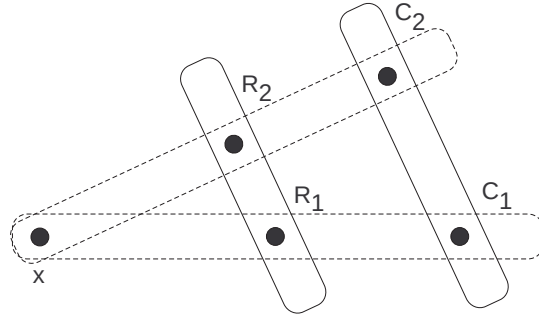


FIGURE 4.7. The deleted hyperedges give the robber an escape route.

incident with the deleted vertex x . The robber now simply moves to R_2 to evade capture. This escape strategy for the robber can be repeated indefinitely. Since the robber can escape the cop indefinitely, $H - x$ is not cop-win. $\square$

The preceding theorem gives the following corollary.

COROLLARY 19. *Cop-win hypergraphs cannot be characterized by successive deletion of corners.*

Many questions about the game of Cops and Robbers on hypergraphs remain open. The principal open problem in this area finding a characterization of cop-win hypergraphs. Many other problems about generalizing the results on graphs to hypergraphs remain open. In this chapter we give examples of 2-cop-win hypergraphs that are analogues of 2-cop-win graphs. Are there other classes of 2-cop-win hypergraphs that are not analogues of 2-cop-win graphs? Another area of interest is the asymptotic behaviour of cop-number in hypergraphs. Is there an analogue of Meyniel's conjecture for hypergraphs?

CHAPTER 5

Conclusions and Future Work

5.1. Summary

In Chapter 2 we gave some background on Meyniel's Conjecture, and discussed some results related to the conjecture. We introduced so-called Meyniel extremal graphs which realize the upper bound in the conjecture. Using the incidence graphs of projective and partial affine planes we gave new examples of such infinite families satisfying certain regularity conditions.

In Chapter 3 we considered the minimum orders of a k -cop-win graph and their relation to Meyniel's Conjecture. We characterized the cop number of all connected graphs of order 10 or less. We also discussed the theoretical underpinnings of the algorithm used to obtain the computational results. We defined two new parameters m_k and M_k that describe the minimum orders of k -cop-win graphs and related the asymptotic behaviour of these parameters to Meyniel's Conjecture. In the course of characterizing the cop number of connected graphs of order 10 or less we determined the previously unknown values of M_3 and m_3 . We also defined a new function $f_k(n)$ to describe the number of k -cop-win graphs of order n . We concluded Chapter 3 with a lower bound for the function $f_k(n)$.

In Chapter 4 we examined the game of Cops and Robbers played on hypergraphs. We discussed some terminology to describe hypergraphs and gave examples of hypergraphs and their cop number. The central result of this chapter was that the characterization of cop-win graphs by successive deletion of corners does not characterize cop-win hypergraphs.

5.2. Open Problems

Meyniel's Conjecture on the asymptotic upper bound of the cop number of a connected graph is perhaps the deepest open problem in Cops and Robbers research. The conjecture is that for a connected graph G of order n ,

$$c(G) = O(\sqrt{n}).$$

The conjecture is far from being proven for general connected graphs. In fact, even the so-called soft Meyniel's conjecture is open, which states that for a fixed constant $c > 0$ and a connected graph G of order n ,

$$c(G) = O(n^{1-c}).$$

Some work has been done towards proving the conjecture for some special classes of graphs such as random graphs [17]. However, there are many classes of graphs for which there is no proof of the conjecture. Another interesting problem is looking at the lower bounds for the cop number. Recently Prałat [16] showed that

$$c(G) \geq \sqrt{\frac{n}{2}} - n^{0.2625}.$$

It remains to be shown whether this is the best possible lower bound.

Recall from Chapter 3 that in general m_k may not be equal to M_k . It may be the case that there exists a $(k + 1)$ -cop-win graph of order n but there do not exist any k -cop-win graphs of order less than or equal to n , thus $m_k \leq M_k$. Although m_k is monotonically increasing, it is unknown whether M_k is monotonically increasing. The first few values of m_k and M_k suggests that these parameters are equal, for all $k \geq 1$, but there is no proof of this intuition.

In Chapter 3 we determined that the Petersen graph is the unique minimum order 3-cop-win graph. While an exhaustive computer search is sufficient to show this fact, a direct proof of this fact would be preferable. The minimum order of a 4-cop-win graph remains unknown. Perhaps identifying the minimum order of a 4-cop-win graph would give insight into the structure of small graphs with large cop number. The minimum order 2-cop-win and 3-cop-win graphs suggest that minimum order k -cop-win graphs are k -regular and have relatively large girth.

The study of Cops and Robbers on hypergraphs and other incidence structures is in its infancy. As such very little is known about the game on hypergraphs. The first step towards understanding the game on these structures is developing a characterization of cop-win hypergraphs. In this document all examples of 2-cop-win hypergraphs are analogues of 2-cop-win graphs. It would be interesting to see examples of 2-cop-win hypergraphs that are not analogues of 2-cop-win

graphs. Another area of interest is the asymptotic behaviour of cop-number in hypergraphs and the problem of developing an analogue of Meyniel's Conjecture for hypergraphs.

APPENDIX A

Source Code

This appendix gives implementations of three algorithms from Chapter 3. All programs given here are implemented in the C programming language and were compiled using the Gnu Compiler Collection (gcc). These programs have been successfully compiled and run on Windows XP, Windows Vista and Windows 7 using MinGW and on Ubuntu. All graph data used by these program is specified by adjacency matrices in text files, delimited by the number of vertices in the graphs. The data on connected graphs used came from Brendan McKay’s Homepage in the graph6 (.g6) format and was converted to adjacency matrix format using the program ‘showg’ from the nauty graph package. Instructions on using nauty and showg are available on Brendan McKay’s Homepage. In order to convert a list of graph in the .g6 format to the adjacency matrix format used here, run ‘showg’ from the command line as follows:

```
showg -aq input.g6 output.txt
```

The file “input.g6” specifies the graph6 file containing the graphs to be converted to adjacency format. The arguments “output.txt” is optional and allows the user to specify the output filename. The output file from the program ‘showg’ can be used as the data for any of the programs presented here. For example, the following text file is

a representation of the Petersen graph that is suitable for use by these programs:

LISTING A.1. Input format for the Petersen graph.

```

1 10
2 0100110000
3 1010001000
4 0101000100
5 0010100010
6 1001000001
7 1000000110
8 0100000011
9 0010010001
10 0001011000
11 0000101100
12 10

```

The first program is the cop-win checker based on the algorithm given by Nowakowski and Winkler in [15] to check whether a given graph is cop-win or not. The second program is a modification of the first program that computes a cop-win ordering of a graph and then simulates the game allowing the user to play the game against a cop controlled by the program. The third program is an implementation of the algorithm to check whether a graph has $c(G) \leq 2$. This implementation is similar in conception to the algorithm given by Bonato and Chiniforooshan in [4].

A.1. Cop-win Checker

```
1  /*
2  Cop-win Checker
3  Liam Baird
4  liam.baird@gmail.com
5  Input: a list of graphs in adjacency matrix format delimited
        by the number of nodes in each graph.
6  Output: a list of statements that state whether each graph in
        the list is cop-win or not.
7  */
8  #include<stdlib.h>
9  #include <time.h>
10 #include<stdio.h>
11 #define MAXNOV 10
12 #define TRUE 1
13 #define FALSE 0
14
15 int nov=0;
16
17 struct node {
18     int val;
19     struct node * next;
20 };
21 typedef struct node node;
22
23 void initialize(node *graph[]){
24     int k;
25     for(k=0;k<MAXNOV;k++){
26         graph[k] = NULL;
27     }
28 }
29
30 void printgraph(node *graph[],int graph_no){
31     int q;
32     node *curr=NULL;
33
34     if(graph_no==0){
35         printf("GRAPH:\n");
36     }else{
37         printf("GRAPH #%d:\n",graph_no);
```



```

38     }
39
40     for(q=0;q<MAXNOV;q++){
41         curr=graph[q];
42         while(curr!=NULL) {
43             if(curr->val!=q){
44                 printf("(%d,%d)\n",q,curr->val);
45             }
46             curr = curr->next ;
47         }
48     }
49 }
50
51 void deletegraph(node *graph[]){
52     int q=0;
53     node *curr=NULL;
54     node *prev=NULL;
55
56     for(q=0;q<MAXNOV;q++){
57         curr=graph[q];
58         prev=graph[q];
59
60         while(curr!=NULL) {
61             curr = curr->next;
62             free(prev);
63             prev=curr;
64         }
65         graph[q] = NULL;
66     }
67 }
68
69 void deletenode(int i,node *graph[]){
70     int q=0;
71     node *curr=NULL;
72     node *prev=NULL;
73
74     curr=graph[i];
75     prev=graph[i];
76
77     while(curr!=NULL) {
78         curr = curr->next;
79         free(prev);

```

```

80     prev=curr;
81 }
82
83 graph[i] = NULL;
84
85 for(q=0;q<MAXNOV;q++){
86     curr=graph[q];
87     prev=graph[q];
88     if(curr!=NULL){
89         if(curr->val==i){
90             graph[q]=curr->next;
91             free(curr);
92             curr=graph[q];
93             prev=graph[q];
94             curr=curr->next;
95         }else{
96             curr=curr->next;
97         }
98     }
99
100     while(curr!=NULL) {
101         if(curr->val==i){
102             prev->next=curr->next;
103             free(curr);
104             curr=prev->next;
105         }else{
106             prev=prev->next;
107             curr=curr->next;
108         }
109     }
110 }
111 }
112
113 int checkcorner(int i,int j,node *graph[]){
114     //compares two adjacency lists.
115     node *a=graph[i];
116     node *b=graph[j];
117
118     if ((a==NULL)|| (b==NULL)){
119         return(FALSE);
120     }
121

```

```

122     while (TRUE){
123         if (( a == NULL && b == NULL )||( a==NULL )){
124             return(TRUE);
125         }else if (( b == NULL )||(a->val < b->val)){
126             return(FALSE);
127         }else if (b->val < a->val){
128             if (b->next!=NULL){
129                 b=b->next;
130             }else{
131                 return(FALSE);
132             }
133         }else{
134             a = a->next;
135             b = b->next;
136         }
137     }
138 }
139
140 int hascorner(node *graph[]){
141     int i,j;
142     for(i=0;i<nov;i++){
143         for(j=0;j<nov;j++){
144             if(checkcorner(i,j,graph)){
145                 if(i!=j){
146                     return(i);
147                 }
148             }
149         }
150     }
151     return(-1);
152 }
153
154 int getsize(node *graph[]){
155     int q=0,size=0;
156     node *curr=NULL;
157
158     for(q=0;q<MAXNOV;q++){
159         curr=graph[q];
160         while(curr!=NULL) {
161             size++;
162             curr = curr->next ;
163         }

```

```

164     }
165     return(size);
166 }
167
168 int main(void) {
169     node *curr=NULL,*tail=NULL;
170     node *graph[MAXNOV];
171     char ch;
172     char str[5];
173     char filenamein[20];
174     char filenameout[20];
175     int i=0,j=0,k,c=0,q,graph_no=0,copwin_count=0;
176
177     //initialize vertices
178     initialize(graph);
179
180     //prompt for filename
181     printf("please enter graph filename\n");
182     scanf("%19s",filenamein);
183     strcpy(filenameout,filenamein);
184
185     //open file
186     FILE *input=fopen(strcat(filenamein,".txt"),"r");
187     FILE *output=fopen(strcat(filenameout,"r.txt"),"w");
188     fgets(str,5,input);
189     nov=atoi(str);
190     printf("nov = %d\n",nov);
191     graph_no++;
192
193     //get input
194     for(ch=getc(input);ch!=EOF;ch=getc(input)){
195         if(i==nov){
196             c=hascorner(graph);
197             while(c>=0){
198                 deletenode(c,graph);
199                 c=hascorner(graph);
200             }
201
202             if(getsize(graph)==1){
203                 fprintf(output,"Graph #%d is COPWIN\n",graph_no);
204                 copwin_count++;
205             }else{

```

```

206     fprintf(output,"Graph #%d is not COPWIN\n",graph_no);
207 }
208
209     deletagraph(graph);
210     i=0;
211     j=0;
212
213     //increment graph # counter
214     graph_no++;
215
216     //grab next line
217     fgets(str,5,input);
218 }
219 else if (ch=='\n'){ //new row
220     i++; //one more row
221     j=0; //start from zeroth column
222 }
223 else if ((ch=='1')||(i==j)){
224     //add edge i,j
225     curr = (node *)malloc(sizeof(node)); //allocate memory
226     curr->val = j; //set value
227     curr->next=NULL; //set next
228
229     //if this is the first edge for this node
230     if (graph[i]==NULL){
231         graph[i]=curr;
232     }
233     else{
234         tail->next=curr;
235     }
236     tail=curr;
237     //increment j
238     j++;
239 } //end add edge i,j
240 else if (ch=='0'){
241     //printf("L: ch=0\n");
242     //add no vertex
243     //increment j
244     j++;
245 }
246 }
247

```

```
248     printf("%d of %d graphs of order %d are copwin\n",
           copwin_count,graph_no-1,nov);
249     fprintf(output,"%d of %d graphs of order %d are copwin\n",
           copwin_count,graph_no-1,nov);
250
251     return(0);
252 }
```

A.2. Cops and Robbers Game Simulator

```
1  /*
2  Cops and Robber Simulator
3  Liam Baird
4  liam.baird@gmail.com
5  Input: a list of graphs in adjacency matrix format delimited
        by the number of nodes in each graph.
6  Output: a list of statements that specify whether each graph
        in the list is cop-win or not. Also
7  determines a cop-win ordering. If the graph is cop-win, then
        the program simulates a game of Cops
8  and Robbers. The program allows the user to specify the
        movements of the robber with the program
9  determining the movements of the cop using the cop-win ordering.
10 */
11
12 #include<stdlib.h>
13 #include <time.h>
14 #include<stdio.h>
15 #define MAXNOV 20
16 #define TRUE 1
17 #define FALSE 0
18
19 int nov=0;
20
21 struct node {
22     int val;
23     struct node * next;
24 };
25 typedef struct node node;
26
27 void initialize(node *graph[]){
28     int k;
29     for(k=0;k<MAXNOV;k++){
30         graph[k] = NULL;
31     }
32 }
33
34 void printgraph(node *graph[],int graph_no){
35     int q;
```

```

36     node *curr=NULL;
37
38     if(graph_no==0){
39         printf("GRAPH:\n");
40     }
41     else
42     {
43         printf("GRAPH #%d:\n",graph_no);
44     }
45
46     for(q=0;q<nov;q++){
47         curr=graph[q];
48         while(curr!=NULL) {
49             if(curr->val!=q){
50                 printf(" (%d,%d) ",q,curr->val);
51             }
52             curr = curr->next ;
53         }
54         printf("\n");
55     }
56 }
57
58 void deletegraph(node *graph[]){
59     int q=0;
60     node *curr=NULL;
61     node *prev=NULL;
62
63     for(q=0;q<MAXNOV;q++){
64         curr=graph[q];
65         prev=graph[q];
66
67         while(curr!=NULL) {
68             curr = curr->next;
69             free(prev);
70             prev=curr;
71         }
72         graph[q] = NULL;
73     }
74 }
75
76 void deletenode(int i,node *graph[]){
77     int q=0;

```



```

78     node *curr=NULL;
79     node *prev=NULL;
80
81     //delete node with index i
82     curr=graph[i];
83     prev=graph[i];
84
85     while(curr!=NULL) {
86         curr = curr->next;
87         free(prev);
88         prev=curr;
89     }
90
91     graph[i] = NULL;
92
93     for(q=0;q<MAXNOV;q++){
94         curr=graph[q];
95         prev=graph[q];
96
97         if(curr!=NULL){
98             if(curr->val==i){
99                 graph[q]=curr->next;
100                 free(curr);
101                 curr=graph[q];
102                 prev=graph[q];
103                 curr=curr->next;
104             }else{
105                 curr=curr->next;
106             }
107         }
108
109         while(curr!=NULL) {
110             if(curr->val==i){
111                 prev->next=curr->next;
112                 free(curr);
113                 curr=prev->next;
114             }
115             else{
116                 prev=prev->next;
117                 curr=curr->next;
118             }
119         }

```

```

120     }
121 }
122
123 void copygraph(node *graph[],node *copygraph[]){
124     int q;
125     node *curr=NULL;
126     node *copy=NULL;
127     node *tail=NULL;
128
129     for(q=0;q<MAXNOV;q++){
130         curr=graph[q];
131         while(curr!=NULL) {
132             //add edge i,j
133             copy = (node *)malloc(sizeof(node)); //allocate memory
134             copy->val = curr->val; //set value
135             copy->next=NULL; //set next
136
137             //if this is the first edge for this node
138             if (copygraph[q]==NULL){
139                 copygraph[q]=copy;
140             }else{
141                 tail->next=copy;
142             }
143
144             tail=copy;
145             curr=curr->next;
146         }
147     }
148 }
149
150 int checkcorner(int i,int j,node *graph[]){
151     node *a=graph[i];
152     node *b=graph[j];
153
154     if ((a==NULL)||(b==NULL)){
155         return(FALSE);
156     }
157
158     while (TRUE){
159         if (( a == NULL && b == NULL )||( a==NULL )){
160             //equal or a smaller => corner
161             return(TRUE);

```

```

162     }else if (( b == NULL )||(a->val < b->val)){
163         //b smaller than a => not corner
164         return(FALSE);
165     }else if (b->val < a->val){
166         //b larger than a => increment b
167         if (b->next!=NULL){
168             b=b->next;
169         }else{
170             return(FALSE);}
171     }else{
172         a = a->next;
173         b = b->next;
174     }
175 }
176 }
177
178 int hascorner(node *graph[],int *c,int *ci){
179     int i,j;
180
181     for(i=0;i<nov;i++){
182         for(j=0;j<nov;j++){
183             if(checkcorner(i,j,graph)){
184                 if(i!=j){
185                     *c=i;
186                     *ci=j;
187                     return(1);
188                 }
189             }
190         }
191     }
192
193     *c=-1;
194     *ci=-1;
195     return(-1);
196 }
197
198 int getsize(node *graph[]){
199     int q=0,size=0;
200     node *curr=NULL;
201
202     for(q=0;q<MAXNOV;q++){
203         curr=graph[q];

```

```

204     while(curr!=NULL) {
205         size++;
206         curr = curr->next ;
207     }
208 }
209 return(size);
210 }
211
212 void relabel(node *graph[],int labeling[]){
213     int i;
214     node *temp[MAXNOV];
215     node *curr,*prev;
216
217     copygraph(graph,temp);
218
219     //remove reflexive edges
220     for(i=0;i<nov;i++){
221
222         curr=graph[i];
223
224         if(curr!=NULL){
225             if(curr->val==i){
226                 graph[i]=curr->next;
227                 free(curr);
228                 curr=graph[i];
229                 prev=graph[i];
230                 curr=curr->next;
231             }else{
232                 curr=curr->next;
233             }
234         }
235
236         while(curr!=NULL){
237             if(curr->val==i){
238                 prev->next=curr->next;
239                 free(curr);
240                 curr=prev->next;
241             }else{
242                 prev=prev->next;
243                 curr=curr->next;
244             }
245         }

```

```

246     }
247
248     //delete node from the graph and relabel
249     for(i=0;i<nov;i++){
250         curr=graph[i];
251         prev=graph[i];
252
253         while(curr!=NULL) {
254             curr = curr->next;
255             free(prev);
256             prev=curr;
257         }
258
259         graph[i]=temp[labeling[i]];
260         curr=graph[i];
261
262         //switch labels
263         while(curr!=NULL){
264             curr->val=labeling[curr->val];
265             curr=curr->next;
266         }
267     }
268
269     //add reflexive edges
270     for(i=0;i<nov;i++){
271         curr=graph[i];
272
273         while(curr->next!=NULL) {
274             curr=curr->next;
275         }
276
277         curr = (node *)malloc(sizeof(node)); //allocate memory
278         curr->val = i;           //set value
279         curr->next=NULL;
280     }
281 }
282
283 int main(void) {
284     node *curr=NULL,*tail=NULL;
285     node *graph[MAXNOV];
286     node *pgraph[MAXNOV];
287     node *temp[MAXNOV];

```

```

288     char ch;
289     char str[5];
290     char filenamein[20];
291     char filenameout[20];
292     int cw_ord[20];
293     int cw_inv[20];
294     node *cmovelist;
295     node *rmovelist;
296     int i=0,j=0,k=0;
297     int c=0,ci=0;
298     int t=0,q=0,graph_no=0,copwin_count=0;
299     int cmovecount=0;
300     int r_move=-1,c_move=-1;
301     int r_move_valid;
302     int F[MAXNOV][MAXNOV];
303     int f[MAXNOV];
304     int temp_f[MAXNOV];
305
306     //initialize F and f
307     for(c=0;c<MAXNOV;c++){
308         f[c]=-1;
309         for(t=0;t<MAXNOV;t++){
310             F[c][t]=-1;
311         }
312     }
313     c=0;
314     t=0;
315
316     //initialize vertices
317     initialize(graph);
318     initialize(pgraph);
319     initialize(temp);
320
321     //prompt for filename
322     printf("please enter the name of a file containing the graph
        you want to play on\n");
323     scanf("%19s",filenamein);
324
325     //open file
326     FILE *input=fopen(strcat(filenamein,".txt"),"r");
327
328     //read first line

```

```

329     fgets(str,5,input);
330     nov=atoi(str);
331     printf("The number of vertices is %d\n",nov);
332     graph_no++;
333
334     //processing input
335     for(ch=getc(input);ch!=EOF;ch=getc(input)){
336         if(i==nov){ //end processing for this graph
337             copygraph(graph,pgraph);
338             printf("the graph is: ");
339             hascorner(graph,&c,&ci);
340
341             //check graph for corner
342             while(c>=0){
343                 f[c]=ci;
344                 cw_ord[t]=c;
345                 t++;
346                 deletenode(c,graph);
347                 hascorner(graph,&c,&ci);
348             }
349
350             //get last remaining element of graph
351             for(q=0;q<MAXNOV;q++){
352                 curr=graph[q];
353                 if(curr!=NULL){
354                     cw_ord[t]=q;
355                 }
356             }
357
358             if(getsize(graph)==1){
359                 printf("COPWIN\n");
360                 copwin_count++;
361             }else{
362                 printf("NOTCOPWIN\n");
363             }
364
365             deletegraph(graph);
366             i=0;
367             j=0;
368             //increment graph # counter
369             graph_no++;
370             //grab next character which is a newline

```

```

371     fgets(str,5,input);
372     }else if (ch=='\n'){
373         i++;
374         j=0;
375     }else if ((ch=='1')||(i==j)){
376         //add edge i,j
377         curr = (node *)malloc(sizeof(node));
378         curr->val = j;
379         curr->next=NULL;
380
381         //if this is the first edge for this node
382         if (graph[i]==NULL){
383             graph[i]=curr;
384         }else{
385             tail->next=curr;
386         }
387         tail=curr;
388
389         //increment j
390         j++;
391     }else if (ch=='0'){
392         j++;
393     }
394 }
395
396 //RESET variables
397 i=0;
398 j=0;
399 c=0;
400
401 //invert cw_ord
402 for(i=0;i<nov;i++){
403     cw_inv[cw_ord[i]]=i;
404 }
405
406 node *prev;
407 copygraph(pgraph,temp);
408
409 //print initial graph
410 printf("\nORIGINAL ");
411 printgraph(pgraph,0);
412 printf("\n");

```



```

413
414 //relabel graph with cw_ord
415     for(i=0;i<nov;i++){
416         //delete node from the pgraph
417         curr=pgraph[i];
418         prev=pgraph[i];
419
420         while(curr!=NULL) {
421             curr = curr->next;
422             free(prev);
423             prev=curr;
424         }
425
426         //assign new node lists
427         pgraph[i]=temp[cw_ord[i]];
428         curr=pgraph[i];
429
430         //change labels
431         while(curr!=NULL){
432             curr->val=cw_inv[curr->val];
433             curr=curr->next;
434         }
435     }
436
437
438 //make temporary f for computing f reordering
439     for(i=0;i<nov;i++){
440         temp_f[i]=f[i];
441     }
442
443 //print relabeled graph
444     printf("\nRELABELED ");
445     printgraph(pgraph,0);
446     printf("\n");
447
448 //FIND F and f
449 //reassign f[]
450     for(i=0;i<nov;i++){
451         f[cw_inv[i]]=cw_inv[temp_f[i]];
452     }
453
454 //assign F[0][]=F_1

```

```

455     for(i=0;i<nov;i++){
456         for(j=0;j<nov;j++){
457             F[i][j]=j;
458         }
459     }
460
461     //assigning F[1][] to F[nov][]
462     for(i=1;i<nov;i++){
463         for(j=i;j<nov;j++){
464             F[j][i-1]=f[i-1];
465         }
466     }
467
468     for(i=1;i<nov;i++){
469         for(j=0;j<nov;j++){
470             if (F[i][j]<i){
471                 for(t=i;t<nov;t++){
472                     F[t][j]=f[F[t][j]];
473                 }
474             }
475         }
476     }
477
478
479     /* game simulation*/
480
481     //pick the start vertex
482     k=nov-1;
483     c_move=k;
484     k--;
485     printf("the cop is starting on vertex %d\nwhich vertex do
         you want to start on?:\n",c_move);
486     scanf("%d",&r_move);
487
488     //get initial robber position
489     while((r_move<0)|| (r_move>=nov)|| r_move==c_move){
490         printf("That is not a valid move, please select another
         vertex\n");
491         scanf("%d",&r_move);
492     }
493
494     printf("you are moving to vertex: %d\n",r_move);

```

```

495
496 while(TRUE){
497     //check possible cop moves
498     cmovelist=pgraph[c_move];
499     curr=cmovelist;
500     cmovecount=0;
501
502     printf("the vertices available to the cop are:\n");
503
504     while(curr!=NULL){
505         printf("%d\n",curr->val);
506         if(curr->val==r_move){
507             F[k][r_move]=r_move;
508         }
509         cmovecount++;
510         curr=curr->next;
511     }
512     scanf("");
513
514     //compute cop move
515     c_move=F[k][r_move];
516     k--;
517     printf("COP MOVES TO:%d\n",c_move);
518
519     //check win condition
520     if(c_move==r_move){
521         printf("the cop wins!\n");
522         break;
523     }
524
525     //check possible robber moves
526     rmovelist=pgraph[r_move];
527     curr=rmovelist;
528
529     printf("the vertices available to you the robber are:\n");
530
531     while(curr!=NULL){
532         printf("%d\n",curr->val);
533         curr=curr->next;
534     }
535     printf("where do you want to move?\n");
536     scanf("%d",&r_move);

```

```

537
538     curr=rmovelist;
539     r_move_valid=FALSE;
540
541     while(curr!=NULL){
542         if(curr->val==r_move){
543             r_move_valid=TRUE;
544         }
545         curr=curr->next;
546     }
547
548     while(!r_move_valid){
549         printf("That is not a valid move, please select another
550             vertex\n");
551         scanf("%d",&r_move);
552         curr=rmovelist;
553
554         //check that the robber move is in valid
555         while(curr!=NULL){
556             if(curr->val==r_move){
557                 r_move_valid=TRUE;
558             }
559             curr=curr->next;
560         }
561         printf("you are moving to vertex: %d\n",r_move);
562     }
563
564     printf("press ctrl-c to quit\n");
565     while(TRUE){}
566     return(0);
567 }

```

A.3. 2-cop-win Checker

```
1  /*
2  2-cop-win Checker
3  Liam Baird
4  liam.baird@gmail.com
5  Input: a list of graphs in adjacency matrix format delimited
        by the number of nodes in each graph.
6  Output: a list of statements that state whether each graph in
        the list has  $c(G) \leq 2$  or not.
7  */
8
9  #include <stdlib.h>
10 #include <time.h>
11 #include <stdio.h>
12 #include <string.h>
13 #define MAXNOV 10
14 #define TRUE 1
15 #define FALSE 0
16 int nov=0;
17
18 //defines a node
19 struct node {
20     int val1;
21     int val2;
22     struct node * next;
23 };
24 typedef struct node node;
25
26 void initialize(node *graph[]){
27     int k;
28     for(k=0;k<MAXNOV;k++){
29         graph[k] = NULL;
30     }
31 }
32
33 void initializep(node *pgraph[MAXNOV][MAXNOV]){
34     int j,k;
35     for(k=0;k<MAXNOV;k++){
36         for(j=0;j<MAXNOV;j++){
37             pgraph[k][j] = NULL;
```

```

38     }
39 }
40 }
41
42 void initializeeff(node *eff[MAXNOV][MAXNOV]){
43     int i,j;
44     for(i=0;i<MAXNOV;i++){
45         for(j=0;j<MAXNOV;j++){
46             eff[i][j] = NULL;
47         }
48     }
49 }
50
51 void printgraph(node *graph[],int graph_no){
52     int q;
53     node *curr=NULL;
54
55     if(graph_no==0){
56         printf("GRAPH:\n");
57     }else{
58         printf("GRAPH #%d:\n",graph_no);
59     }
60
61     for(q=0;q<MAXNOV;q++){
62         curr=graph[q];
63         while(curr!=NULL) {
64             if(curr->val1!=q){
65                 printf("(%d,%d)\n",q,curr->val1);
66             }
67             curr = curr->next ;
68         }
69     }
70 }
71
72 void printpgraph(node *graph[MAXNOV][MAXNOV],int graph_no){
73     int i=0,j=0;
74     node *curr=NULL;
75     printf("P.GRAPH #%d:\n",graph_no);
76
77     for(i=0;i<MAXNOV;i++){
78         for(j=0;j<MAXNOV;j++){
79             curr=graph[i][j];

```

```

80         while(curr!=NULL) {
81             if((curr->val1!=i)|| (curr->val2!=j)){
82                 printf("(%d,%d), (%d,%d)\n", i, j, curr->val1, curr->
                        val2);
83             }
84             curr = curr->next ;
85         }
86     }
87 }
88 }
89
90 void printeff(node *eff [MAXNOV] [MAXNOV], int nov){
91     int i=0, j=0;
92     node *curr=NULL;
93     printf("EFF:\n");
94     for(i=0; i<nov; i++){
95         for(j=0; j<nov; j++){
96             curr=eff[i][j];
97             printf("F([%d,%d]): ", i, j);
98             while(curr!=NULL){
99                 printf("%d ", curr->val1);
100                 curr = curr->next ;
101             }
102             printf("\n");
103         }
104     }
105 }
106
107 void printadj(node *adj [MAXNOV] [MAXNOV], int nov){
108     int i=0, j=0;
109     node *curr=NULL;
110     printf("ADJ:\n");
111
112     for(i=0; i<nov; i++){
113         for(j=0; j<nov; j++){
114             curr=adj[i][j];
115             printf("F([%d,%d]): ", i, j);
116             while(curr!=NULL){
117                 printf("%d ", curr->val1);
118                 curr = curr->next ;
119             }
120             printf("\n");

```

```

121     }
122 }
123 printf("\n");
124 }
125
126 void deletograph(node *graph[]){
127     int q=0;
128     node *curr=NULL;
129     node *prev=NULL;
130
131     for(q=0;q<MAXNOV;q++){
132         curr=graph[q];
133         prev=graph[q];
134
135         while(curr!=NULL) {
136             curr = curr->next;
137             free(prev);
138             prev=curr;
139         }
140         graph[q] = NULL;
141     }
142 }
143
144 void deletepgraph(node *graph[MAXNOV][MAXNOV]){
145     int i=0,j=0;
146     node *curr=NULL;
147     node *prev=NULL;
148
149     for(i=0;i<MAXNOV;i++){
150         for(j=0;j<MAXNOV;j++){
151             curr=graph[i][j];
152             prev=graph[i][j];
153
154             while(curr!=NULL){
155                 curr = curr->next;
156                 free(prev);
157                 prev=curr;
158             }
159
160             graph[i][j] = NULL;
161         }
162     }

```



```

163 }
164
165 void deleteeff(node *graph[MAXNOV][MAXNOV]){
166     int i=0,j=0;
167     node *curr=NULL;
168     node *prev=NULL;
169
170     for(i=0;i<MAXNOV;i++){
171         for(j=0;j<MAXNOV;j++){
172             curr=graph[i][j];
173             prev=graph[i][j];
174
175             while(curr!=NULL){
176                 curr = curr->next;
177                 free(prev);
178                 prev=curr;
179             }
180
181             graph[i][j] = NULL;
182
183         }
184     }
185 }
186
187 void deletenode(int i,node *graph[]){
188     int q=0;
189     node *curr=NULL;
190     node *prev=NULL;
191
192     curr=graph[i];
193     prev=graph[i];
194
195     while(curr!=NULL) {
196         curr = curr->next;
197         free(prev);
198         prev=curr;
199     }
200
201     graph[i] = NULL;
202
203     for(q=0;q<MAXNOV;q++){
204

```

```

205     curr=graph[q];
206     prev=graph[q];
207
208     if(curr!=NULL){
209         if(curr->val1==i){
210             graph[q]=curr->next;
211             free(curr);
212             curr=graph[q];
213             prev=graph[q];
214             curr=curr->next;
215         }
216         else{
217             curr=curr->next;
218             //prev=prev->next;
219         }
220     }
221
222     while(curr!=NULL) {
223         if(curr->val1==i){
224             prev->next=curr->next;
225             free(curr);
226             curr=prev->next;
227         }
228         else
229         {
230             prev=prev->next;
231             curr=curr->next;
232         }
233     }
234
235
236 }
237
238 }
239
240 int isadj(int i, int j,node *graph[]){
241     //determine whether or not i and j are adjacent in graph
242     node *curr=NULL;
243     curr=graph[i];
244
245     while(curr!=NULL) {
246         if (curr->val1==j){

```

```

247         return(TRUE);
248     }
249     curr=curr->next;
250 }
251
252     return(FALSE);
253 }
254
255 int iseq(int i, int j){
256     if (i==j){
257         return(TRUE);
258     }
259
260     return(FALSE);
261 }
262
263 void makeeff(node *eff[MAXNOV][MAXNOV], node *adj[MAXNOV][
    MAXNOV], int nov){
264     int i=0, j=0, k=0, v=0;
265     node *curr=NULL;
266     node *prev=NULL;
267     node *list=NULL;
268
269     for(i=0; i<nov; i++){
270         for(j=0; j<nov; j++){
271             prev=eff[i][j];
272             list=adj[i][j];
273
274             for(k=0; k<nov; k++){
275                 if(list!=NULL){
276                     v=list->val1;
277                 }else{
278                     v=MAXNOV+10;
279                 }
280
281                 if(k!=v){
282                     //add node with value k
283                     curr=(node *)malloc(sizeof(node));
284                     curr->val1=k;
285                     curr->val2=-1;
286                     curr->next=NULL;
287

```

```

288         if (eff[i][j]==NULL){
289             eff[i][j]=curr;
290         }else{
291             prev->next=curr;
292         }
293
294         prev=curr;
295     }else{
296         list=list->next;
297     }
298 }
299 }
300 }
301 }
302
303 void makeadjacency(node *adj[MAXNOV][MAXNOV],node *graph[
    MAXNOV],int nov){
304     int i=0,j=0,va=0,vb=0;
305     node *curr=NULL;
306     node *prev=NULL;
307     node *list_a=NULL;
308     node *list_b=NULL;
309
310     for(i=0;i<nov;i++){
311         for(j=0;j<nov;j++){
312             prev=adj[i][j];
313             list_a=graph[i];
314             list_b=graph[j];
315
316             while((list_a!=NULL)|| (list_b!=NULL)){
317
318                 if(list_a!=NULL){
319                     va=list_a->val1;
320                 }else{
321                     va=MAXNOV+10;
322                 }
323
324                 if(list_b!=NULL){
325                     vb=list_b->val1;
326                 }else{
327                     vb=MAXNOV+10;
328                 }

```

```

329
330         curr=(node *)malloc(sizeof(node));
331         curr->next=NULL;
332
333         if(va<vb){
334             curr->val1=va;
335             list_a=list_a->next;
336         } else if(vb<va){
337             curr->val1=vb;
338             list_b=list_b->next;
339         } else if(va==vb){
340             curr->val1=va;
341             list_a=list_a->next;
342             list_b=list_b->next;
343         }
344
345         if (adj[i][j]==NULL){
346             adj[i][j]=curr;
347         }else{
348             prev->next=curr;
349         }
350         prev=curr;
351     }
352 }
353 }
354 }
355
356 int intersect(node *adj[MAXNOV][MAXNOV],node *eff[MAXNOV][
    MAXNOV],int i,int j, int k, int l){
357     node *curr_a=adj[i][j];
358     node *curr_f=eff[k][l];
359     node *prev_f=NULL;
360     node *curr=NULL;
361     node *prev=NULL;
362     int changed=FALSE;
363     int va=0,vf=0;
364
365     eff[k][l]=NULL;
366     prev_f=curr_f;
367
368     while(curr_f!=NULL){
369         if(curr_a!=NULL){

```

```

370     va=curr_a->val1;
371 }else{
372     changed=TRUE;
373     return(changed);
374 }
375
376 vf=curr_f->val1;
377 curr=(node *)malloc(sizeof(node));
378 curr->next=NULL;
379
380 if(va<vf){
381     curr_a=curr_a->next;
382 } else if(vf<va){
383     curr_f=curr_f->next;
384     free(prev_f);
385     prev_f=curr_f;
386     changed=TRUE;
387 } else if(va==vf){
388     curr->val1=va;
389     curr_a=curr_a->next;
390     curr_f=curr_f->next;
391
392     if (eff[k][l]==NULL){
393         eff[k][l]=curr;
394     }else{
395         prev->next=curr;
396     }
397     prev=curr;
398 }
399 }
400 return(changed);
401 }
402
403 int main(void) {
404     node *curr=NULL,*tail=NULL,*curr_a=NULL,*tail_a=NULL,*
        tail_p=NULL;
405     node *graph[MAXNOV];
406     node *(eff[MAXNOV][MAXNOV]);
407     node *(adj[MAXNOV][MAXNOV]);
408     node *(pgraph[MAXNOV][MAXNOV]);
409     char ch;
410     char str[5];

```

```

411     char filenamein[20];
412     char filenameout[20];
413     int i=0,j=0,k,c=0,q,graph_no=0,twocopwin_count=0;
414     int x=0,y=0,u=0,v=0,u1=0,u2=0,v1=0,v2=0;
415     int flag1=FALSE,flag2=FALSE,changed=TRUE,twocopwin=FALSE;
416
417     //initialize data structures;
418     initialize(graph);
419     initializep(pgraph);
420     initializeeff(eff);
421     initializeeff(adj);
422
423     //prompt for filename
424     printf("please enter graph filename\n");
425     scanf("%19s",filenamein);
426     strcpy(filenameout,filenamein);
427
428     //open file
429     FILE *input=fopen(strcat(filenamein,".txt"),"r");
430     FILE *output=fopen(strcat(filenameout,"r.txt"),"w");
431
432     //read first character, should be a # (need to fix, input
       could be "10" ie not a single char
433     fgets(str,5,input);
434     nov=atoi(str);
435     graph_no++;
436
437     //loop through characters
438     for(ch=getc(input);ch!=EOF;ch=getc(input)){
439         if(i==nov){
440             //make product graph
441             for(x=0;x<nov;x++){
442                 for(y=0;y<nov;y++){
443                     for(u=0;u<nov;u++){
444                         for(v=0;v<nov;v++){
445                             if((iseq(x,u)&&iseq(y,v))||(isadj(x,u,graph)&&
                                   iseq(y,v))||(iseq(x,u)&&isadj(y,v,graph))||(
                                   isadj(x,u,graph)&&isadj(y,v,graph))){
446                                 curr = (node *)malloc(sizeof(node));
447                                 curr->val1 = u;
448                                 curr->val2 = v;
449                                 curr->next=NULL;

```

```

450
451         //add the node
452         if (pgraph[x][y]==NULL){
453             pgraph[x][y]=curr;
454         }else{
455             tail_p->next=curr;
456         }
457         tail_p=curr;
458     }
459 }
460 }
461 }
462 }
463
464 makeadjacency(adj,graph,nov);
465 makeeff(eff,adj,nov);
466 printadj(adj,nov);
467 printeff(eff,nov);
468 changed=TRUE;
469
470 while(changed==TRUE){
471     //LOOP THROUGH vertices u1 v1
472     for(u1=0;u1<nov;u1++){
473         for(u2=0;u2<nov;u2++){
474             curr=pgraph[u1][u2];
475             if(curr!=NULL){
476                 //pick out the vertex this is connected to
477                 v1=curr->val1;
478                 v2=curr->val2;
479                 //find intersection of eff[u1][u2] and adj[v1][v2]
480                 flag1=intersect(adj,eff,u1,u2,v1,v2);
481                 //find intersection of eff[v1][v2] and adj[u1][u2]
482                 flag2=intersect(adj,eff,v1,v2,u1,u2);
483
484                 //CHECK FOR EMPTY EFF
485                 if((eff[u1][u2]==NULL) || (eff[v1][v2]==NULL)){
486                     flag1=FALSE;
487                     flag2=FALSE;
488                     changed=FALSE;
489                 }
490
491                 //check for flags

```



```

492         if((flag1==TRUE)|| (flag2==TRUE)){
493             changed=TRUE;
494         }
495     }
496 }
497 }
498 printf("eff, nov");
499 }
500
501 twocopwin=FALSE;
502 for(x=0;x<nov;x++){
503     for(y=0;y<nov;y++){
504         if(eff[x][y]==NULL){
505             twocopwin=TRUE;
506         }
507     }
508 }
509
510 if(twocopwin==TRUE){
511     twocopwin_count++;
512     printf("GRAPH %d is 2cw\n",graph_no);
513 }else{
514     printf("GRAPH %d is NOT 2cw\n",graph_no);
515 }
516
517 deleteeff(eff);
518 deletetgraph(graph);
519 deletetpgraph(pgraph);
520 deleteeff(adj);
521 i=0;
522 j=0;
523 //increment graph # counter
524 graph_no++;
525 fgets(str,5,input);
526 }else if (ch=='\n'){
527     i++;
528     j=0;
529 }else if ((ch=='1')||(i==j)){
530     //add edge i,j
531     curr = (node *)malloc(sizeof(node)); //allocate memory
532     curr->val1 = j;           //set value
533     curr->next=NULL;         //set next

```

```

534
535     //if this is the first edge for this node
536     if (graph[i]==NULL){
537         graph[i]=curr;
538     }else{
539         tail->next=curr;
540     }
541     tail=curr;
542     j++;
543 }else if (ch=='0'){
544     j++;
545 }
546 }
547
548 printf("%d of %d graphs of order %d are <=2 copwin\n",
        twocopwin_count,graph_no-1,nov);
549 fprintf(output,"%d of %d graphs of order %d are copwin\n",
        twocopwin_count,graph_no-1,nov);
550 return(0);
551 }

```


Bibliography

- [1] M. Aigner, M. Fromme, A game of cops and robbers, *Discrete Applied Mathematics* **8** (1984) 1–12.
- [2] A. Berarducci, B. Intrigila, On the cop number of a graph, *Advances in Applied Mathematics* **14** (1993) 389–403.
- [3] C Berge, *Hypergraphs: Combinatorics of Finite Sets*, North Holland, 1989.
- [4] A. Bonato, E. Chiniforooshan, P. Prałat, Cops and Robbers from a distance, *Theoretical Computer Science* **411** (2010) 3834–3844.
- [5] A. Bonato, R.J. Nowakowski, *The Game of Cops and Robbers on Graphs*, American Mathematical Society, Providence, Rhode Island, 2011.
- [6] P.J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, Cambridge, 1995.
- [7] P. Chebyshev, Mémoire sur les nombres premiers, Mém. Acad. Sci. St. Pétersbourg **7** (1850) 17–33.
- [8] E. Chiniforooshan, A better bound for the cop number of general graphs, *Journal of Graph Theory* **58** (2008) 45–48.
- [9] N.E. Clarke, R.J. Nowakowski, Cops, robber, and traps, *Utilitas Mathematica* **60** (2001) 91–98.
- [10] P. Frankl, On a pursuit game on Cayley graphs, *Combinatorica* **7** (1987) 67–70.
- [11] P. Frankl, Cops and robbers in graphs with large girth and Cayley graphs, *Discrete Applied Mathematics* **17** (1987) 301–305.
- [12] A. Frieze, M. Krivelevich, P. Loh, Variations on Cops and Robbers, accepted to *Journal of Graph Theory*.
- [13] L. Lu, X. Peng, On Meyniel’s conjecture of the cop number, Preprint 2011.

- [14] B.D. McKay, Combinatorial Data, *Brendan McKay's Home Page* published electronically at <http://cs.anu.edu.au/~bdm/data/>, Accessed June 20, 2011.
- [15] R.J. Nowakowski, P. Winkler, Vertex-to-vertex pursuit in a graph, *Discrete Mathematics* **43** (1983) 235–239.
- [16] P. Prałat, When does a random graph have constant cop number?, *Australasian Journal of Combinatorics* **46** (2010), 285–296.
- [17] P. Prałat, N. Wormald, Meyniel's conjecture holds in random graphs, Preprint 2011.
- [18] A. Quilliot, Jeux et pointes fixes sur les graphes, *Thèse de 3ème cycle*, Université de Paris VI, 1978, 131–145.
- [19] A. Quilliot, *Problèmes de jeux, de point Fixe, de connectivité et de représentation sur des graphes, des ensembles ordonnés et des hypergraphes*, Thèse d'Etat, Université de Paris VI, 1983, 131–145.
- [20] A. Quilliot, A short note about pursuit games played on a graph with a given genus, *J. Combin. Theory (B)* **38** (1985) 89–92.
- [21] A. Scott, B. Sudakov, A new bound for the cops and robbers problem, Preprint 2011.
- [22] N.J.A. Sloane, Sequences A000088 and A001349, *The On-Line Encyclopedia of Integer Sequences* published electronically at <http://oeis.org>, 2010.
- [23] V.I. Voloshin, *Introduction to Graph and Hypergraph Theory*, Nova Science Publishers, Inc., 2009.
- [24] D.B. West, *Introduction to Graph Theory, 2nd edition*, Prentice Hall, 2001.