Three Models of the Game of Cops and Robber

by

© **Caleb Jones**

A thesis submitted to the Department of Mathematics and Statistics in partial fulfillment of the requirements for the degree of BSc. (Honours) in Pure Mathematics.

Department of Mathematics and Statistics

Memorial University

April 2021

St. John's, Newfoundland and Labrador, Canada

Abstract

We analyze the original [6, 7], surrounding [3], and containment [4] versions of the game of cops and robber on graphs. In particular, we investigate the relationship between the parameters $c(G)$, $s(G)$, and $\xi(G)$ that correspond to these three games. We present an algorithm similar to one from [1] that computes $\xi(G)$ for a given graph G, and use it to obtain results for a number of generalized Petersen graphs. As well, we introduce a new framework for analyzing all three versions of the game that is simpler in that it only requires that we observe one object traversing a graph.

Dedicated to my parents.

Lay summary

A graph is a collection of points (called vertices) and lines (called edges) where each line connects two specific points. Graphs are useful for describing many different concepts. A helpful analogy is as follows: The vertices represent cities, and cities A and B are connected by an edge if there is a direct flight from city A to city B and from city B to city A. In this paper, we analyze a cops and robber game that is played on graphs. The game has two players: C, who controls multiple cops, and R, who controls one robber. Naturally, the cops and robber occupy vertices of the graph. If a cop or robber occupies some vertex u, they may only move onto a vertex v if there is an edge between u and v (in this case u and v are "adjacent"). The game starts with C placing all the cops on vertices of the graph. Then R places the robber on a vertex of the graph. Then C and R take turns alternately. On C's turn, they move every cop once each and on R's turn they move the robber. A cop or robber may "move" by either staying put or moving to an adjacent vertex. Unsurprisingly, it is C's goal to catch the robber by having at least one cop occupy the same vertex as them. It is R's goal to never get caught. In mathematics we are interested in optimization, so we assume each player has perfect information and plays perfectly. Thus, once we pick a graph to play on and a number of cops, the winner of the game is settled. Given a graph G, we wish to find the least number of cops that can win on the graph, called the cop number of G and denoted $c(G)$. Two variations of interest are the surrounding and containment versions of the game which alter the rules for how the cops may move and/or the conditions under which they win. The least number of cops that can win on G is $s(G)$ for the surrounding game and $\xi(G)$ for the containment game. In this thesis we analyze the relationship between the three games. For example, if k cops win in the containment version of the game, do k cops win in the surrounding version? We introduce an algorithm that computes $\xi(G)$ for any graph G, and record the $\xi(G)$ values for a family of graphs. We also introduce a new framework with which to analyze the games that could potentially be useful for solving open questions such as the nature of C's winning strategy.

Acknowledgements

I would like to acknowledge the help of my supervisor Dr. David Pike, who directed the research for this paper. He conceived the ideas for many results in this paper and offered guidance on how to prove them. He also provided editorial comments on the manuscript, such as how best to adhere to mathematical conventions, and suggested inclusions that would enhance the paper. I would also like to acknowledge Dr. Candemir Cigsar, who offered valuable feedback on the manuscript and ensured milestones for the project were completed on time.

Statement of contribution

Dr. David Pike directed the research, offering expertise on which avenues would be best to explore and conceiving key results that should be included in the paper. He also offered expertise that greatly helped me with the manuscript and the coding aspect of the project. I (Caleb Jones) wrote the manuscript and code (found in sections A.0.1 and A.0.2). I also conceived and proved some results independently. Multiple results were conceived by Dr. Pike, after which I either proved them or followed the proof given by Dr. Pike.

Table of contents

List of tables

List of figures

Chapter 1

Introduction

The game of cops and robber on graphs is a vertex pursuit game that has attracted much attention. The game was introduced by Nowakowski and Winkler [6] and Quilliot [7]. Given a simple connected graph G and a number of cops k, the game proceeds as follows. Let C be the player controlling the cops and R be the player controlling the robber. C chooses a starting vertex for each of the k cops, and then R chooses their starting vertex. Then C and R take turns alternately, with a turn for C consisting of each cop either staying put or moving to an adjacent vertex, and a turn for R being defined analogously. Both C and R play optimally and are aware of the positions of all of the cops and the robber. The game ends when a cop "catches" the robber by occupying the same vertex as him. C wins if they have a strategy to catch the robber after finitely many turns, and R wins if they have a strategy to infinitely evade capture. Given a graph G, a natural question is what is the least number of cops required to catch the robber? This parameter is known as the cop number of the graph, $c(G)$. The following conjecture was posed by Henri Meyniel in 1985 and was first published in [5]: given a connected graph G of order n, $c(G) \in O(\sqrt{n})$. Currently, Meyniel's conjecture is one of the biggest open questions in this field of study.

One avenue for research that could shed light on this question is analyzing variations on the original game. Two variations of interest are the surrounding and containment versions of the game.

In the surrounding game, which was introduced in [3], R does not lose if a cop moves onto the same vertex as the robber. Instead, R is compelled to move the robber to a vertex that is not occupied by a cop on their next turn. Additionally, R cannot move the robber onto a cop's vertex. In order to win, the cops need to "surround" the robber by occupying each vertex adjacent to the robber. The least number of cops that can do this is the surrounding

cop number of the graph, denoted $s(G)$. It is known that $c(G) \leq s(G)$ [3].

In the containment version of the game, which was introduced in [4], the cops occupy edges of the graph and can stay put or move to an adjacent edge each turn. The robber still occupies vertices and moves using edges. R cannot move the robber across an edge that is occupied by a cop. In order to win, the cops must "contain" the robber by occupying every edge incident to the robber's vertex. The least number of cops that can do this is the containment number of the graph, denoted $\xi(G)$. It is known that $c(G) \leq \xi(G)$ [4].

1.0.1 Preliminary Examples

We first analyze some simpler families of graphs; cycles and trees. Figures 1.1 and 1.2 show a cycle and a tree respectively.

Lemma 1.0.1. *If G is a cycle on 4 or more vertices, then $c(G) = s(G) = \xi(G) = 2$.*

Proof. When playing on the cycle with 3 vertices, clearly $c(G) = 1$ and $s(G) = \xi(G) = 2$. Thus, we insist that $|V(G)| \geq 4$.

In the original game, a single cop cannot win on a cycle with 4 or more vertices. Given the cop's initial placement, R can win by choosing an initial vertex that is of maximum distance from the cop. If the cop moves clockwise (respectively counterclockwise) around the graph, R responds by also moving clockwise (respectively counterclockwise). The resulting configuration is identical to the initial configuration in which R and C are of maximum distance from one another. Thus, R can always maintain a distance of at least 1 from the cop.

In the original game, two cops win on a cycle with 4 or more vertices. C begins by placing both cops at the same vertex. R will then choose a vertex of distance at least 2 from the cops. On each of C's turns, they move both cops in opposite directions around the graph. Clearly this strategy results in R's capture after a finite number of moves. So $c(G) = 2$.

In the surrounding game, a single cop cannot win on a cycle. This is because each vertex on the cycle has degree 2, and $s(G) \geq \delta(G)$ for all graphs G, where $\delta(G) = \min\{\deg(v) : v \in V(G)\}$ [3].

In the surrounding game, two cops can win on a cycle. C begins by placing both cops on the same initial vertex. R chooses any other initial vertex. Then, on each of C's turns, the two cops move in opposite directions around the graph. If this strategy would compel a cop to move onto R's vertex, that cop instead stays put for that turn. Then, throughout the

Figure 1.1: A Cycle

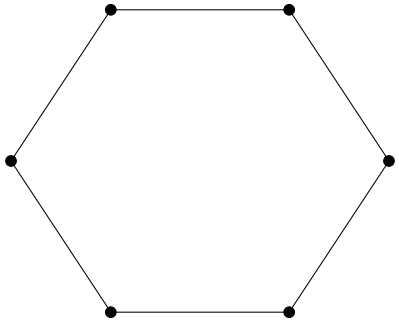
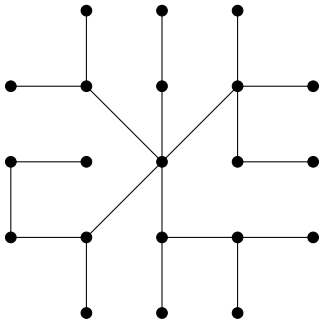


Figure 1.2: A Tree



game, R's movement is confined to a path in G starting at one cop's vertex and ending at the other's, with R's vertex somewhere on the path. With each move, C shortens this path until it contains only 3 vertices, at which point R has been surrounded. So $s(G) = 2$.

In the containment game, a single cop cannot win on a cycle. Each vertex has degree 2, and $\xi(G) \geq \Delta(G)$ for all graphs G, where $\Delta(G) = \max\{\deg(v) : v \in V(G)\}$ [4].

In the containment game, two cops win on a cycle. C begins by placing them both on the same initial edge. R may place the robber on any initial vertex. On each of C's turns, they move the cops in opposite directions around the graph. If this strategy would compel a cop to move from an edge uv to an edge vw while the robber occupies v, then this cop instead stays put for the turn. This confines R's movement to a path whose first and last edges are occupied by cops, and the path includes both edges incident to R's vertex. With each move, C shortens this path until it contains only two edges, at which point R has been contained. So $\xi(G) = 2$. □

Lemma 1.0.2. *If G is a tree then $c(G) = 1$.*

Proof. The initial positions chosen by C and R do not affect this outcome. C's winning

strategy is simply to move along the shortest path to R's vertex each turn. Then, since the graph is finite, eventually R will be forced to move onto a leaf. C's next move will be to occupy the only other vertex adjacent to this leaf, so C wins on the next turn. Thus, $c(G) = 1$. □

Lemma 1.0.3. *If G is a tree not of the form $K_{1,m}$, $m \in \mathbb{N}$, then $s(G) = 2$.*

Proof. If we are playing on $K_{1,m}$ then C can place a single cop on the vertex with degree m. Then, no matter which initial vertex R chooses, they will immediately be surrounded. So $s(K_{1,m}) = 1$ for any $m \in \mathbb{N}$.

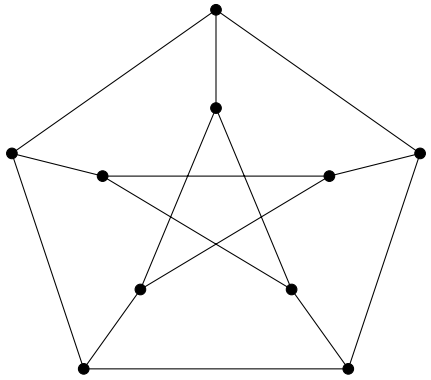
Now, let G be a tree not of the form $K_{1,m}$. When C plays with only 1 cop, R wins the game using the following strategy. R chooses their initial vertex to be any non-leaf vertex a that is adjacent to another non-leaf vertex b. Such vertices must exist since G is not of the form $K_{1,m}$. If C ever moves onto a, then R moves to b. Then if C moves onto b, R moves back to a, and so on. So one cop cannot surround the robber.

Now, let C play with two cops and show that C has a winning strategy. C initially places the cops on vertices u and v that are adjacent. On each turn, both cops move along a shortest path to the robber, so the cops always occupy adjacent vertices. Let us call the cop closest to R the "front" cop, and the cop furthest from R the "rear" cop. Since this graph has a cop number of 1, eventually the front cop will land on R's vertex. If R is on a leaf at this point then the game is over. If not, then R is forced to move, but they cannot move onto the vertex occupied by the rear cop. Suppose R moves to some vertex w. Then the front cop can move onto w, and the rear cop can move onto R's previously occupied vertex. Thus, R is forced to move "away" from the cops once again. Since the graph is finite, R will eventually end up on a leaf with the front cop occupying its only neighbour. Thus, $s(G) = 2$. □

Lemma 1.0.4. *If G is a tree then $\xi(G) = \Delta(G)$.*

Proof. We know from [4] that fewer than $\Delta(G)$ cops will lose the containment game. So, we must show that exactly $\Delta(G)$ cops have a winning strategy. Let each cop start the game on the same edge. Then, on each of C's turns, each cop moves along the shortest path towards R. Since G is finite, the cops will eventually manage to occupy an edge incident to R's vertex. Suppose R is on a vertex v and each cop is on an edge uv. R cannot move onto u. Also, R cannot stay put, since if they do then on C's turn the cops can move onto each edge incident to v to win the game. This is the case for every vertex in G since there is no vertex in G with a higher degree than the number of cops, $\Delta(G)$. Thus, R is forced to move

Figure 1.3: The Petersen Graph, $GP(5, 2)$

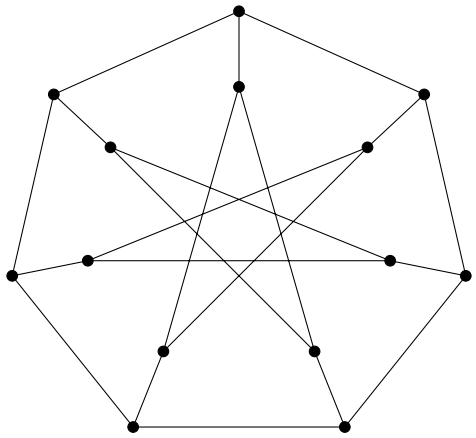


to some other adjacent vertex w. Then, C moves every cop onto the edge vw, and R is once again forced to move "away" from the cops. Since G is finite, R will eventually be forced to move onto a leaf. C can then move onto the incident edge to win the game. □

1.0.2 Generalized Petersen Graphs

The Petersen graph can be seen in Figure 1.3. It has five "outer" vertices that make up an outer cycle, and another five "inner" vertices that make up the inner portion of the graph. The edges that connect the outer and inner vertices are called spokes. Notice how the inner star-shape is formed by connecting inner vertices that are not immediately next to each other. Similar graphs can be constructed with a different number of inner and outer vertices, and with a different formation of the inner star-shape. These are known as generalized Petersen graphs, denoted $GP(n, k)$. A graph $GP(n, k)$ has n inner vertices and n outer vertices (with the outer cycle and spokes analogous to the Petersen graph). The parameter k determines how the inner star-shape is formed. Given an inner vertex u, we count k inner vertices clockwise, and connect u to the vertex v that we counted to. Then we repeat this process starting at v until the inner portion of the graph is complete. The Petersen graph is $GP(5, 2)$. Note that generalized Petersen graphs are 3-regular. Also, given $n \in \mathbb{N}$, there exist generalized Petersen graphs $GP(n, k)$ for $k \in \{1, 2, ..., \lceil \frac{n}{2} - 1 \rceil\}$. $GP(7, 3)$ can be seen in Figure 1.4.

Figure 1.4: A Generalized Petersen Graph, $GP(7, 3)$



Chapter 2

Methodology

A natural goal is to determine some kind of relationship between $s(G)$ and $\xi(G)$. Currently, we know of graphs with $s(G) < \xi(G)$ and $s(G) = \xi(G)$ (for example the generalized Petersen graphs $GP(5,2)$ and $GP(5,1)$ respectively), but there are no known graphs with $s(G) > \xi(G)$. Thus, the following conjecture is an open question.

Conjecture 2.0.1. *$s(G) \leq \xi(G)$ for all graphs G.*

In [4], it was proven that $c(G) \leq \xi(G)$ for all graphs G by taking a winning strategy for the cops playing the containment version of the game, and converting it into a winning strategy for the cops in the original game. However, such a strategy cannot be used to prove Conjecture 2.0.1. Before we can show this, let us summarize the proof of $c(G) \leq \xi(G)\ \forall G$ given in [4].

Let $\mathcal{A}$ be a winning strategy for the $\xi(G)$ cops playing the containment version of the game on a graph G. Let $\mathcal{B}$ be the strategy for $\xi(G)$ cops playing the original version of the game on G that is derived from $\mathcal{A}$ in the following way.

> *Step 1.* If $\mathcal{A}$ places the cops on edges $e_1 = u_1v_1, e_2 = u_2v_2, ..., e_{\xi(G)} = u_{\xi(G)}v_{\xi(G)}$ as their first move, then $\mathcal{B}$ places the cops on vertices $v_1, v_2, ..., v_{\xi(G)}$ as their first move (we may choose arbitrarily between the endpoints of the corresponding edges).
>
> *Step 2.* Whenever $\mathcal{A}$ directs a cop to move from edge uv to edge vw, $\mathcal{B}$ directs the corresponding cop to move onto vertex v. Suppose R is on a vertex x and a cop occupies an edge incident to x, call it $e = xy$. In the containment version of the game, R cannot cross e because it is occupied by a cop. Similarly, in the

original game R cannot cross e because there is either a cop on x (in which case R has already lost) or there is a cop on y (and thus if R crosses e they end up on y with the cop). Therefore, the robber does not have any moves available to them in the original game that are not available to them in the containment game. That is, no matter what move R makes in the original game, strategy $\mathcal{A}$ will have a valid response for the cops when R makes that same move in the containment game. This valid move in strategy $\mathcal{A}$ can then be translated into a move for strategy $\mathcal{B}$.

Step 3. The final step in $\mathcal{A}$ is to place a cop on each edge $e = xy$ where R occupies vertex x. Then, the final step of $\mathcal{B}$ is to either place at least one cop on x, capturing the robber immediately, or to place one cop on each neighbour of x, in which case R loses on the next turn.

By following the above strategy $\mathcal{B}$, $\xi(G)$ cops can catch the robber in the original game, so $c(G) \leq \xi(G)$ for all graphs G [4].

This proof strategy does not work when attempting to prove $s(G) \leq \xi(G)$, and the fault can be found in *step 2*. Suppose R is on a vertex x and a cop occupies an edge incident to x, call it $e = xy$. In the containment version of the game, R cannot cross e because it is occupied by a cop. However, the robber can sometimes cross this edge in the surrounding version of the game. The corresponding cop in the surrounding game occupies either vertex x or vertex y. If they occupy vertex x, then instead of immediately losing, R is instead forced to move to a vertex not occupied by a cop. So, this cop does not prevent the robber from crossing the edge $e = xy$. If R chooses to move onto y, strategy $\mathcal{A}$ has no response, and thus the proof fails. This problem cannot be solved by deliberately choosing the starting vertices of the cops in *step 1*. To see this, suppose R occupies vertex x, and strategy $\mathcal{A}$ directs a cop to move from edge zx to edge xy. Then, the corresponding cop in the surrounding game must move onto vertex x. Now, in the surrounding game, R can move from x to y, which is something they cannot do in the containment game. This shows us that when the cops in the surrounding game attempt to follow strategy $\mathcal{A}$, R has moves available to them in the surrounding game that are not available to them in the containment game, and thus $s(G) \leq \xi(G)$ is not assured.

However, we can establish a weaker upper bound on $s(G)$ in terms of $\xi(G)$ using the proof strategy above.

Theorem 2.0.1. *For all graphs G, $s(G) \leq 2\xi(G)$.*

Proof. Let G and H be two identical graphs. Let $\xi(G)$ cops play the containment version of the game on G, and let $2\xi(G)$ cops play the surrounding version of the game on H. Let us denote the players controlling the cops on G and H by C_G and C_H respectively. We label the cops of C_G as $C_1, C_2, ..., C_{\xi(G)}$, and the cops of C_H as $C_1', C_1'', C_2', C_2'', ..., C_{\xi(G)}', C_{\xi(G)}''$. We know C_G has a winning strategy, and we must use this strategy to derive a winning strategy for C_H. When R moves on H, R will make the same move on G and C_G will move according to their winning strategy (which yields a win for the cops no matter how R moves). Then, C_H will determine their move based on the move made by C_G. First, to determine the starting placement of the cops on H we simply observe the starting placement of the cops on G according to their winning strategy. Suppose C_G chooses the following initial placements; C_1 on $e_1 = u_1v_1$, C_2 on $e_2 = u_2v_2$,..., and $C_{\xi(G)}$ on $e_{\xi(G)} = u_{\xi(G)}v_{\xi(G)}$. C_H places their corresponding cops on the end vertices of these edges, so C_i' is placed on u_i and C_i'' is placed on v_i for each $i \in \{1, 2, ..., \xi(G)\}$.

Now, when R makes a move (on both graphs), C_G responds with a move according to their winning strategy. C_H will "follow" C_G in this way; if C_i moves from edge uv to vw, then C_i' and C_i'' move so that one occupies v and the other occupies w (this is always possible since previously one was on u and the other was on v). It is necessary that R does not have more movement options on H than he does on G and this is indeed true. Suppose that R is on a vertex u in G and there is a cop occupying the edge uv. In H, this corresponds to R being at u, and a cop being on each of u and v. In both games, R is unable to move from u to v. In fact, in the game on H, R's movement is even more restricted, as the above scenario forces R to move off of u, whereas R can sometimes stay put in this scenario in the game on G. C_H can still follow a winning strategy however, because C_G has a move in response to R moving off u that is part of their winning strategy. The game on G will eventually end with R occupying some vertex v and at least one cop on every edge adjacent to v. Suppose there are k cops occupying these edges at the end of the game. Then, the game in H will end with R on the same vertex v, k cops on v, and at least one cop occupying each vertex adjacent to v. Therefore C_H wins by following the above strategy. That is, $2\xi(G)$ cops always have a winning strategy in the surrounding game on a graph G, so $s(G) \leq 2\xi(G)$. □

A successful containment strategy in which no cop ever needs to "jump" over the robber by moving from uv to vu while the robber occupies v can be translated into a successful surrounding strategy.

Lemma 2.0.1. *If m cops have a strategy to contain R on G such that no cop ever needs to move from uv to vw while R occupies v, then $s(G) \leq m$. In particular, if $m = \xi(G)$ then $s(G) \leq \xi(G)$.*

Proof. Suppose m cops have a winning strategy on G in the containment version of the game such that no cop ever needs to move from uv to vw while R occupies v. Call this winning strategy $\mathcal{A}$. Let m cops play the surrounding version of the game on G. We must construct a winning strategy $\mathcal{B}$ for these cops using $\mathcal{A}$.

If $\mathcal{A}$ places the cops initially on edges $e_1 = u_1v_1$, $e_2 = u_2v_2$,..., $e_m = u_mv_m$ than $\mathcal{B}$ places the surrounding cops initially on vertices v_1, v_2,..., v_m.

Whenever $\mathcal{A}$ directs a cop to move from edge uv to vw, $\mathcal{B}$ directs the corresponding cop in the surrounding game to move onto v. If a cop in the containment game ever moves onto an edge incident to R's vertex, we know they did not do so by "jumping" over R. Thus, if R occupies v and a cop playing containment moves onto uv, $\mathcal{B}$ directs the corresponding cop playing the surrounding game to move onto u. Therefore, R is unable to cross the edge uv on their next turn in both games.

The final step in $\mathcal{A}$ is to place a cop on each edge $e = xy$ where R occupies vertex x. Since none of the cops did this by "jumping" over R, the corresponding cops following strategy $\mathcal{B}$ now place themselves on all vertices y adjacent to the robber's vertex, x. Since m cops can surround R on G, $s(G) \leq m$. □

Note that in the above strategy $\mathcal{B}$, no cop ever moves onto R's vertex in order to force them to move. The above lemma could also be useful for proving Conjecture 2.1. If $\xi(G)$ cops can always win on G in the containment version of the game without ever jumping over R, then Lemma 2.1 would imply that Conjecture 2.1 is true.

2.0.1 An Algorithmic Approach

Since a proof for Conjecture 2.0.1 is not obvious, a natural direction to take is to search for a counterexample, that is, a graph G with $s(G) > \xi(G)$. The following theorem and algorithm (both adapted from those in [1]) can be used to calculate $\xi(G)$, thus making our search for a counterexample much easier.

We will use the notation $L(G)$ to represent the line graph of a graph G. To construct $L(G)$, for each edge e in G we add a corresponding vertex v_e to $L(G)$. Then, we say two vertices v_{e_1} and v_{e_2} are adjacent in $L(G)$ if and only if the corresponding edges e_1 and e_2 are adjacent in G.

Note that in the containment game, each configuration of k cops can be represented by a k-tuple of edges of G. If $\forall i \in \{1, 2, ..., k\}$ cop C_i occupies edge e_i, then this configuration of

cops can be written $(e_1, e_2, ..., e_k)$. Therefore, the set of all configurations of cops is exactly the same as the vertex set of $\boxtimes^k L(G)$, where $\boxtimes$ is the strong product of graphs. Also, if configuration T_1 is $(e_1, e_2, ..., e_k)$ and configuration T_2 is $(e'_1, e'_2, ..., e'_k)$, then T_1 and T_2 are adjacent configurations if and only if $e_i = e'_i$ or e_i is adjacent to e'_i in G $\forall i \in \{1, 2, ..., k\}$ (and thus the cops can move from T_1 to T_2 in one move). Equivalently, by the definition of $\boxtimes$, configurations T_1 and T_2 are adjacent if and only if their corresponding vertices in $\boxtimes^k L(G)$ are adjacent.

Also note that in the following theorem, we use a slight abuse of notation. Each configuration T is a k-tuple of edges in G, not a set of edges in G. Thus, when we write $G - T$, we mean the spanning subgraph of G induced by removing all edges listed in T from G.

Theorem 2.0.2. *Let $k \in \mathbb{N}$. Then, $\xi(G) > k$ if and only if there exists a function $\psi : V(\boxtimes^k L(G)) \to \mathcal{P}(V(G))$ with the following three properties.*

(1) $\forall T \in V(\boxtimes^k L(G))$, $\psi(T) \neq \emptyset$

(2) $\forall T \in V(\boxtimes^k L(G))$, $\psi(T) \subseteq V(G) \setminus (A_T \cup B_T)$ *where A_T is the set of all vertices in G that are already contained by T, and B_T is the set of vertices in G that can be contained by T in one move.*

(3) $\forall T_1T_2 \in E(\boxtimes^k L(G))$, $\psi(T_1) \subseteq N_{G-T_2}[\psi(T_2)]$

Proof. ($\Leftarrow$): Suppose such a function ψ exists for k cops. We must show that the robber has a strategy to win the game. Let T_0 be the initial positions of the cops. In round 0, the robber moves into any vertex of $\psi(T_0)$. This is possible by property (1), as $\psi(T_0)$ is nonempty. By property (2), the robber is not contained on round 0 because no vertices in $\psi(T_0)$ are contained by T_0. Also, the robber is not contained on round 1, because by property (2), the cops in T_0 cannot contain any vertex of $\psi(T_0)$ in only one move. In general, if the cops are at T and the robber moves into $\psi(T) \subseteq V(G)$, then the robber will be safe until their next move. This is an immediate result of property (2).

We argue that for all rounds $t \geq 0$, the robber can move into $\psi(T_t)$ to avoid capture on round $t + 1$. Suppose this claim is true for all $t \leq m$. We will prove that it is also true at $t = m + 1$. At the end of round m the cops are in configuration T_m and the robber is at a vertex in $\psi(T_m)$. The cops move into an adjacent configuration T_{m+1} at the start of round $m + 1$, and the robber must be able to move into $\psi(T_{m+1})$ to avoid being contained on the next round. By property (3), we know that $\psi(T_m) \subseteq N_{G-T_{m+1}}[\psi(T_{m+1})]$. The robber is in $\psi(T_m)$, so he is able to move into $\psi(T_{m+1})$ via an edge in $G - T_{m+1}$ (*i.e.* an edge not occupied

by a cop of T_{m+1}) at the end of round $m+1$ to escape capture at the start of the next round.

($\Rightarrow$): Suppose $\xi(G) > k$ and let k cops play in G. Since the robber has a winning strategy, he will win no matter the starting configuration of the cops. For all configurations of cops T, define $\psi(T)$ to be the set of all vertices in G such that if the cops start at configuration T and the robber chooses his initial vertex to be in $\psi(T)$, then the robber ultimately wins the game. Since the robber must win for all starting configurations of cops, we know that $\psi(T) \neq \emptyset$ for all T, so (1) is proven.

To prove property (2), let the cops be at configuration T, and suppose the robber has just moved into $\psi(T)$. We will show that the robber is not in $A_T \cup B_T$. If the robber is in A_T then he is already contained and thus he loses the game. This is a contradiction since the robber must win by our assumption that $\xi(G) > k$, so he is not in A_T. Thus $A_T \cap \psi(T) = \emptyset$ for all configurations T. Now, if the robber has moved onto a vertex in B_T then he can be contained by the cops of T in one move, so the robber loses once the cops move. This is a contradiction since the robber must win the game. Therefore $B_T \cap \psi(T) = \emptyset$ for all configurations T. If the robber has just moved into $\psi(T)$ and is following a winning strategy he cannot also be in $A_T \cup B_T$, therefore $\psi(T) \subseteq V(G) \setminus (A_T \cup B_T)$ for all configurations T, so (2) is proven.

Finally, to prove (3), let $T_1 \subseteq E(G)$ and $T_2 \subseteq E(G)$ be two adjacent configurations of cops, and fix $z \in \psi(T_1)$. By our supposition, the robber wins if he plays according to the strategy that if the cops are at T, then the robber must move to $\psi(T)$. Let the cops be at T_1 and the robber be at z. The cops move from T_1 to T_2 which is legal since these are adjacent configurations. Since the robber has a winning strategy, he must be able to move from z to some safe vertex z' ($z = z'$ is possible) via an edge that is not occupied by the cops of T_2. We know that z' is in $\psi(T_2)$ since this is the collection of all safe vertices for the robber when the cops are at T_2. Since the robber was at an arbitrary vertex in $\psi(T_1)$ and must be able to move into $\psi(T_2)$ without crossing an edge in T_2, it must be true that every vertex in $\psi(T_1)$ is in the neighborhood in $G - T_2$ of $\psi(T_2)$. That is, $\psi(T_1) \subseteq N_{G-T_2}[\psi(T_2)]$. T_1 and T_2 were arbitrary as well, so (3) is proven. □

Theorem 2.0.3. *Let G be a connected graph and let $\psi : V(\boxtimes^k L(G)) \to \mathcal{P}(V(G))$ satisfy properties (2) and (3) from Theorem 2.0.2. If $\exists T_1 \in V(\boxtimes^k L(G))$ with $\psi(T_1) = \emptyset$ then $\psi(T) = \emptyset$ $\forall T \in V(\boxtimes^k L(G))$.*

Proof. Since G is connected, so are $L(G)$ and $\boxtimes^k L(G)$. Since $\boxtimes^k L(G)$ is connected, there exists a finite walk $\mathcal{W} = T_1, T_2, ..., T_m$ in $\boxtimes^k L(G)$ that starts at T_1 and visits each $T \in$

$V(\boxtimes^k L(G))$ at least once. For our inductive hypothesis, suppose that $\psi(T_i) = \emptyset$ for some $i \in \{1, 2, ..., m\}$. The next vertex in the walk is T_{i+1}, so $T_iT_{i+1} \in E(\boxtimes^k L(G))$. Then, by property (3), we have $\psi(T_{i+1}) \subseteq N_{G-T_i}(\psi(T_i)) = N_{G-T_i}(\emptyset) = \emptyset$ and thus $\psi(T_{i+1}) = \emptyset$. By induction we therefore have $\psi(T_i) = \emptyset \ \forall i \in \{1, 2, ..., m\}$. Since $\mathcal{W}$ contains every vertex of $\boxtimes^k L(G)$, the result follows. □

We will now present an algorithm that can be used to determine the containment number of a graph G. Again, it was adapted from an algorithm in [1]. It requires as input the graph G as well as a proposed number of cops k. Using this, the algorithm constructs $\psi(T)$ for every configuration of cops T based on Theorem 2.0.2. Line 1 of the algorithm initializes the $\psi(T)$ values such that property (2) of Theorem 2.0.2 is satisfied. Lines 2-7 continuously alter the $\psi(T)$ values to ensure that property (3) is satisfied. Finally, lines 8-12 check whether or not property (1) is satisfied. If not (*i.e.* $\exists T$ with $\psi(T) = \emptyset$), then by Theorem 2.0.2, the cops win, and the algorithm returns $\xi(G) \leq k$. Otherwise, the robber wins, and the algorithm returns $\xi(G) > k$.

input : graph $G = (V, E)$, number of cops $k \in \mathbb{N}$
output: either $\xi(G) \leq k$ or $\xi(G) > k$

1 initialize $\psi(T)$ to $V(G) \setminus (A_T \cup B_T)$ for all configurations T
2 **repeat**
3 **forall** *adjacent configurations* T *and* T' **do**
4 $\psi(T) \leftarrow \psi(T) \cap N_{G-T'}[\psi(T')]$
5 $\psi(T') \leftarrow \psi(T') \cap N_{G-T}[\psi(T)]$
6 **end**
7 **until** *the value of* ψ *is unchanged*;
8 **if** $\exists$ *configuration* T *such that* $\psi(T) = \emptyset$ **then**
9 **return** $\xi(G) \leq k$ (cops win)
10 **else**
11 **return** $\xi(G) > k$ (robber wins)
12 **end**

Algorithm 1: check potential $\xi(G)$ value

Theorem 2.0.4. *Algorithm 1 runs in time* $O(2^k k m^k n^{k+2} + k m^{3k} n^3)$ *where* $n = |V(G)|$, $m = |E(G)|$, *and* k *is the number of cops.*

Proof. The processes in lines 1, 2-7, and 8-12 run independently of one another, and in order, and therefore their individual run-times contribute additively to the total run-time

of the algorithm. Also note that we are assuming that the adjacency matrix for $\boxtimes^k L(G)$ is available to us, and so doing a look-up to see if two vertices in $\boxtimes^k L(G)$ are adjacent (i.e. two configurations of cops are adjacent) takes constant time. Also, let us compute $N_G(u)$ for all $u \in V(G)$ as a one-time preprocessing. Finally, note that throughout this proof, when calculating the run-time of steps in the algorithm, certain terms and/or coefficients will often be ignored, as they are insignificant to the overall run-time when n, m, or k are large.

First we analyze line 1, which consists of three steps that happen one-after-the-other, and therefore contribute additively to the total run-time of line 1.

The first step is to initialize $\psi(T)$ to $V(G)$ for all $T \in \boxtimes^k L(G)$. We must traverse each of the m^k $\psi(T)$ values and assign n elements to each. This requires us to perform nm^k operations, which takes time $O(nm^k)$.

Now we must remove A_T from $\psi(T)$ for each $T \in \boxtimes^k L(G)$. We must traverse each of the m^k $\psi(T)$ values, and then for each $\psi(T)$, traverse each of its elements (of which there are at most n) to determine which of them must be removed. If we are looking at element u of our current $\psi(T)$ value, we must check if vertex u is contained by configuration T. This requires that we check each edge incident to u, of which there are at most $n-1$, and then check if each edge belongs to T, which requires at most k checks each time. Step 2 of line 1 therefore takes time $O(m^k \cdot n \cdot (n-1) \cdot k) = O(kn^2 m^k)$.

Now we must remove B_T from $\psi(T)$ for each $T \in \boxtimes^k L(G)$. We must traverse each of the m^k $\psi(T)$ values, and then for each $\psi(T)$, traverse each of its elements (of which there are at most n) to determine which of them must be removed. For some fixed element u in our current $\psi(T)$, we must determine if vertex u in G can be contained by the cops in configuration T in one move. First, we must check all configurations T' that are adjacent to T, as these are the configurations the cops can reach in one move. We already checked if T contains u in the previous step, so we don't have to check T itself here. By Theorem 4.0.4 (proven in Chapter 4.0.2), if the cops at T occupy edges $e_1 = u_1v_1, e_2 = u_2v_2, ..., e_k = u_kv_k$ then C has

$$\Big(\deg(u_1) + \deg(v_1) - 1\Big)\Big(\deg(u_2) + \deg(v_2) - 1\Big) \cdots \Big(\deg(u_k) + \deg(v_k) - 1\Big)$$

moves. The number of adjacent configurations of cops is therefore at most

$$\Big((n-1) + (n-1) - 1\Big)\Big((n-1) + (n-1) - 1\Big) \cdots \Big((n-1) + (n-1) - 1\Big) - 1 = (2n-3)^k - 1$$

since $\deg(v)$ is at most $n-1$ for each $v \in V(G)$. We subtract 1 at the end because we

are not counting the move where the cops stay put on T. Now, for each of these adjacent configurations T', we must check if u is contained by T'. This requires that we check each edge incident to u, of which there are at most $n-1$, and then check if each edge belongs to T', which requires at most k checks each time. Step 3 of line 1 therefore takes time $O(m^k \cdot n \cdot ((2n-3)^k - 1) \cdot (n-1) \cdot k) = O(2^k k m^k n^{k+2})$.

The total run time of line 1 is therefore $O(nm^k + kn^2m^k + 2^k k m^k n^{k+2}) = O(2^k k m^k n^{k+2})$.

Now we analyze the run time of lines 2-7. First we consider the outer loop: "repeat... until the value of ψ is unchanged." There are m^k different $\psi(T)$ values, each with at most n elements. Each time this loop is executed, some elements are removed from at least one $\psi(T)$ value, except on the last loop. The worst case scenario is that each execution of the outer loop removes just one element from one $\psi(T)$ value. Then, the outer loop would be executed $nm^k + 1$ times (the $+1$ is because one final loop is required to go through every $\psi(T)$ value without making any changes). So the outer loop is executed at most $nm^k + 1$ times.

Now we consider the inner loop which traverses all pairs of adjacent configurations of cops, that is, the edge set of $\boxtimes^k L(G)$. The maximum number of edges on $\boxtimes^k L(G)$ is $\binom{|V(\boxtimes^k L(G))|}{2} = \binom{m^k}{2} = \frac{m^{2k}-m^k}{2}$. Traversing this set therefore takes time $O(\frac{m^{2k}-m^k}{2}) = O(m^{2k})$.

Finally, we look at the intersection step found within the two loops. We must first compute $N_{G-T'}[\psi(T')]$. We look at each element $u \in \psi(T')$, of which there are at most n. Then, we add the vertices of $N_{G-T'}[u]$ to $N_{G-T'}[\psi(T')]$. This requires that we add u to $N_{G-T'}[\psi(T')]$, and then traverse the vertices of $N_G(u)$, of which there are at most $n-1$. Then, for each vertex v in this neighbourhood, we determine if the edge uv is in T' (if so, we do not add v to $N_{G-T'}[\psi(T')]$), which requires that we check each of the k elements of T'. This takes at most time $O(n \cdot (n-1) \cdot k) = O(kn^2)$. Now, set intersection takes time $O(x)$ where x is the size of the largest set involved. We are intersecting $\psi(T)$ with $N_{G-T'}[\psi(T')]$, so this will take time $O(n)$. Therefore line 4 takes time $O(kn^2 + n) = O(kn^2)$. Line 5 takes the same amount of time, so the total time of lines 4 and 5 is $O(kn^2 + kn^2) = O(kn^2)$.

Now, the outer loop, inner loop, and intersection step all happen one-inside-the-other, and therefore their run times contribute multiplicatively to the total run time of lines 2-7. The total run time of lines 2-7 is therefore $O((nm^k + 1) \cdot m^{2k} \cdot kn^2) = O(km^{3k}n^3)$.

We now consider the run time of lines 8-12. We know that if one $\psi(T)$ is empty then they are all empty, so it suffices to check one $\psi(T)$ value. We must determine if $u \notin \psi(T)\ \forall u \in V(G)$ is true, which requires at most n checks. This takes time $O(n)$.

Finally, to get the total run time of the algorithm, we sum the run times of lines 1, 2-7,

and 8-12. The total run time is therefore $O(2^k km^k n^{k+2} + km^{3k} n^3 + n) = O(2^k km^k n^{k+2} + km^{3k} n^3)$. □

2.0.2 Cartesian Products

We define the Cartesian product of graphs, $G\square H$. If $V(G) = \{v_1, v_2, ..., v_n\}$ and $V(H) = \{u_1, u_2, ..., u_m\}$, then $|V(G\square H)| = nm$ and the vertices of $G\square H$ are indexed by ordered pairs (v_i, u_j). A vertex $a = (v, u)$ is adjacent to a vertex $b = (v', u')$ in $G\square H$ if (i) $v = v'$ and $uu' \in E(H)$, or (ii) $vv' \in E(G)$ and $u = u'$. A helpful visualization is as follows. Let us rearrange G such that its vertices are in a vertical line, and H such that its vertices are in a horizontal line. The vertex set of $G\square H$ can then be visualized as a grid that is n vertices tall and m vertices wide. If we observe the subgraph induced by taking all the vertices in a column and all edges where both endpoints are vertices in this column, we get a graph that is isomorphic to G, call it a "copy" of G. Similarly, if we observe the subgraph induced by taking a row of vertices and all edges with both endpoints in this row, we get a copy of H. All edges in $G\square H$ belong to exactly one of these copies. That is, there are no edges that connect two vertices in different rows *and* different columns.

Given two graphs G and H, the following useful results regarding $c(G\square H)$ and $s(G\square H)$ are known: $c(G\square H) \leq c(G) + c(H)$ [8], and $s(G\square H) \leq s(G) + s(H)$ [3]. We present the analogous result for the containment version of the game as a conjecture.

Conjecture 2.0.2. *For all connected graphs G and H, $\xi(G\square H) \leq \xi(G) + \xi(H)$.*

Conjecture 2.0.2 is true for the k-track $K_2\square C_k$, since all k tracks have containment number 3 [4], $\xi(K_2) = 1$, and $\xi(C_k) = 2\ \forall k \geq 3$. Also, The containment numbers of $P_3\square P_3$, $C_3\square C_3$, and $K_{1,3}\square K_{1,3}$ were calculated using Algorithm 1, and none of these graphs are a counterexample to Conjecture 2.0.2.

The strategies used to prove $c(G\square H) \leq c(G) + c(H)$ and $s(G\square H) \leq s(G) + s(H)$ cannot be used to prove Conjecture 2.0.2. Before we can show why this is the case we will investigate how R and C traverse the vertices and edges of a graph of the form $G\square H$ when playing the containment version of the game.

Suppose the robber is traversing the vertices of $G\square H$. Note that each vertex in this graph belongs to exactly one copy of G and one copy of H. If they move from a vertex in row i, column j to a vertex in row i column k, then they have moved from a vertex in a copy of G to a vertex in another copy of G using an edge in a copy of H, and they are still in the same copy of H. If they move from a vertex in row i, column j to a vertex in row k, column

j, then they have moved from one copy of H to another copy of H via an edge in a copy of G, and they have remained in the same copy of G. Thus, whenever the robber moves to a new vertex, they either enter a new copy of G but remain in the same copy of H, or move to a new copy of H but remain in the same copy of G. They cannot move to a new copy of G and a new copy of H in a single move.

Now, suppose a cop is traversing the edges of $G\Box H$. Note that each edge of this graph belongs to either a single copy of G or a single copy of H. If the cop is in a copy of G and wishes to move to a new copy, they must first move onto an edge of some copy of H, since all edges in a copy of G only connect pairs of vertices in that same copy of G. That is, a cop cannot move from an edge in one copy of G to an edge in another copy of G in one move since they must first move onto an edge in a copy of H (and vice-versa).

[8] and [3] use the same strategy to prove $c(G\Box H) \leq c(G) + c(H)$ and $s(G\Box H) \leq s(G) + s(H)$ respectively. To illustrate this strategy, we summarize the proof of $s(G\Box H) \leq s(G) + s(H)$ given in [3].

> *Step 1.* We must show that $s(G) + s(H)$ cops have a winning strategy on $G\Box H$. The cops' first objective is to have a team of size $s(G)$ enter the same copy of G as R, and a team of size $s(H)$ enter the same copy of H as R. If on R's turn, R stays within their copy of G, the G-seeking cops can move closer to this copy of G (by traversing edges in some copy of H). Similarly, if on R's turn, R stays in their copy of H, the H-seeking cops can move closer to this copy of H (by traversing edges in some copy of G). As discussed above, no matter how R moves, they cannot move to a new copy of G and a new copy of H in a single move. Since G is finite, both teams will achieve their goal after finitely many moves. Without loss of generality, assume the G-seeking cops succeed first.
>
> *Step 2.* We may now assume that all the $s(G)$ G-seeking cops occupy vertices in the same copy of G as R. If R makes a move in which they remain in the same copy of G, the G-seeking cops advance their strategy to surround R within the copy of G. If R moves to a new copy of G, the G-seeking cops mirror R's move. Thus, once the G-seeking cops have succeeded in their initial goal, they can move into R's copy of G each turn for the rest of the game. Meanwhile, the H-seeking cops continue to move closer to R's copy of H. We know the H-seeking cops will succeed after finitely many moves.
>
> *Step 3.* We may now assume that all the $s(H)$ H-seeking cops occupy vertices in

the same copy of H as R. The G-seeking and H-seeking cops now follow similar strategies to surround R in their copies of G and H respectively. If R moves to a new copy of G, the G-seeking cops follow, and the H-seeking cops advance their strategy to surround R in their copy of H. Similarly, if R moves to a new copy of H then the H-seeking cops follow, the G-seeking cops advance their strategy to surround R in their copy of G. After finitely many moves, one of the teams will have surrounded R in their copy of G or H. Without loss of generality, let the G-seeking cops succeed first. Then, R cannot move to a new copy of H since this would require that they traverse an edge in their current copy of G. But they are surrounded in their current copy of G, so they cannot traverse any of these edges. Thus, R is restricted to their current copy of H. Whenever R moves to a new copy of G (via an edge in some copy of H), the G-seeking cops follow, immediately surrounding R in the new copy of G. Meanwhile, the $s(H)$ H-seeking cops surround R in this copy of H. Once R is surrounded by both the G- and H-seeking cops, R is surrounded in $G\square H$.

The above proof strategy does not work when we consider the containment version of the game. Suppose the $\xi(G)$ G-seeking cops have occupied edges in the same copy of G as R. Then, for the above strategy to work, they must be able to (i) advance their strategy to contain R if R remains in this copy of G, and (ii) follow R into a new copy of G if R moves to a new copy of G. However, when R moves into a new copy of G, the G-seeking cops can only move onto edges that belong to copies of H. As discussed above, edges in two different copies of G (or H) are never adjacent. Thus, the cops cannot move from edges in one copy of G to edges in a new copy of G in one turn. They must take one turn to move onto edges in copies of H. If on R's next turn they stay within this new copy of G, then the G-seeking cops cannot advance their strategy to contain R, since they currently occupy edges in copies of H.

Thus, we leave Conjecture 2.0.2 as an open problem.

Chapter 3

Computational Results

If $\Delta(G)$ is the maximum degree of all vertices on G then $\Delta(G) \leq \xi(G)$ [4]. Thus, we can narrow the search for a counterexample to Conjecture 2.0.1 by restricting our search to graphs with $c(G) < s(G)$ and $\Delta(G) < s(G)$. This is because if $c(G) = s(G)$ or $\Delta(G) \geq s(G)$ then $s(G) \leq \xi(G)$ and thus G is not a counterexample. Generalized Petersen graphs are a useful family of graphs for this investigation, as they all have $\Delta(G) = 3$, and many have $c(G) = 3 < 4 = s(G)$ [3]. We are therefore looking for a generalized Petersen graph with $\xi(G) = 3$ and $s(G) = 4$. Clearly Generalized Petersen graphs of the form $GP(n, 1)$ will not yield any counterexamples as these graphs are identical to the k-track (with $k = n$). Thus, [4] and [3] tell us $\xi(GP(n, 1)) = 3 = s(GP(n, 1))\ \forall n$.

We can further narrow our search due to Propositions 3.0.1 and 3.0.2 (from [4]).

Proposition 3.0.1 ([4])**.** *If G is a δ-regular graph ($\delta > 2$) with girth at least 5, then G is not containable. That is, if G is δ-regular, and the length of the shortest cycle on G is at least 5, then δ cops cannot contain the robber on G.*

Proposition 3.0.2 ([4])**.** *On any δ-regular ($\delta > 2$) graph G, the deliberate and intelligent robber will never be contained by δ cops at a vertex which is not part of a 3- or 4-cycle.*

Proposition 3.0.1 tells us that only generalized Petersen graphs with girth 3 or 4 can possibly be counterexamples. Although graphs of the form $GP(n, 1)$ have girth 4, they will not yield any counterexamples since $s(G) = 3 = \Delta(G)$ for this family of graphs [3]. The only other family of generalized Petersen graphs that have girth less than 5 are those generalized Petersen graphs with either a 3-cycle or a 4-cycle on the inner vertices. These occur on graphs $GP(n, k)$ with $\frac{n}{k} = 3$ and $\frac{n}{k} = 4$ respectively. Therefore if a generalized Petersen graph that is a counterexample exists, it must be of the form $GP(n, k)$ with $\frac{n}{k} \in \{3, 4\}$ and

$\xi(GP(n,k))$		k				
		1	2	3	4	5
n	3	3				
	4	3				
	5	3	4			
	6	3	3			
	7	3	4	4		
	8	3	3	4		
	9	3	4	4	4	
	10	3	4	≥ 4		
	11	3				
	12	3		≥ 4	≥ 4	

Table 3.1: Containment Numbers of Generalized Petersen graphs, $\xi(GP(n,k))$

$k > 1$.

Furthermore, Proposition 3.0.2 tells us that if 3 cops suffice to contain the robber on one of these generalized Petersen graphs (*i.e.* if one of these graphs is a counterexample to Conjecture 2.0.1), then the robber must be contained on an inner vertex of the graph. Each inner vertex of $GP(n,k)$ with $\frac{n}{k} \in \{3,4\}$ will belong to a 3- or 4-cycle, and each vertex on the outer cycle of the graph will not.

We computed $\xi(G)$ for generalized Petersen graphs with n as large as 12. The results of our code can be seen in Table 3.1 (note that some cells are empty due to time constraints). For comparison, we also include tables from [3] that record $c(G)$ and $s(G)$ for many generalized Petersen graphs (see Tables 3.2 and 3.3). Since we are looking for a graph with $s(G) = 3$ and $\xi(G) = 4$, and $\frac{n}{k} \in \{3,4\}$ is required, the next smallest potential counterexample to Conjecture 2.0.1 is $GP(15,5)$.

$s(GP(n,k))$		k								
		1	2	3	4	5	6	7	8	9
n	3	3								
	4	3								
	5	3	3							
	6	3	3							
	7	3	4	4						
	8	3	3	3						
	9	3	4	3	4					
	10	3	4	4	4					
	11	3	4	4	4	4				
	12	3	4	3	4	4				
	13	3	4	4	4	4	4			
	14	3	4	4	4	4	4			
	15	3	4	4	4	4	4	4		
	16	3	4	4	4	4	4	4		
	17	3	4	4	4	4	4	4	4	
	18	3	4	4	4	4	4	4	4	
	19	3	4	4	4	4	4	4	4	4
	20	3	4	4	4	4	4	4	4	4

Table 3.2: Surrounding Cop Numbers of Generalized Petersen Graphs, $s(GP(n,k))$

$c(GP(n,k))$		k													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
n	3	2													
	4	2													
	5	2	3												
	6	2	2												
	7	2	3	3											
	8	2	2	3											
	9	2	3	2	3										
	10	2	3	3	3										
	11	2	3	3	3	3									
	12	2	3	2	3	3									
	13	2	3	3	3	3	3								
	14	2	3	3	3	3	3								
	15	2	3	3	3	3	3	3							
	16	2	3	3	3	3	3	3							
	17	2	3	3	3	3	3	3	3						
	18	2	3	3	3	3	3	3	3						
	19	2	3	3	3	3	3	3	3	3					
	20	2	3	3	3	3	3	3	3	3					
	21	2	3	3	3	3	3	3	3	3	3				
	22	2	3	3	3	3	3	3	3	3	3				
	23	2	3	3	3	3	3	3	3	3	3	3			
	24	2	3	3	3	3	3	3	3	3	3	3			
	25	2	3	3	3	3	3	4	3	3	3	3	3		
	26	2	3	3	3	3	3	3	3	3	4	3	3		
	27	2	3	3	3	3	4	3	3	3	3	3	3	3	
	28	2	3	3	3	3	4	3	4	3	3	3	3	3	
	29	2	3	3	3	3	3	3	4	3	3	4	4	3	3
	30	2	3	3	3	3	3	3	3	3	3	3	3	3	3

Table 3.3: Cop Numbers of Generalized Petersen Graphs, $c(GP(n,k))$

Chapter 4

A New Framework

4.0.1 A New Way to Analyze the Original and Surrounding Games

Given k cops controlled by player C, a single robber controlled by player R, and a graph G on which to play, we can model the original and surrounding versions of the game on the graph $\boxtimes^{k+1}G$ in such a way that we only need to pay attention to a single entity moving in this graph. For clarity, we will call vertices of $\boxtimes^{k+1}G$ "nodes" and vertices of G "vertices" as usual. Nodes of $\boxtimes^{k+1}G$ can be represented by $(k+1)$-tuples of vertices on G. Also, two nodes $(v_1, v_2, ..., v_{k+1})$ and $(u_1, u_2, ..., u_{k+1})$ of $\boxtimes^{k+1}G$ are adjacent if and only if v_i is adjacent or equal to u_i in G for all $i \in \{1, 2, ..., k+1\}$.

Let the first k entries in each $(k+1)$-tuple be the positions of the cops on G, and the last entry be the position of the robber on G. C begins the game by placing the k cops on vertices of G. Then, R places a "marker" over a node of $\boxtimes^{k+1}G$ where the first k entries of the $(k+1)$-tuple corresponding to this node coincide with the starting configuration chosen by C. The last entry in the $(k+1)$-tuple corresponds to the robber's starting vertex on G. The rest of the game is played on $\boxtimes^{k+1}G$. C and R take turns alternately moving the marker to adjacent nodes of $\boxtimes^{k+1}G$ (or they may choose to not move the marker on their turn). C may only move the marker from node $(v_1, v_2, ..., v_{k+1})$ to node $(u_1, u_2, ..., u_{k+1})$ if $v_{k+1} = u_{k+1}$, as this corresponds to each of the cops moving on G while the robber stays put. That is, the robber cannot move on C's turn. R may only move the marker from node $(v_1, v_2, ..., v_{k+1})$ to node $(u_1, u_2, ..., u_{k+1})$ if $v_i = u_i$ for all $i \in \{1, 2, ..., k\}$, as this corresponds to each cop staying put and the robber moving. That is, no cop may move on R's turn. This partitions the edge set of $\boxtimes^{k+1}G$ into three disjoint sets: X, the set of all edges the marker can cross on C's turn, Y, the set of all edges the marker can cross on R's turn, and Z, the

set of edges that the marker cannot cross on either player's turn.

Now, there are multiple nodes $(v_1, v_2, ..., v_{k+1})$ on $\boxtimes^{k+1}G$ where $v_{k+1} = v_i$ for at least one $i \in \{1, 2, ..., k\}$. Such nodes correspond to configurations in which the cops have caught the robber, let us call them "caught" nodes. Thus, when playing the original game, it is C's objective to use their half of the turns to move the marker onto any caught node, and it is R's objective to use their half of the turns to move the marker in such a way that it never lands on a caught node. If we increase the number of cops by some number $\ell \in \mathbb{N}$, we play on a new graph $\boxtimes^{k+\ell+1}G$ which has more caught nodes, and in which a greater proportion of edges of $\boxtimes^{k+\ell+1}G$ belong to X, thus making it easier for the cops to win.

Similarly, there are multiple nodes $(v_1, v_2, ..., v_{k+1})$ on $\boxtimes^{k+1}G$ where $N_G(v_{k+1}) \subseteq \{v_1, v_2, ..., v_k\}$. Such nodes correspond to configurations in which the cops have surrounded the robber, let us call them "surrounded" nodes. Note that if C uses their turn to move the marker onto a caught node, then R must move the marker to a non-caught node on their turn, as the robber is compelled to move when sharing a vertex in G with a cop. Thus, when playing the surrounding version of the game, it is C's objective to use their half of the turns to move the marker onto any surrounded node, and it is R's objective to use their half of the turns to move the marker in such a way that it never lands on a surrounded node. If we increase the number of cops by $\ell \in \mathbb{N}$, we play on a new graph $\boxtimes^{k+\ell+1}G$ which has more surrounded nodes, and in which a greater proportion of edges of $\boxtimes^{k+\ell+1}G$ belong to X, thus making it easier for the cops to win.

Theorem 4.0.1. *If we are playing with k cops and the marker is on node $(v_1, v_2, ..., v_{k+1})$, then C has $(\deg_G(v_1) + 1)(\deg_G(v_2) + 1) \cdots (\deg_G(v_k) + 1)$ moves from this node, and R has at most $(\deg_G(v_{k+1}) + 1)$ moves from this node.*

Proof. Each of the k cops can choose to either stay put or move to an adjacent vertex, so a cop on vertex v has $\deg_G(v) + 1$ options for how to move. The robber must stay put. So C has $(\deg_G(v_1) + 1)(\deg_G(v_2) + 1) \cdots (\deg_G(v_k) + 1)$ options.

When it is R's turn, each cop must stay put. If R occupies vertex u, then R may either stay put or move onto a neighbour of u that is not occupied by a cop. If none of u's neighbours are occupied by a cop then R has exactly $\deg_G(u) + 1$ legal moves. Otherwise R has strictly less than $\deg_G(u) + 1$ moves. Therefore when occupying vertex u, the robber has at most $\deg_G(u) + 1$ legal moves. □

Theorem 4.0.2. *If we are playing with k cops on a graph G with n vertices, then $\boxtimes^{k+1}G$ has $n(n^k - (n-1)^k)$ caught nodes.*

Proof. We must find the number of $(k+1)$-tuples $(v_1, v_2, ..., v_{k+1})$ that have $v_{k+1} = v_i$ for at least one $i \in \{1, 2, ..., k\}$. First, choose the vertex that the robber is going to be caught on, that is, fix a vertex $u = v_{k+1}$. There are n options. Now, the total number of $(k+1)$-tuples of the form $(v_1, v_2, ..., v_k, u)$ is n^k (because we have n options for each v_i). The total number of $(k+1)$-tuples $(v_1, v_2, ..., v_k, u)$ with $u \neq v_i$ for all $i \in \{1, 2, ..., k\}$ is $(n-1)^k$ (because we have $n-1$ options for each v_i). Thus, the number of $(k+1)$-tuples $(v_1, v_2, ..., v_k, u)$ with $v_i = u$ for at least one i is $n^k - (n-1)^k$. Since there are n different choices for u this means that the total number of caught nodes is $n(n^k - (n-1)^k)$. □

Theorem 4.0.3. *If we are playing with k cops on a graph G with n vertices, then $\boxtimes^{k+1}G$ has $\sum_{u \in V(G)} h(u)$ surrounded nodes, where*

$$h(u) = \begin{cases} \sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m} (-1)^m (n-m)^k & \text{if } \deg_G(u) \leq k \\ 0 & \text{if } \deg_G(u) > k \end{cases}.$$

Proof. First, we choose a vertex in G that the robber will be surrounded on, call it u. Now, we must count the number of distinct configurations of cops that surround u. Without loss of generality, let the vertices indexed from 1 to $\deg_G(u)$ be the neighbours of u. Let a_i represent the number of cops occupying vertex i in G for each $i \in \{1, 2, ..., n\}$. Now, consider the equation

$$a_1 + a_2 + \cdots + a_{\deg_G(u)} + \cdots + a_n = k$$

with $a_i \geq 1$ for each $i \in \{1, 2, ..., \deg_G(u)\}$ and $a_i \geq 0$ for each $i \in \{\deg_G(u) + 1, \deg_G(u) + 2, ..., n\}$. So a_i with $i \in \{1, 2, ..., \deg_G(u)\}$ represents the number of cops occupying the i^{th} neighbour of u. The corresponding exponential generating function is

$$\begin{aligned} g(x) &= \left(x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots\right)^{\deg_G(u)} \left(1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots\right)^{n - \deg_G(u)} \\ &= (e^x - 1)^{\deg_G(u)} (e^x)^{n - \deg_G(u)} \\ &= \left[\sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m} (e^x)^{\deg_G(u) - m} (-1)^m\right] e^{nx} e^{-x \deg_G(u)} \\ &= \sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m} e^{x \deg_G(u)} e^{-mx} (-1)^m e^{nx} e^{-x \deg_G(u)} \\ &= \sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m} e^{x(n-m)} (-1)^m \end{aligned}$$

$$= \sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m}(-1)^m \left(1 + (n-m)x + \frac{(n-m)^2x^2}{2!} + \cdots + \frac{(n-m)^kx^k}{k!} + \cdots\right)$$

We wish to find the coefficient on the $\frac{x^k}{k!}$ term in $g(x)$. When $g(x)$ is expanded, this term is

$$\binom{\deg_G(u)}{0}(-1)^0\left(\frac{(n-0)^kx^k}{k!}\right) + \binom{\deg_G(u)}{1}(-1)^1\left(\frac{(n-1)^kx^k}{k!}\right) + \cdots + \binom{\deg_G(u)}{\deg_G(u)}(-1)^{\deg_G(u)}\left(\frac{(n-\deg_G(u))^kx^k}{k!}\right)$$

$$= \left[\sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m}(-1)^m(n-m)^k\right]\frac{x^k}{k!}$$

The coefficient on $\frac{x^k}{k!}$ in $g(x)$ is equal to the number of distinct configurations of k cops where at least one cop occupies each neighbour of u (*i.e.* the number of surrounded nodes on $\boxtimes^{k+1}G$ corresponding to the vertex u, given that u can indeed be surrounded by k cops). Clearly the number of configurations that surround u is zero whenever $deg(u) > k$. Let us define $h(u)$ as the number of distinct configurations of k cops that surround a vertex $u \in V(G)$, so $h(u)$ is also equal to the number of surrounded nodes on $\boxtimes^{k+1}G$ corresponding to u. Then, clearly

$$h(u) = \begin{cases} \sum_{m=0}^{\deg_G(u)} \binom{\deg_G(u)}{m}(-1)^m(n-m)^k & \text{if } \deg_G(u) \leq k \\ 0 & \text{if } \deg_G(u) > k \end{cases}.$$

We must sum over all $u \in V(G)$ to get the number of surrounded nodes. Thus, the exact number of surrounded nodes is $\sum_{u \in V(G)} h(u)$. □

4.0.2 A New Way to Analyze the Containment Game

Given k cops controlled by player C, a single robber controlled by player R, and a graph G on which to play, we can model the containment version of the game on the graph $(\boxtimes^k L(G)) \boxtimes G$ in such a way that we only need to pay attention to a single entity moving in this graph. Once again, we will call vertices of $(\boxtimes^k L(G)) \boxtimes G$ "nodes" and vertices of G "vertices" as usual. Nodes of $(\boxtimes^k L(G)) \boxtimes G$ can be represented by $(k+1)$-tuples, where the first k entries are the edges of G (or vertices of $L(G)$) occupied by the cops, and the last entry is the vertex of G occupied by R. Also, two nodes $(e_1, e_2, ..., e_k, u)$ and $(e'_1, e'_2, ..., e'_k, v)$ of $(\boxtimes^k L(G)) \boxtimes G$ are adjacent if and only if edge e_i is adjacent or equal to edge e'_i in G for all $i \in \{1, 2, ..., k\}$, and vertex u is adjacent or equal to vertex v in G.

C begins the game by placing the k cops on edges of G. Then, R places a "marker" over a node on $(\boxtimes^k L(G)) \boxtimes G$ where the first k entries of the $k+1$-tuple corresponding to this node coincide with the starting configuration of edges chosen by C. The rest of the game is played on $(\boxtimes^k L(G)) \boxtimes G$. C and R take turns alternately moving the marker to adjacent nodes of $(\boxtimes^k L(G)) \boxtimes G$ (or they may choose to not move the marker on their turn). C may only move the marker from node $(e_1, e_2, ..., e_k, u)$ to node $(e'_1, e'_2, ..., e'_k, v)$ if $u = v$, as this corresponds to each of the cops moving on the edges of G while the robber stays put on their vertex. That is, the robber cannot move on C's turn. R may only move the marker from node $(e_1, e_2, ..., e_k, u)$ to node $(e'_1, e'_2, ..., e'_k, v)$ if $e_i = e'_i$ for all $i \in \{1, 2, ..., k\}$, as this corresponds to each cop staying put on their edges and the robber moving. That is, no cop may move on R's turn. This partitions the edge set of $(\boxtimes^k L(G)) \boxtimes G$ into three disjoint sets: X, the set of all edges the marker can cross on C's turn, Y, the set of all edges the marker can cross on R's turn, and Z, the set of edges that the marker cannot cross on either player's turn.

Now, there are multiple nodes $(e_1, e_2, ..., e_k, u)$ on $(\boxtimes^k L(G)) \boxtimes G$ where $\{v_1v_2 \in E(G) : v_1 = u \text{ or } v_2 = u\} \subseteq \{e_1, e_2, ..., e_k\}$. Such nodes correspond to configurations in which the cops have contained the robber, let us call them "contained" nodes. Thus, it is C's objective to use their half of the turns to move the marker onto any contained node, and it is R's objective to use their half of the turns to move the marker in such a way that it never lands on a contained node. If we increase the number of cops by $\ell \in \mathbb{N}$, we play on a new graph $(\boxtimes^{k+\ell} L(G)) \boxtimes G$ which has more contained nodes, and in which a greater proportion of edges of $(\boxtimes^{k+\ell} L(G)) \boxtimes G$ belong to X, thus making it easier for the cops to win.

Theorem 4.0.4. *If k cops occupy edges $e_1 = u_1v_1, e_2 = u_2v_2, ..., e_k = u_kv_k$ in G then C has $(\deg_G(u_1) + \deg_G(v_1) - 1)(\deg_G(u_2) + \deg_G(v_2) - 1) \cdots (\deg_G(u_k) + \deg_G(v_k) - 1)$ moves.*

Proof. A cop on edge $e_i = u_iv_i$ can move onto one of the $\deg_G(u_i) - 1$ *other* edges adjacent to u_i, move onto one of the $\deg_G(v_i) - 1$ *other* edges adjacent to v_i, or stay put on e_i. Thus each cop has $(\deg_G(u_i) - 1) + (\deg_G(v_i) - 1) + 1 = \deg_G(u_i) + \deg_G(v_i) - 1$ options. □

Theorem 4.0.5. *If we are playing with k cops on a graph G with m edges, then $(\boxtimes^k L(G)) \boxtimes G$ has $\sum_{u \in V(G)} h(u)$ contained nodes where*

$$h(u) = \begin{cases} \sum_{j=0}^{\deg_G(u)} \binom{\deg_G(u)}{j} (-1)^j (m-j)^k & \text{if } \deg_G(u) \leq k \\ 0 & \text{if } \deg_G(u) > k \end{cases}.$$

Also, if $\exists u \in V(G)$ such that $h(u) = 0$ then R wins the game.

Proof. First, we choose a vertex in G that the robber will be contained on, call it u. Now, we must count the number of distinct configurations of cops that contain u. Without loss of generality, let the edges incident to u be indexed from 1 to $\deg_G(u)$. Let a_i represent the number of cops occupying edge i in G for each $i \in \{1, 2, ..., m\}$. Now, consider the equation

$$a_1 + a_2 + \cdots + a_{\deg_G(u)} + \cdots + a_m = k$$

with $a_i \geq 1$ for each $i \in \{1, 2, ..., \deg_G(u)\}$ and $a_i \geq 0$ for each $i \in \{\deg_G(u) + 1, \deg_G(u) + 2, ..., m\}$. So, a_i with $i \in \{1, 2, ..., \deg_G(u)\}$ represents the number of cops occupying the i^{th} edge incident to u. The corresponding exponential generating function is

$$\begin{aligned} g(x) &= \left(x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots\right)^{\deg_G(u)} \left(1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots\right)^{m - \deg_G(u)} \\ &= \sum_{j=0}^{\deg_G(u)} \binom{\deg_G(u)}{j} (-1)^j \left(1 + (m-j)x + \frac{(m-j)^2 x^2}{2!} + \cdots + \frac{(m-j)^k x^k}{k!} + \cdots\right). \end{aligned}$$

We wish to find the coefficient on the $\frac{x^k}{k!}$ term in $g(x)$. When $g(x)$ is expanded, this term is

$$\left[\sum_{j=0}^{\deg_G(u)} \binom{\deg_G(u)}{j} (-1)^j (m-j)^k\right] \frac{x^k}{k!}$$

The coefficient on $\frac{x^k}{k!}$ in $g(x)$ is equal to the number of distinct configurations of k cops where at least one cop occupies each edge incident to u (*i.e.* the number of contained nodes on $(\boxtimes^k L(G)) \boxtimes G$ corresponding to the vertex u, given that u can indeed be contained by k cops). Clearly the number of configurations that contain u is zero whenever $\deg_G(u) > k$. Let us define $h(u)$ as the number of distinct configurations of k cops that contain a vertex $u \in V(G)$, so $h(u)$ is also equal to the number of contained nodes on $(\boxtimes^k L(G)) \boxtimes G$ corresponding to u. Then, clearly

$$h(u) = \begin{cases} \sum_{j=0}^{\deg_G(u)} \binom{\deg_G(u)}{j} (-1)^j (m-j)^k & \text{if } \deg_G(u) \leq k \\ 0 & \text{if } \deg_G(u) > k \end{cases}.$$

We must sum over all $u \in V(G)$ to get the number of contained nodes. Thus, the exact number of contained nodes is $\sum_{u \in V(G)} h(u)$. Also, if $h(u) = 0$ for some vertex u, then $\deg_G(u) > k$. In this case R can win by simply staying put on u. Thus, if there are not

exactly

$$\sum_{u \in V(G)} \left[\sum_{j=0}^{\deg_G(u)} \binom{\deg_G(u)}{j} (-1)^j (m-j)^k \right]$$

contained nodes on $(\boxtimes^k L(G)) \boxtimes G$ then $h(u) = 0$ for some vertex u, so R wins the game. $\square$

Chapter 5

Future Work

We have shown why Conjectures 2.0.1 and 2.0.2 cannot be proven true with straightforward methods. Thus, it is an open problem to either prove or disprove these conjectures.

In each version of the game, when R wins we can identify their winning strategy through the proofs of Theorem 2.0.2 and the analogous theorems from [1] and [3]. One important open question is as follows: when C wins the game, what is their strategy? Theorem 2.0.2 and the analogous theorems do not tell us much about this, as when C wins all the ψ values are empty. One possible approach could be to observe how the ψ values are altered as Algorithm 1 (and the corresponding algorithms from [1] and [3]) progresses. As an alternative approach, we may attempt to analyze C's strategy using the methods discussed in Chapters 4.2 and 4.3. Although the graphs used in these methods are more complicated than the graph G that the game is actually played on, these new methods significantly reduce the complexity of the moves made by C, as they now move a single entity across a more complicated graph instead of moving multiple entities across a simple graph. Thus, one use for these new methods could be to help analyze C's winning strategy.

As discussed in [3], the parameter $s(G)$ could be useful in proving or disproving Meyniel's conjecture. Similarly, the parameter $\xi(G)$ could be useful. Clearly if $s(G) \in O(\sqrt{n})$ or $\xi(G) \in O(\sqrt{n})$ then $c(G) \in O(\sqrt{n})$. Thus, we can assume $s(G), \xi(G) \notin O(\sqrt{n})$ when attempting to prove Meyniel's conjecture. A proof of Meyniel's conjecture could potentially be obtained in the following manner; Graphs with $s(G), \xi(G) \notin O(\sqrt{n})$ have relatively large values of $s(G)$ and $\xi(G)$, which corresponds to graphs that have a relatively high average vertex degree. Such graphs tend to have small values for $c(G)$, and a more in-depth look at this idea could lead to the conclusion that $c(G) \in O(\sqrt{n})$ for all graphs G with $s(G), \xi(G) \notin O(\sqrt{n})$, thus proving the conjecture.

Bibliography

[1] A. Bonato and E. Chiniforooshan. Pursuit and evasion from a distance: algorithms and bounds. Society for Industrial and Applied Mathematics 2009. Proceedings of the Workshop on Analytic Algorithmics and Combinatorics (ANALCO), pp.1-10.

[2] A. Bonato and R.J. Nowakowski. The game of cops and robbers on graphs. Student Mathematical Library, Vol. 61, American Mathematical Society, Providence, RI, 2011.

[3] A.C. Burgess, R.A. Cameron, N.E. Clarke, P. Danziger, S. Finbow, C.W. Jones, and D.A. Pike. Cops that surround a robber. *Discrete Appl. Math* 285 (2020) 552–566.

[4] D. Crytser, N. Komarov, and J. Mackey. Containment: a variation of cops and robber. *Graphs and Combinatorics* 36 (2020) 591–605.

[5] P. Frankl. Cops and robbers in graphs with large girth and Cayley graphs. *Discrete Appl. Math* 17 (1987) 301–305.

[6] R. Nowakowski and P. Winkler. Vertex-to-vertex pursuit in a graph. *Discrete Mathematics* 43 (1983) 235–239.

[7] A. Quilliot. Jeux et pointes fixes sur les graphes (Thèse de 3ème cycle). L'Université de Paris VI, 1978.

[8] R. Tošić. On cops and robber game. *Studia Sci. Math. Hungar.* 23 (1988) 225–229.

Appendix A

Source Code

Many subroutines from the code in [3] were used. We created new subroutines as necessary to make code that calculates $\xi(G)$.

A.0.1 Main Program File

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <limits.h>
#include "funcs.h"
#include "newfuncs.h"

int main(){

    /*finds xi(G) for Generalized Petersen graphs*/

    time_t start_time;
    start_time = time(NULL);

    int x;
    int n=12;
    int k=4;
    int order_G=2*n;
```

```
    char *G=build_GP_nk(n, k);

    printf("\ntesting containment on GP(%d,%d) \n", n, k);
    fflush(stdout);
    check_graph(G, order_G);

    x=find_containment_num_ver1(G, order_G);

    free(G);

    time_t elapsed_time;
    elapsed_time = time(NULL) - start_time;
    printf("\nelapsed time: %ld \n", elapsed_time);

    return 0;

}
```

A.0.2 Header File with Containment Subroutines

```
int find_size_EG(char *G, int order_G);
char *build_line_graph(char *G, int order_G);
int number_edge(char *G, int order_G, int x, int y);
void make_edge_matrix(char *G, int order_G, int *H);
int number_edge_ver2(int order_G, int x, int y, int *edge_matrix);
void get_coords(char *G, int order_G, int x, int coords[2]);
void store_endpoints(char *G, int order_G, int endpoints[]);
int find_endpoints(int num_edges, int x, int Left_or_Right, int endpoints[]);
int is_contained(char *G, int order_G, char *LG, int num_edges, int num_cops, int T_index, int vertex_w, int *edge_info);
int can_be_contained(char *G, int order_G, char *LG, int edges, int num_cops, int T_index, int vertex_w, int *edge_info);
char NEW_what_is(char *G, int order_G, int edges, int cops, int x, int y, long j, int *edge_info);
int NEW_alter_psi_value(char *G, int order_G, int edges, int L, char *psimatrix, long i, long j, char *nhoodj, int *edge_info);
void NEW_set_psi(char *psi_of_T, int order_G, int edges, int exp, long i, int j, char value);
char NEW_what_is_psi(char *psi_of_T, int order_G, int edges, int exp, long i, int j);
char do_cops_contain_ver1(char *G, int order_G, int k, char *LG, int *edge_info);
int find_containment_num_ver1(char *G, int order_G);



int find_size_EG(char *G, int order_G){

int i,j;
int num_edges=0;

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

       if(what_is(G, order_G, i, j))
         num_edges++;

    }
}

return num_edges;

}



char *build_line_graph(char *G, int order_G){

//given graph G, outputs the line graph L(G)
//could be sped up by using store_endpoints instead of get_coords

int i,j;
int order_LG=0;
char *LG; //array for L(G)

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

       if(what_is(G, order_G, i, j))
         order_LG++;

    }
} //determine the size of E(G), also the size of V(L(G))

//printf("\n number of edges in G is %d \n", order_LG);

if((LG =  (char*) malloc(((order_LG*order_LG-order_LG)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
          exit(1);
     }

for(i=0; i<((order_LG*order_LG)-order_LG)/2; i++)
    LG[i]=0;

int A[2];
int B[2];

for(i=1; i<=order_LG; i++){       //vertices of LG are edges of G
```

```
    for(j=i+1; j<=order_LG; j++){ //are edges i and j adjacent in G?

       get_coords(G, order_G, i, A); //A holds end vertices for edge i
       get_coords(G, order_G, j, B); //B holds end vertices for edge j

       //printf("\n edge %d has endpoints v_%d,v_%d ", i, A[0], A[1]);
       //printf("\n edge %d has endpoints v_%d,v_%d \n", j, B[0], B[1]);

       if(A[0]==B[0] || A[0]==B[1] || A[1]==B[0] || A[1]==B[1]){
         set_bit (LG, order_LG, i, j, 1); //if the edges share end vertices then they are adjacent
         //printf("\n edges %d and %d are adjacent\n", i, j);
       }
       else {//printf("\n edges %d and %d are not adjacent\n", i, j);
            }
    }
}

return LG;

}



int number_edge(char *G, int order_G, int x, int y){

//Gives a number to the edge at row x column y of the adj matrix for G. We number the edges
//starting at 1 and in increasing order from left to right, top to bottom in the adj matrix
//for G. If an edge is numbered k then it will be the kth vertex in L(G)

if(x>order_G || y>order_G || x<1 || y<1){
  printf("\n\n please pick valid inputs for x and y \n");
  exit(1);
}

if(what_is(G, order_G, x, y)==0){
  printf("\n\n there is no edge between v_%d and v_%d \n", x, y);
  exit(1);
}

if(x==y){
  printf("\n\n there is no edge between v_%d and itself \n", x);
  exit(1);
}

if(y<x)
  return number_edge(G, order_G, y, x);

int i,j;
int label=0;

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

      if(what_is(G, order_G, i, j))
        label++;

      if(x==i && y==j)
        return label;

    }
}

printf("\n\n the edge could not be found \n");
//exit(1);

return 0;

}



void make_edge_matrix(char *G, int order_G, int *H){

//given the adj matrix for G, makes a new matrix H where if the i-j entry in G
```

```
//is a 1 then the i-j entry in H is x where v_iv_j is the xth edge in G. So going
//left to right, top to bottom, the nonzero entries of G are 1,1,1,...,1 and the
//nonzero entries of H are 1,2,3,...,|E(G)|.
//H must be initialized to a matrix of all zeros of the same size as G

int i,j;
int count=0;

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

      if(what_is(G, order_G, i, j)){

        count++;
        H[(i-1)*order_G - (i*(i-1))/2 + (j-i) - 1]=count;

      }
   }
}

}



int number_edge_ver2(int order_G, int x, int y, int *edge_matrix){

//takes as input the matrix of ints described in make_edge_matrix
//returns the x-y entry i.e. the number of the edge v_xv_y

if(y<x)
  return number_edge_ver2(order_G, y, x, edge_matrix);

if(x>order_G || y>order_G || x<1 || y<1){
  printf("\n\n please pick valid inputs for x and y \n");
  exit(1);
}

if(edge_matrix[(x-1)*order_G - (x*(x-1))/2 + (y-x) - 1]==0){
  printf("\n\n there is no edge between v_%d and v_%d \n", x, y);
  exit(1);
}

if(x==y){
  printf("\n\n there is no edge between v_%d and itself \n", x);
  exit(1);
}

return edge_matrix[(x-1)*order_G - (x*(x-1))/2 + (y-x) - 1];

}



void get_coords(char *G, int order_G, int x, int coords[2]){

//given the number of an edge in G as per the rules outlined in number_edge,
//finds the row and column corresponding to this edge in the adj matrix of G
//and stores them in the array coords

int max = (order_G*(order_G - 1))/2;

if(x<1 || x>max){
  printf("\n\n please pick a valid input for x \n");
  exit(1);
}

coords[0]=0;
coords[1]=0;
int i,j;
int count=0;
int found_x=0;

for(i=1; i<=order_G; i++){
    for(j=i+1; j<=order_G; j++){
```

```
            if(what_is(G, order_G, i, j))
              count++;

            if(count==x){//go through adj matrix of G until we find our xth 1 i.e. our xth edge

              coords[0]=i;
              coords[1]=j;
              found_x=1;

              //printf("\n edge %d has endpoints v_%d and v_%d", x, coords[0], coords[1]);
              break;//this only breaks out of inner loop. we must break out of both loops!

            }

        }

        if(found_x)
          break;//breaks out of outer loop
    }

    if(found_x==0){
      printf("could not find edge %d, see line %d", x, __LINE__);
      exit(1);
    }

    }



    void store_endpoints(char *G, int order_G, int *endpoints){

    //endpoints MUST be of size 2 * |E(G)| when it is passed into this function

    int count=0;
    int i,j;

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

            if(what_is(G, order_G, i, j)){

              endpoints[count]=i;
              endpoints[count + 1]=j;
              count+=2;

            }

       }
    }

    }



    int find_endpoints(int num_edges, int x, int Left_or_Right, int *endpoints){

    /*returns one of the endpoints of edge x by looking at the 2 X |E(G)| matrix endpoints[]
      endpoints[] MUST be of size 2*|E(G)| and MUST already contain the values of the endpoint
      when it is passed into this function*/

    if(x<1 || x>num_edges || Left_or_Right<0 || Left_or_Right>1){
      printf("\n\n please pick valid inputs, see line %d\n", __LINE__);
      exit(1);
    }

    if(Left_or_Right==0) //find "left" endpoint
      return endpoints[2*(x-1)];

    if(Left_or_Right==1)//find "right" endpoint
      return endpoints[2*(x-1) + 1];

    printf("something went wrong, see line %d", __LINE__);
    return 0;
```

```
}


int is_contained(char *G, int order_G, char *LG, int num_edges, int num_cops, int T_index, int vertex_w, int *edge_info){

//returns 1 if vertex w is contained by cops at configuration T, 0 otherise

if(vertex_w<1 || vertex_w>order_G || T_index<1 || T_index>power(num_edges, num_cops)){
printf("error, see line %d", __LINE__);
exit(1);
}

int x,y;
int *T_cops;

if((T_cops = (int*) malloc(num_cops*sizeof(int)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

convert_index_ver2(num_edges, num_cops, T_index, T_cops);//remember T_cops contains the positions
                                                         //of the cops as vertices on L(G)!!!
int way_out=0;
int edge_number;

for(x=1; x<=order_G; x++){

   if(what_is(G, order_G, x, vertex_w)){ //find all vertices x that are adj to w

      edge_number = number_edge_ver2(order_G, x, vertex_w, edge_info); //find the number associated with edge xw
                                                  //this can be sped up by storing a matrix with this info!
      way_out=1;//start by assuming xw is not occupied by a cop

      for(y=0; y<num_cops; y++){ //traverse entries in T_cops (they are edges occupied by T)

         if(T_cops[y]==edge_number){
           way_out=0;
           break; //there is a cop from T on edge edge_number
         }

      }

      if(way_out){
        free(T_cops);
        return 0;//w is not contained by T
      }

   }

}

free(T_cops);
return 1;//w is contained by T

}


int can_be_contained(char *G, int order_G, char *LG, int edges, int num_cops, int T_index, int vertex_w, int *edge_info){

/*in can_w_be_surrounded from funcs.h we use the ford fulkerson algorithm... this is probably unnecessary.
  Here, we will just look at all adjacent configurations of cops and use is_contained*/

//remember we must first check if w is already contained by T_index! If so, then this step is unnecesary

//returns 1 if T_index can contain w in one move, 0 otherwise

long m = power(edges, num_cops);

if(vertex_w<1 || vertex_w>order_G || T_index<1 || T_index>m){
printf("error, see line %d", __LINE__);
exit(1);
```

```
}

long a;

for(a=1; a<=m; a++){//traverse vertices in LG(str prod)^num_cops i.e. all configurations of cops

   if(what_is_in_GpowL(LG, edges, num_cops, a, T_index)){//if T_a and T_index are adj configurations

     if(is_contained(G, order_G, LG, edges, num_cops, a, vertex_w, edge_info))//and w is contained by T_a
       return 1;//then T_index can contin w in one move

   }

}

return 0;

}



char NEW_what_is(char *G, int order_G, int edges, int cops, int x, int y, long j, int *edge_info){

/*Given a graph G, two vertices x and y in G, and a configuration T_j of cops (i.e.
  edges), determines if x and y are adjacent in G-T_j. Returns 1 if yes, 0 if no. */

if(j<1 || j>power(edges,cops)){
  printf("error, see line %d", __LINE__);
  exit(1);
}

if(what_is(G, order_G, x, y)==0)//what_is checks if 1<=x,y<=order_G
  return 0;

else{ /*x and y are adj in G... must determine if they are adj
        in G-T_j. we simply check if xy is an edge of T_j!*/

    int *T_j;
    int m;

    if((T_j = (int*) malloc(cops*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit(1);
    }

    int e = number_edge_ver2(order_G, x, y, edge_info);
    convert_index_ver2(edges, cops, j, T_j);

    for(m=0; m<cops; m++){

       if(e==T_j[m]){ //then xy is an edge in T_j
         free(T_j);
         return 0;
       }

    }

    free(T_j);
    return 1; //xy is an edge in G-T_j

}

printf("something went wrong, see line %d", __LINE__);
exit(1);

}//instead of using number_edge we could store this info (for every pair of vertices) in a matrix



int NEW_alter_psi_value(char *G, int order_G, int edges, int L, char *psimatrix, long i, long j, char *nhoodj, int *edge_info){

//NEW versions of functions are to be used in the containment version of the game.

//the original version of this function is in funcs.c
```

```
//this version intersects psi(T_i) with N_(G-T_j)[psi(T_j)]

    /*1<=i,j<=order_G^L is required. returns 0,1,2, or 3 depending on
      whether psi value was changed and/or an empty psi value was found.
      changes psi(T_i), depending on the neighborhood in G-T_j of psi(T_j)
      (see NEW algorithm). does NOT change psi(T_j). row containing entries
      of psi(T_A) is row A, 1<=A<=edges^L. also takes nbrhoodj as
      input so memory for it  is not constantly being created and
      freed up by this function.*/

    int i_empty=1;
    int j_empty=1;
    int found_empty=0;
    int x,y;

    if(order_G<1 || L<1 || L>order_G || i<1 || j<1 || i>power(edges,L) || j>power(edges,L)){
      printf("error, see line %d", __LINE__);
      exit(1);
    }

    for(x=0; x<order_G; x++)
       nhoodj[x]=0;

    for(x=1; x<=order_G; x++){

       if(NEW_what_is_psi(psimatrix, order_G, edges, L, j, x)){//i.e. if vertex x is in psi(T_j)

         j_empty=0;//psi(T_j) is nonempty

         nhoodj[x-1]=1;

         for(y=1; y<=order_G; y++){
                if(y!=x){ //this is required because NEW_what_is calls number_edge which exits if x==y
              if(NEW_what_is(G, order_G, edges, L, x, y, j, edge_info))
                nhoodj[y-1]=1;
           }
         }
       }
    }//the use of NEW_what_is ensures that nhoodj holds N_{G-T_j}[psi(T_j)]

    int changes_made=0;

    for(x=1; x<=order_G; x++){

       if(NEW_what_is_psi(psimatrix, order_G, edges, L, i, x)){//"if x is in psi(T_i)"

         i_empty=0;//psi(T_i) is nonempty

         if(nhoodj[x-1]==0){                                    //"and x is NOT in N_{G-T_j}[psi(T_j)]"
           NEW_set_psi(psimatrix, order_G, edges, L, i, x, 0); //"then remove x from psi(T_i)"
           changes_made=1;
         }
       }
    }

    if(i_empty || j_empty)
      found_empty=1;

    if(changes_made==0 && found_empty==0)
      return 0;

    if(changes_made==1 && found_empty==0)
      return 1;

    if(changes_made==1 && found_empty==1)
      return 2;

    if(changes_made==0 && found_empty==1)
      return 3;

    printf("something went wrong in NEW_alter_psi_value function");
    return 4;

}
```

```
void NEW_set_psi(char *psi_of_T, int order_G, int edges, int exp, long i, int j, char value){

     /*NEW versions of functions are to be used in the containment version of the game. sets new
       value to entry in row i, col j of (edges^exp)x(order_G) matrix holding psi(T) values*/

     if(i<1 || j<1 || i>power(edges,exp) || j>order_G || (value!=0 && value!=1)){
       printf("error, see line %d", __LINE__);
       exit(1);
     }                                        //access entry (row,col) of a 2D matrix
                                              //being represented by a 1D array by accessing
     psi_of_T[(i-1)*order_G + (j-1)] = value;//entry row*(#ofcols)+col in 1D array. note
                                              //that 0<=row<=#ofrows-1 and 0<=col<=#ofcols-1
}//row index, i, must be type long bc order_G^exp=(numberofrows) may be too large for an int



char NEW_what_is_psi(char *psi_of_T, int order_G, int edges, int exp, long i, int j){

   /*NEW versions of functions are to be used in the containment version of the game. returns
     value in row i column j of (edges^exp)x(order_G) matrix holding psi(T) values*/

     if(i<1 || j<1 || i>power(edges,exp) || j>order_G ){
       printf("error, see line %d", __LINE__);
       exit(1);
     }                                        //access entry (row,col) of a 2D matrix
                                              //being represented by a 1D array by accessing
     return psi_of_T[(i-1)*order_G + (j-1)];//entry row*(#ofcols)+col in 1D array. note
                                              //that 0<=row<=#ofrows-1 and 0<=col<=#ofcols-1
}//row index, i, must be type long bc order_G^exp=(numberofrows) may be too large for an int



char do_cops_contain_ver1(char *G, int order_G, int k, char * LG, int *edge_info){

/*returns 1 if k cops are enough to contain the robber, 0 otherwise*/

printf("Started testing with %d cops\n", k);
fflush(stdout);

long a,b;//used for loops going up to order_LGpowk or higher
int c,d;//used for loops going up to order_G or k

int edges = find_size_EG(G, order_G);
long order_LGpowk=power(edges, k);

if(order_LGpowk > LONG_MAX/order_G){
  printf("values have become too large, see line %d\n", __LINE__);
  exit(1);
}//makes sure that order_LGpowk*order_G > LONG_MAX is false

char *psiofT;

if((psiofT = (char*) malloc(order_LGpowk*order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

//initialize all psi(T) to V(G)

printf("Initializing psi(T) to V(G) for all T\n");
fflush(stdout);

for(a=0; a<order_LGpowk*order_G; a++)
   psiofT[a]=1;

//remove A_T and B_T

printf("Removing A_T and B_T from psi(T) for all T\n");
fflush(stdout);

int p,q;
```

```
for(a=1; a<=order_LGpowk; a++){ //traverse rows of psiofT[] i.e. configurations of cops
   for(c=1; c<=order_G; c++){   //traverse the entries of these rows, left to right i.e. vertices of G

      p=is_contained(G, order_G, LG, edges, k, a, c, edge_info);
                                          //check if vertex c is contained by congiguration T_a
      if(p==1)                                    //vertex c is contained by T_a, dont need to
        NEW_set_psi(psiofT, order_G, edges, k, a, c, 0);//check if it can be contained in one move

      if(p==0){

        q=can_be_contained(G, order_G, LG, edges, k, a, c, edge_info);

        if(q==1)                                  //vertex c can be contined by T_a in one move
          NEW_set_psi(psiofT, order_G, edges, k, a, c, 0);
      }

   }
}

//intersection step

printf("Starting intersection step\n");
fflush(stdout);

long num_changes=1;
int result1, result2;
char *NEW_nhood;//used in intersection step, holds N_(G-T')[psi(T')]

if((NEW_nhood = (char*) malloc(order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

for(c=0; c<order_G; c++)
   NEW_nhood[c]=0;

//long count = 0;

while(num_changes>0){

          //count++;
          //printf("starting loop %ld \n", count);
          //fflush(stdout);

          num_changes=0;

          for(a=1; a<=order_LGpowk; a++){//a represents T, b represents T' (see NEW algorithm).
             for(b=a+1; b<=order_LGpowk; b++){
          //for and if statements ensure we traverse all edges of (str prod)^k(LG).
                if(what_is_in_GpowL(LG, edges, k, a, b)){ //then (T_a,T_b) is an edge in (str prod)^k(LG)

                  result1 = NEW_alter_psi_value(G, order_G, edges, k, psiofT, a, b, NEW_nhood, edge_info);
                                                                  //alter psi(T_a)
                  num_changes += result1;

                  result2 = NEW_alter_psi_value(G, order_G, edges, k, psiofT, a, b, NEW_nhood, edge_info);
                                                                  //alter psi(T_b)
                  num_changes += result2;

                  if(result1==2 || result2==2 || result1==3 || result2==3){
                    printf("found empty psi value when testing with %d cops\n", k);
                    fflush(stdout);
                    free(psiofT);
                    return 1;
                  }
                }
             }//while loop is exited if we go through all the edges of
          }   //(str prod)^k(LG) without changing any psi values, or if
}             //NEW_alter_psi_value ever returns 2 or 3 (meaning psi(T_a)
              //or psi(T_b) is empty, see NEW_alter_psi_value() function).
free(NEW_nhood);

printf("Done intersection step\n");
```

```
fflush(stdout);

//check if psi(T_1) is empty

char cops_win=1;

for(c=0; c<order_G; c++){

    if(psiofT[c]==1){
       cops_win=0;
       printf("psi(T_1) is nonempty, therefore %d cops cannot contain the robber\n", k);
       fflush(stdout);
       break;
    }
}//only need to check if psi(T_1) is empty (write new corollary)

/*   // print psi(T) values

     printf("psi(T) values are as follows: \n\n");
     fflush(stdout);

     for(a=1; a<=order_LGpowk; a++){
        for(c=1; c<=order_G; c++){
           printf("%d", NEW_what_is_psi(psiofT, order_G, k, a, c));
        }
        printf("\n");
     }
     printf("\n");
     fflush(stdout);
*/

free(psiofT);
return cops_win;

}



int find_containment_num_ver1(char *G, int order_G){

    //printf("\nrunning find_containment_num_ver1\n");
    //fflush(stdout);

    int hold=0;
    int deg,i,j,c;

    for(i=1; i<=order_G; i++){

       deg=get_vertex_degree (G, order_G, i);

       if(deg>hold)
         hold=deg;

    }

    printf("\nmaximum degree of all vertices on G is %d\n", hold);
    fflush(stdout);

    char *LG = build_line_graph(G, order_G);

    int *edge_info; //used in number_edge_ver2

    if((edge_info =  (int*) malloc(((order_G*order_G-order_G)/2)*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
      exit(1);
    }

    for(c=0; c<(order_G*order_G-order_G)/2; c++)
       edge_info[c]=0;

    make_edge_matrix(G, order_G, edge_info);
                                  //using number_edge_ver2 and edge_info is faster than using number_edge

    //start j at "(hold)" since xi(G) cannot be less than the max degree of vertices in G
```

```
    for(j=hold; j<=order_G; j++){
       if(do_cops_contain_ver1(G, order_G, j, LG, edge_info)){
         printf("containment number of G is %d\n", j);
         fflush(stdout);
         free(LG);
         free(edge_info);
         return j;
       }
       printf("xi(G) is not %d\n", j);
       fflush(stdout);
    }

    printf("error - no containment number was found. see line %d", __LINE__);
    exit(1);
    return 0;

}
```

A.0.3 Header File from [3]

```
   typedef struct path_node
   {
   int vertex;
   struct path_node *next;
   } path_node;

long power(int base, int exp);
void print_graph(char *graph, int order_graph);
void check_graph(char *G, int order_G);
void convert_index(int order_G, int order_H, int index, int newindex[2]);
void convert_index_ver2(int order_G, int exp, long index, int *newindex);
void compute_str_prod(char *G, int order_G, char *H, int order_H, char *GXH);
char *build_str_prod(char *G, int order_G, char *H, int order_H);
char *str_prod_pow(char *G, int order_G, int exp);
char do_cops_win_ver2B(char *G, int order_G, int L);
char do_cops_win_ver2C(char *G, int order_G, int L);
int find_cop_number_ver2B(char *G, int order_G);
int find_cop_number_ver2C(char *G, int order_G);
char do_cops_surround_ver2B(char *G, int order_G, int L);
char do_cops_surround_ver2C(char *G, int order_G, int L);
int find_surround_num_ver2B(char *G, int order_G);
int find_surround_num_ver2C(char *G, int order_G);
void compute_something(char *G, int order_G, int choice);
int alter_psi_value(char *G, int order_G, int L, char *psimatrix, long i, long j, char *nhoodj);
int can_w_be_surrounded(char *G, int order_G, int exp, int T, int vertex_w);
void find_nhood(char *G, int order_G, int v, int deg_v, int *nhood);
char what_is_in_GpowL(char *G, int order_G, int exp, long i, long j);
char what_is_psi(char *psi_of_T, int order_G, int exp, long i, int j);
void set_psi(char *psi_of_T, int order_G, int exp, long i, int j, char value);
char *build_K_mn_graph(int m, int n);
char *build_Kmn_str_prod_Kmn(int m, int n);
char *build_GP_nk(int n, int k);
char *build_Pm_cartprod_Pn(int m, int n);
char *build_Pm_strprod_Pn(int m, int n);

char what_is (char *A, int n, int i, int j);
void set_bit (char *A, int n, int i, int j, char value);
int get_vertex_degree (char *A, int n, int vertex);
int what_is_D (int *D, int n, int i, int j);
void set_bit_D (int *D, int n, int i, int j, int value);
char find_xy_path (int *D, int n, int x, int y, path_node **Path);
void enqueue (path_node **Queue, int value);
int Ford_Fulk(int *A, int n, int x, int y);


long power(int base, int exp){

     if(exp<0){
       printf("error, see line %d \n", __LINE__);
       exit(1);
     }
```

```
    if(base==0)
      return 0;

    if(exp==0)
      return 1;

    int i;
    long temp=base;

    for(i=1; i<exp; i++){

       if(temp > LONG_MAX/base || temp<=0){//then temp*base > LONG_MAX
         printf("values have become too large, see line %d\n", __LINE__);
         exit(1);
       }//checks to make sure temp*base never exceeds the max value for long data type

       temp*=base;

    }

    return temp;

}


void print_graph(char *graph, int order_graph){

    for(int i=1; i<=order_graph; i++){
       for(int j=1; j<=order_graph; j++){
          printf("%d", what_is(graph, order_graph, i, j));
          //fflush(stdout);
       }
       printf("\n");
       fflush(stdout);
    }
}


void check_graph(char *G, int order_G){

    printf("\n  |V(G)|=%d \n", order_G);
    fflush(stdout);

    int num_edges=0;
    int i,j,deg;

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

          if(what_is(G, order_G, i, j))
            num_edges++;
       }
    }

    printf("  |E(G)|=%d \n", num_edges);
```

```
    fflush(stdout);

    int hold_1=order_G;
    int hold_2=1;

    for(i=1; i<=order_G; i++){

       deg=get_vertex_degree (G, order_G, i);

       if(deg<hold_1)
         hold_1=deg;

       if(deg>hold_2)
         hold_2=deg;
    }

    printf("  minimum degree of all vertices on G is %d \n", hold_1);
    printf("  maximum degree of all vertices on G is %d \n\n", hold_2);
    fflush(stdout);

}


void convert_index(int order_G, int order_H, int index, int newindex[2]){

    if(order_G<1 || order_H<1 || index<1 || index>order_G*order_H){
      printf("error, see line %d \n", __LINE__);
      exit(1);
    }//NOTE: ensure we are indexing G(str prod)H, NOT the other way around

    newindex[0]=((index-1)/order_H)+1;//set 1st value of ordered pair (will be truncated)
    newindex[1]=index-((newindex[0]-1)*order_H);//set 2nd value of ordered pair

}


void convert_index_ver2(int order_G, int exp, long index, int *newindex){

    /*converts indices from 1 to (order_G^exp) into indices that are exp-tuples.
      these exp-tuples represent the positions of the cops in configuration T_index.
      newindex is a pointer to an array of integers that MUST be initialized to be
      of order "(exp)" BEFORE it is passed into this function*/

    if(order_G<1 || exp<1 || index<1 || index>power(order_G,exp)){
      printf("error. see line %d \n", __LINE__);
      exit(1);
    }

    long temp=index-1;
    int y;

    for(y=0; y<exp; y++){
       newindex[exp-1-y]=(temp%order_G)+1; //values in newindex can
       temp /= order_G;                    //range from 1 to order_G
    }                                      //bc they are vertices in G
```

```
}


void compute_str_prod(char *G, int order_G, char *H, int order_H, char *GXH){

     /*G, H, and GXH are adj matrices for the graphs G, H, and G(str prod)H respectively. GXH
     MUST be defined beforehand as a (order_G*order_H) x (order_G*order_H) matrix OF ALL ZEROS*/

     int order_GXH=order_G*order_H;
     int i,j;
     int R[2];
     int C[2];

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

           convert_index(order_G, order_H, i, R);//map i to ordered pair R[]. indexes row of GXH
           convert_index(order_G, order_H, j, C);//map j to ordered pair C[]. indexes col of GXH

           if(R[0]==C[0])          //entry is in main diagonal of "blocks" in adj. matrix of GXH
             set_bit(GXH, order_GXH, i, j, what_is(H, order_H, R[1], C[1]));

           else{  if(what_is(G, order_G, R[0], C[0])==1){//"block" in GXH corresponds to 1 in G

                    if(R[1]==C[1])                      //this section of code is why we must
                      set_bit(GXH, order_GXH, i, j, 1);//ensure GXH is initialized to all 0's.
                                                        //If an entry in GXH is supposed
                    else                                //to be set to 0, we do nothing.
                      set_bit(GXH, order_GXH, i, j,  what_is(H, order_H, R[1], C[1]));
                  }
                }//if "block" in GXH corresponds to a 0 in G, then all entries
        }        //in the block must be set to 0... if this is the case, we do
     }           //nothing bc GXH has been initialized to all 0's.
}


char *build_str_prod(char *G, int order_G, char *H, int order_H){

     /*G and H are adj matrices for the graphs G and H respectively.
       function returns the adj matrix of G(str prod)H*/

     int i,j;
     int order_GXH=order_G*order_H;
     char *GXH;

     if((GXH =  (char*) malloc(((order_GXH*order_GXH-order_GXH)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
          exit(1);
     }

     for(i=0; i<((order_GXH*order_GXH)-order_GXH)/2; i++)
        GXH[i]=0;

     int R[2];
     int C[2];
```

```
    for(i=1; i<=order_GXH; i++){
       for(j=i+1; j<=order_GXH; j++){

          convert_index(order_G, order_H, i, R);//map i to ordered pair R[]. indexes row of GXH
          convert_index(order_G, order_H, j, C);//map j to ordered pair C[]. indexes col of GXH

          if(R[0]==C[0])          //entry is in main diagonal of "blocks" in adj. matrix of GXH
            set_bit(GXH, order_GXH, i, j, what_is(H, order_H, R[1], C[1]));

          else{  if(what_is(G, order_G, R[0], C[0])){//"block" in GXH corresponds to 1 in G

                   if(R[1]==C[1])                    //this section of code is why we must
                     set_bit(GXH, order_GXH, i, j, 1);//ensure GXH is initialized to all 0's.
                                                      //If an entry in GXH is supposed
                   else                               //to be set to 0, we do nothing.
                     set_bit(GXH, order_GXH, i, j,  what_is(H, order_H, R[1], C[1]));
                 }
               }//if "block" in GXH corresponds to a 0 in G, then all entries
       }        //in the block must be set to 0... if this is the case, we do
    }           //nothing bc GXH has been initialized to all 0's.

    return GXH;

}


char *str_prod_pow(char *G, int order_G, int exp){

    /*returns pointer to adj matrix of (str prod)^(exp)(G). note that
      (str prod)^(exp)(G) involves "(exp)" G's in total, NOT (exp)+1.*/

    if(exp<1){
      printf("error, see line %d \n", __LINE__);
      exit(1);
    }

    char *temp1;
    char *temp2;
    int order_t1=order_G;
    int order_t2;
    int i,j;

    if((temp1 = (char*) malloc(((order_t1*order_t1-order_t1)/2)*sizeof(char)))==NULL){
         printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
         exit(1);
    }

    for(i=0; i<((order_t1*order_t1)-order_t1)/2; i++)
       temp1[i]=G[i];

    for(i=1; i<exp; i++){ //want to perform G(str prod)G "(exp - 1)" many times

       order_t2=order_G*order_t1;
```

```
        if((temp2 = (char*) malloc(((order_t2*order_t2-order_t2)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
          printf("memory failed when i=%d \n", i);
          exit(1);
        }

        for(j=0; j<((order_t2*order_t2)-order_t2)/2; j++)
           temp2[j]=0; //this is why temp1 and temp2 cannot point to the
                       //same thing...temp1 would become an array of 0's
        compute_str_prod(G, order_G, temp1, order_t1, temp2);

        free(temp1); //this is why we initialized temp1 as a COPY
                     //of G... otherwise this line would delete G
        order_t1=order_t2;

        if((temp1 = (char*) malloc(((order_t1*order_t1-order_t1)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
          printf("memory failed when i=%d \n", i);
          exit(1);
        }

        for(j=0; j<((order_t1*order_t1)-order_t1)/2; j++)
           temp1[j]=temp2[j]; //we do this instead of "temp1=temp2;" because
                              //temp1 & temp2 cannot point to the same thing
        free(temp2);

      }

      if(order_t1!=power(order_G,exp)){
        printf("error, see line %d \n", __LINE__);
        exit(1);
      }

      return temp1;

}


char do_cops_win_ver2B(char *G, int order_G, int L){

      /*returns 1 if L cops is enough to catch robber, 0 otherwise.
        Version 2B does NOT store adj matrix of (str prod)^L(G),
        and returns "cops win" immediately if the program ever
        encounters an empty psi value.*/

      //printf("\nstarted testing with %d cops\n", L);
      //fflush(stdout);

      long a,b;//used for loops going up to order_GpowL or higher
      int c,d;//used for loops going up to order_G or L
      long order_GpowL=power(order_G, L);//power() subroutine will check if this
                                         //value is too large for the long data type
      if(order_GpowL > LONG_MAX/order_G){
        printf("values have become too large, see line %d\n", __LINE__);
        exit(1);
```

```
}//makes sure that order_GpowL*order_G > LONG_MAX is false

char *psiofT;

if((psiofT = (char*) malloc(order_GpowL*order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

/////////////// initialize all psi(T) to V(G) ///////////////

for(a=0; a<order_GpowL*order_G; a++)
   psiofT[a]=1;

//printf("done initializing psi(T) to V(G) for all T\n");
//fflush(stdout);

/////////////// remove N_G[T] from all psi(T) ///////////////

int *cop_places; //pointer to array of ints that will serve as L-tuple form of
                 //index of a T-value. This rep's the layout of cops at that T-value
if((cop_places = (int*) malloc(L*sizeof(int)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT[]

   convert_index_ver2(order_G, L, a, cop_places);

   for(c=0; c<L; c++){/*entries of cop_places are #'s corresponding to vertices in G that
                       are occupied by cops. must remove N_G[cop_places[c]] from psi(T)*/
      for(d=1; d<=order_G; d++){ //traverse columns of G.
         //in below if statement, set row # to be cop_places[c]
         if(what_is(G, order_G, cop_places[c], d) || cop_places[c]==d)
           set_psi(psiofT, order_G, L, a, d, 0);  //"if vertex k is in N_G[cop_places[j]]"
      }
   }
}

//printf("done removing N_G(T) from psi(T) for all T\n");
//printf("starting intersection step\n");
//fflush(stdout);

////////////////////////// do intersection step //////////////////////////

long num_changes=1;
int result1, result2;
char *nhoodGT2;//used in intersection step, holds N_G[psi(T_2)]

if((nhoodGT2 = (char*) malloc(order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}
```

```
    for(c=0; c<order_G; c++)
       nhoodGT2[c]=0;

    //long count = 0;

    while(num_changes>0){

         //count++;
         //printf("starting loop %ld \n", count);
         //fflush(stdout);

         num_changes=0;

         for(a=1; a<=order_GpowL; a++){//a represents T, b represents T' (see algorithm).
            for(b=a+1; b<=order_GpowL; b++){
         //for and if statements ensure we traverse all elements of E[(str prod)^L (G)].
                 if(what_is_in_GpowL(G, order_G, L, a, b)){ //then (T_a,T_b) is an edge

                   result1 = alter_psi_value(G, order_G, L, psiofT, a, b, nhoodGT2);
                                                              //alter psi(T_a)
                   num_changes += result1;

                   result2 = alter_psi_value(G, order_G, L, psiofT, b, a, nhoodGT2);
                                                              //alter psi(T_b)
                   num_changes += result2;

                   if(result1==2 || result2==2 || result1==3 || result2==3){
                     printf("found empty psi value when testing with %d cops\n", L);
                     fflush(stdout);
                     free(psiofT);
                     return 1;
                   }
                 }
            }//while loop is exited if we go through all the edges of
         }   //(str prod)^L(G) without changing any psi values, or if
    }        //alter_psi_value ever returns 2 or 3 (meaning psi(T_a)
             //or psi(T_b) is empty, see alter_psi_value() function).
    free(nhoodGT2);

    //printf("done intersection step\n");
    //fflush(stdout);

    /////////////// determine whether or not L cops win ///////////////

    char cops_win=1;

    for(c=0; c<order_G; c++){

       if(psiofT[c]==1){
         cops_win=0;
         break;
       }
    }//by corollary, only need to check if psi(T_1) is empty

/*
```

```
    ////////////////////// print psi(T) values //////////////////////

    for(a=1; a<=order_GpowL; a++){
       for(c=1; c<=order_G; c++){
          printf("%d", what_is_psi(psiofT, order_G, L, a, c));
       }
       printf("\n");
    }
    printf("\n");
    fflush(stdout);
*/

    free(cop_places);
    free(psiofT);
    return cops_win;

}


char do_cops_win_ver2C(char *G, int order_G, int L){

    /*returns 1 if L cops is enough to catch robber, 0 otherwise.
      Version 2C does NOT store adj matrix of (str prod)^L(G).
      Ver2C skips the intersection step for T-values that we
      know will not have their psi(T) values altered, and
      returns "cops win" immediately if it ever encounters
      an empty psi value.*/

    //printf("\nstarted testing with %d cops\n", L);
    //fflush(stdout);

    long a,b;//used for loops going up to order_GpowL or higher
    int c,d;//used for loops going up to order_G or L
    long order_GpowL=power(order_G, L);//power() subroutine will check if this
                                       //value is too large for the long data type
    if(order_GpowL > LONG_MAX/order_G){
      printf("values have become too large, see line %d\n", __LINE__);
      exit(1);
    }//makes sure that order_GpowL*order_G > LONG_MAX is false

    char *psiofT;

    if((psiofT = (char*) malloc(order_GpowL*order_G*sizeof(char)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit(1);
    }

    /////////////// initialize all psi(T) to V(G) ///////////////

    for(a=0; a<order_GpowL*order_G; a++)
       psiofT[a]=1;

    //printf("done initializing psi(T) to V(G) for all T\n");
    //fflush(stdout);
```

```
/////////////// remove N_G[T] from all psi(T) ///////////////

int *cop_places; //pointer to array of ints that will serve as L-tuple form of
                 //index of a T-value. This rep's the layout of cops at that T-value
if((cop_places = (int*) malloc(L*sizeof(int)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT[]

   convert_index_ver2(order_G, L, a, cop_places);

   for(c=0; c<L; c++){/*entries of cop_places are #'s corresponding to vertices in G that
                        are occupied by cops. must remove N_G[cop_places[c]] from psi(T)*/
      for(d=1; d<=order_G; d++){ //traverse columns of G.
         //in below if statement, set row # to be cop_places[c]
         if(what_is(G, order_G, cop_places[c], d) || cop_places[c]==d)
           set_psi(psiofT, order_G, L, a, d, 0);  //"if vertex k is in N_G[cop_places[j]]"
      }
   }
}

//printf("done removing N_G(T) from psi(T) for all T\n");
//printf("starting intersection step\n");
//fflush(stdout);

////////////////////////// do intersection step //////////////////////////

char sthn_has_changed=1;
char *prev_changed;//one element of this array for every T... this is a 1-D array

if((prev_changed =  (char*) malloc((order_GpowL)*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
  exit(1);
}

char *curr_changes;//one element of this array for every T... this is a 1-D array

if((curr_changes =  (char*) malloc((order_GpowL)*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
  exit(1);
}

for(a=0; a<order_GpowL; a++)
   prev_changed[a]=1;

char *nhoodGT2;//used for intersection step, holds N_G[psi(T_2)]

if((nhoodGT2 = (char*) malloc(order_G*sizeof(char)))==NULL){
 printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
 exit(1);
}

for(c=0; c<order_G; c++)
```

```
   nhoodGT2[c]=0;

int found_empty_psi=0;
int value_1, value_2;

//long count = 0;

while(sthn_has_changed){

     //count++;
     //printf("starting loop %ld \n", count);
     //fflush(stdout);

     sthn_has_changed=0;

     for(a=0; a<order_GpowL; a++)
        curr_changes[a]=0;

     for(a=1; a<=order_GpowL; a++){//a represents T, b represents T' (see algorithm).

        if(prev_changed[a] || curr_changes[a]){

          for(b=1; b<=order_GpowL; b++){//in this version b must start at 1, NOT a+1

             if(what_is_in_GpowL(G, order_G, L, a, b)){ //then (T_a,T_b) is an edge

               value_1 = alter_psi_value(G, order_G, L, psiofT, a, b, nhoodGT2);

               if(value_1==1 || value_1==2){
                 sthn_has_changed=1;
                 curr_changes[a]=1;//psi(T_a) was just altered
               }

               if((value_1==2 || value_1==3) && found_empty_psi==0){
                 found_empty_psi=1;
                 printf("found first empty psi value when testing with %d cops\n", L);
                 fflush(stdout);
                 free(psiofT);
                 return 1;
               }

               value_2 = alter_psi_value(G, order_G, L, psiofT, b, a, nhoodGT2);

               if(value_2==1 || value_2==2){
                 sthn_has_changed=1;
                 curr_changes[b]=1;//psi(T_b) was just altered
               }

               if((value_2==2 || value_2==3) && found_empty_psi==0){
                 found_empty_psi=1;
                 printf("found first empty psi value when testing with %d cops\n", L);
                 fflush(stdout);
                 free(psiofT);
                 return 1;
               }
```

```
                }
              }
            }
          }

          for(a=0; a<order_GpowL; a++)
             prev_changed[a]=curr_changes[a];
      }

      free(nhoodGT2);

      //printf("done intersection step\n");
      //fflush(stdout);

      /////////////// determine whether or not L cops win ///////////////

      char cops_win=1;

      for(c=0; c<order_G; c++){

         if(psiofT[c]==1){
           cops_win=0;
           break;
         }
      }//by corollary, only need to check if psi(T_1) is empty

/*
      ////////////////////// print psi(T) values //////////////////////

      for(a=1; a<=order_GpowL; a++){
         for(c=1; c<=order_G; c++){
            printf("%d", what_is_psi(psiofT, order_G, L, a, c));
         }
         printf("\n");
      }
      printf("\n");
      fflush(stdout);
*/

      free(cop_places);
      free(psiofT);
      return cops_win;

}


int find_cop_number_ver2B(char *G, int order_G){

    //printf("\nrunning find_cop_number_ver2B\n");
    //fflush(stdout);

    int i;

    for(i=1; i<=order_G; i++){
       if(do_cops_win_ver2B(G,order_G,i)){
```

```
        return i;
      }
      printf("c(G) is not %d\n", i);
      fflush(stdout);
    }

    printf("error - no cop number was found. see line %d", __LINE__);
    exit(1);
    return 0;

}


int find_cop_number_ver2C(char *G, int order_G){

    //printf("\nrunning find_cop_number_ver2C\n");
    //fflush(stdout);

    int i;

    for(i=1; i<=order_G; i++){
      if(do_cops_win_ver2C(G,order_G,i)){
        return i;
      }
      printf("c(G) is not %d\n", i);
      fflush(stdout);
    }

    printf("error - no cop number was found. see line %d", __LINE__);
    exit(1);
    return 0;

}


char do_cops_surround_ver2B(char *G, int order_G, int L){

     /*returns 1 if L cops is enough to surround robber, 0 otherwise.
       Version 2B does NOT store adj matrix of (str prod)^L(G), and
       returns "cops win" immediately if it ever encounters an
       empty psi value.*/

     //printf("\nstarted testing with %d cops\n", L);
     //fflush(stdout);

     long a,b;//used for loops going up to order_GpowL or higher
     int c,d,e;//used for loops going up to order_G or L
     long order_GpowL=power(order_G, L);//power() subroutine will check if this
                                        //value is too large for the long data type
     if(order_GpowL > LONG_MAX/order_G){
       printf("values have become too large, see line %d\n", __LINE__);
       exit(1);
     }//makes sure that order_GpowL*order_G > LONG_MAX is false

     char *psiofT;
```

```
if((psiofT = (char*) malloc(order_GpowL*order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

/////////////// initialize all psi(T) to V(G) ///////////////

for(a=0; a<order_GpowL*order_G; a++)
   psiofT[a]=1;

//printf("done initializing psi(T) to V(G) for all T\n");
//fflush(stdout);

////////// remove vertices in T, vertices currently surrounded by T, and  //////////
////////// vertices that can be surrounded by T in 1 move from all psi(T) //////////

int *cop_places; //pointer to array of ints that will serve as L-tuple form of
                 //index of a T-value. This rep's the layout of cops at that T-value
if((cop_places = (int*) malloc(L*sizeof(int)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

for(c=0; c<L; c++)
   cop_places[c]=0;

int is_surrounded, occupied;

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT

  convert_index_ver2(order_G, L, a, cop_places);//find positions of cops in T_a

  for(c=0; c<L; c++)                                    //remove vertices in
     set_psi(psiofT, order_G, L, a, cop_places[c], 0);//T_a from psi(T_a)

  //remove vertices in G that are surrounded by T_a from psi(T_a)

  for(c=1; c<=order_G; c++){ //traverse vertices of G

     if(what_is_psi(psiofT, order_G, L, a, c)){//only need to remove entry in
                                               //column c if it is currently a 1
       is_surrounded=1; //start by assuming vertex c in G is surrounded by T_a

       for(d=1; d<=order_G; d++){

          if(what_is(G, order_G, c, d)){ //traverse vertices of N_G(c)

            occupied=0; //assume vertex d is not in T

            for(e=0; e<L; e++){//check if vertex d is in T

               if(d==cop_places[e]){
                 occupied=1;
                 break;
```

```
            }
          }

          if(occupied==0){    //vertex c is not surrounded if there
            is_surrounded=0;  //is a vertex in N_G(c) but not in T_a
            break;
          }
        }
      }

      if(is_surrounded)
        set_psi(psiofT, order_G, L, a, c, 0);//c is surrounded, remove c from psi(T_a)
    }
  }
}

free(cop_places);

//printf("done removing A_T and B_T from psi(T) for all T\n");
//fflush(stdout);

//remove vertices in G that can be surrounded by T_i in 1 move from psi(T_i)

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT

   for(c=1; c<=order_G; c++){  //traverse cols of psiofT i.e. vertices of G
                               //only need to remove vertex c from psi(T_a)
      if(what_is_psi(psiofT, order_G, L, a, c)){//if vertex c is still in psi(T_a)

        if(can_w_be_surrounded(G, order_G, L, a, c)) //can vertex c be surrounded by
          set_psi(psiofT, order_G, L, a, c, 0);      //configuration T_a in 1 move?
      }                                              //if yes, then c is not safe.
   }
}

//printf("done removing C_T from psi(T) for all T\n");
//printf("starting intersection step\n");
//fflush(stdout);

////////////////////////// do intersection step ////////////////////////

long num_changes=1;
int result1, result2;
char *nhoodGT2;//used in intersection step, holds N_G[psi(T_2)]

if((nhoodGT2 = (char*) malloc(order_G*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
  exit(1);
}

for(c=0; c<order_G; c++)
   nhoodGT2[c]=0;

//long count = 0;
```

```
    while(num_changes>0){

        //count++;
        //printf("starting loop %ld \n", count);
        //fflush(stdout);

        num_changes=0;

        for(a=1; a<=order_GpowL; a++){//a represents T, b represents T' (see algorithm).
           for(b=a+1; b<=order_GpowL; b++){
        //for and if statements ensure we traverse all elements of E[(str prod)^L (G)].
              if(what_is_in_GpowL(G, order_G, L, a, b)){ //then (T_a,T_b) is an edge

                result1 = alter_psi_value(G, order_G, L, psiofT, a, b, nhoodGT2);
                                                          //alter psi(T_a)
                num_changes += result1;

                result2 = alter_psi_value(G, order_G, L, psiofT, b, a, nhoodGT2);
                                                          //alter psi(T_b)
                num_changes += result2;

                if(result1==2 || result2==2 || result1==3 || result2==3){
                  printf("found empty psi value when testing with %d cops\n", L);
                  fflush(stdout);
                  free(psiofT);
                  return 1;
                }
              }
           }//while loop is exited if we go through all the edges of
        }   //(str prod)^L(G) without changing any psi values, or if
    }        //alter_psi_value ever returns 2 or 3 (meaning psi(T_a)
             //or psi(T_b) is empty, see alter_psi_value() function).
    free(nhoodGT2);

    //printf("done intersection step\n");
    //fflush(stdout);

    /////////////// determine whether or not L cops win ///////////////

    char cops_win=1;

    for(c=0; c<order_G; c++){

       if(psiofT[c]==1){
         cops_win=0;
         break;
       }
    }//by corollary, only need to check if psi(T_1) is empty

/*
    ////////////////////// print psi(T) values //////////////////////

    for(a=1; a<=order_GpowL; a++){
       for(c=1; c<=order_G; c++){
          printf("%d", what_is_psi(psiofT, order_G, L, a, c));
```

```
        }
        printf("\n");
    }
    printf("\n");
    fflush(stdout);
*/

    free(psiofT);
    return cops_win;

}


char do_cops_surround_ver2C(char *G, int order_G, int L){

    /*returns 1 if L cops is enough to surround robber, 0 otherwise.
      Version 2C does NOT store adj matrix of (str prod)^L(G). Ver2C
      skips the intersection step for T-values that we know will not
      have their psi(T) values altered, and returns "cops win"
      immediately if it ever encounters an empty psi value.*/

    //printf("\nstarted testing with %d cops\n", L);
    //fflush(stdout);

    long a,b;//use these for counting up to order_GpowL or higher
    int c,d,e;//use these for counting up to order_G or L
    long order_GpowL=power(order_G, L);//power() subroutine will check if this
                                       //value is too large for the long data type
    if(order_GpowL > LONG_MAX/order_G){
      printf("values have become too large, see line %d\n", __LINE__);
      exit(1);
    }//makes sure that order_GpowL*order_G > LONG_MAX is false

    char *psiofT;

    if((psiofT = (char*) malloc(order_GpowL*order_G*sizeof(char)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit(1);
    }

    /////////////// initialize all psi(T) to V(G) ///////////////

    for(a=0; a<order_GpowL*order_G; a++)
       psiofT[a]=1;

    //printf("done initializing psi(T) to V(G) for all T\n");
    //fflush(stdout);

    ////////// remove vertices in T, vertices currently surrounded by T, and  //////////
    ////////// vertices that can be surrounded by T in 1 move from all psi(T) //////////

    int *cop_places; //pointer to array of ints that will serve as L-tuple form of
                     //index of a T-value. This rep's the layout of cops at that T-value
    if((cop_places = (int*) malloc(L*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
```

```
  exit(1);
}

for(c=0; c<L; c++)
   cop_places[c]=0;

int is_surrounded, occupied;

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT

   convert_index_ver2(order_G, L, a, cop_places);//find positions of cops in T_a

   for(c=0; c<L; c++)                                    //remove vertices in
      set_psi(psiofT, order_G, L, a, cop_places[c], 0);//T_a from psi(T_a)

   //remove vertices in G that are surrounded by T_a from psi(T_a)

   for(c=1; c<=order_G; c++){ //traverse vertices of G

      if(what_is_psi(psiofT, order_G, L, a, c)){//only need to remove entry in
                                                //column c if it is currently a 1
        is_surrounded=1; //start by assuming vertex c in G is surrounded by T_a

        for(d=1; d<=order_G; d++){

           if(what_is(G, order_G, c, d)){ //traverse vertices of N_G(c)

             occupied=0; //assume vertex d is not in T

             for(e=0; e<L; e++){//check if vertex d is in T

                if(d==cop_places[e]){
                  occupied=1;
                  break;
                }
             }

             if(occupied==0){    //vertex c is not surrounded if there
               is_surrounded=0;  //is a vertex in N_G(c) but not in T_a
               break;
             }
           }
        }

        if(is_surrounded)
          set_psi(psiofT, order_G, L, a, c, 0); //c is surrounded, remove c from psi(T_a)
      }
   }
}

free(cop_places);

//printf("done removing A_T and B_T from psi(T) for all T\n");
//fflush(stdout);
```

```
//remove vertices in G that can be surrounded by T_a in 1 move from psi(T_a)

for(a=1; a<=order_GpowL; a++){ //traverse rows of psiofT

   for(c=1; c<=order_G; c++){  //traverse cols pf psiofT i.e. vertices of G
                                                    //only need to remove vertex c from psi(T_a)
      if(what_is_psi(psiofT, order_G, L, a, c)){//if vertex c is still in psi(T_a)

        if(can_w_be_surrounded(G, order_G, L, a, c)) //can vertex c be surrounded by
          set_psi(psiofT, order_G, L, a, c, 0);      //configuration T_a in 1 move?
      }                                              //if yes, then c is not safe.
   }
}

//printf("done removing C_T from psi(T) for all T\n");
//printf("starting intersection step\n");
//fflush(stdout);

////////////////////////// do intersection step //////////////////////////

char sthn_has_changed=1;
char *prev_changed;//one element of this array for every T... this is a 1-D array

if((prev_changed =  (char*) malloc((order_GpowL)*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
  exit(1);
}

char *curr_changes;//one element of this array for every T... this is a 1-D array

if((curr_changes =  (char*) malloc((order_GpowL)*sizeof(char)))==NULL){
  printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
  exit(1);
}

for(a=0; a<order_GpowL; a++)
   prev_changed[a]=1;

char *nhoodGT2;//used fo intersection step, holds N_G[psi(T_2)]

if((nhoodGT2 = (char*) malloc(order_G*sizeof(char)))==NULL){
 printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
 exit(1);
}

for(c=0; c<order_G; c++)
   nhoodGT2[c]=0;

int found_empty_psi=0;
int value_1, value_2;

//long count = 0;

while(sthn_has_changed){
```

```
    //count++;
    //printf("starting loop %ld \n", count);
    //fflush(stdout);

    sthn_has_changed=0;

    for(a=0; a<order_GpowL; a++)
       curr_changes[a]=0;

    for(a=1; a<=order_GpowL; a++){//a represents T, b represents T' (see algorithm).

       if(prev_changed[a] || curr_changes[a]){

         for(b=1; b<=order_GpowL; b++){//in this version b must start at 1, NOT a+1

            if(what_is_in_GpowL(G, order_G, L, a, b)){ //then (T_a,T_b) is an edge

              value_1 = alter_psi_value(G, order_G, L, psiofT, a, b, nhoodGT2);

              if(value_1==1 || value_1==2){
                sthn_has_changed=1;
                curr_changes[a]=1;//psi(T_a) was just altered
              }

              if((value_1==2 || value_1==3) && found_empty_psi==0){
                found_empty_psi=1;
                printf("found first empty psi value when testing with %d cops\n", L);
                fflush(stdout);
                free(psiofT);
                return 1;
              }

              value_2 = alter_psi_value(G, order_G, L, psiofT, b, a, nhoodGT2);

              if(value_2==1 || value_2==2){
                sthn_has_changed=1;
                curr_changes[b]=1;//psi(T_b) was just altered
              }

              if((value_2==2 || value_2==3) && found_empty_psi==0){
                found_empty_psi=1;
                printf("found first empty psi value when testing with %d cops\n", L);
                fflush(stdout);
                free(psiofT);
                return 1;
              }
            }
         }
       }
    }

    for(a=0; a<order_GpowL; a++)
       prev_changed[a]=curr_changes[a];
}
```

```
    free(nhoodGT2);

    //printf("done intersection step\n");
    //fflush(stdout);

    /////////////// determine whether or not L cops win ///////////////

    char cops_win=1;

    for(c=0; c<order_G; c++){

       if(psiofT[c]==1){
         cops_win=0;
         break;
       }
    }//by corollary, only need to check if psi(T_1) is empty

/*
    ////////////////////// print psi(T) values //////////////////////

    for(a=1; a<=order_GpowL; a++){
       for(c=1; c<=order_G; c++){
          printf("%d", what_is_psi(psiofT, order_G, L, a, c));
       }
       printf("\n");
    }
    printf("\n");
    fflush(stdout);
*/

    free(psiofT);
    return cops_win;

}


int find_surround_num_ver2B(char *G, int order_G){

    //printf("\nrunning find_surround_num_ver2B\n");
    //fflush(stdout);

    int hold=order_G;
    int deg,j;

    for(int i=1; i<=order_G; i++){

       deg=get_vertex_degree (G, order_G, i);

       if(deg<hold)
         hold=deg;

    }

    printf("minimum degree of all vertices on G is %d\n", hold);
    fflush(stdout);
```

```
    //start j at "(hold)" since s(G) cannot be less than the min degree of vertices in G

    for(j=hold; j<=order_G; j++){
       if(do_cops_surround_ver2B(G,order_G,j)){
         return j;
       }
       printf("s(G) is not %d\n", j);
       fflush(stdout);
    }

    printf("error - no surrounding cop number was found. see line %d", __LINE__);
    exit(1);
    return 0;

}


int find_surround_num_ver2C(char *G, int order_G){

    //printf("\nrunning find_surround_num_ver2C\n");
    //fflush(stdout);

    int hold=order_G;
    int deg,j;

    for(int i=1; i<=order_G; i++){

       deg=get_vertex_degree (G, order_G, i);

       if(deg<hold)
         hold=deg;

    }

    printf("minimum degree of all vertices on G is %d\n", hold);
    fflush(stdout);

    //start j at "(hold)" since s(G) cannot be less than the min degree of vertices in G

    for(j=hold; j<=order_G; j++){
       if(do_cops_surround_ver2C(G,order_G,j)){
         return j;
       }
       printf("s(G) is not %d\n", j);
       fflush(stdout);
    }

    printf("error - no surrounding cop number was found. see line %d", __LINE__);
    exit(1);
    return 0;

}
```

```
void compute_something(char *G, int order_G, int choice){

     if(choice!=1 && choice!=2 && choice!=3 && choice!=4){
       printf("pick a valid choice, see line %d", __LINE__);
       exit(1);
     }

     int ans;

     if(choice==1){
       ans = find_cop_number_ver2B(G, order_G);
       printf("*****cop number of graph G is %d***** \n\n", ans);
       fflush(stdout);
     }

     if(choice==2){
       ans = find_cop_number_ver2C(G, order_G);
       printf("*****cop number of graph G is %d***** \n\n", ans);
       fflush(stdout);
     }

     if(choice==3){
       ans = find_surround_num_ver2B(G, order_G);
       printf("*****surrounding cop number of graph G is %d***** \n\n\n", ans);
       printf("////////////////////////////////////////////////// \n\n");
       fflush(stdout);
     }

     if(choice==4){
       ans = find_surround_num_ver2C(G, order_G);
       printf("*****surrounding cop number of graph G is %d***** \n\n\n", ans);
       printf("////////////////////////////////////////////////// \n\n");
       fflush(stdout);
     }

}


int alter_psi_value(char *G, int order_G, int L, char *psimatrix, long i, long j, char *nhoodj){

    /*1<=i,j<=order_G^L is required. returns 0,1,2, or 3 depending on
      whether psi value was changed and/or an empty psi value was found.
      changes psi(T_i), depending on the neighborhood in G of psi(T_j)
      (see algorithm). does NOT change psi(T_j). row containing entries
      of psi(T_A) is row A, 1<=A<=order_G^L. also takes nbrhoodj as
      input so memory for it  is not constantly being created and
      freed up by this function. NOTE that the return of this
      function is used differently in ver2B than in ver2C of code.*/

    int i_empty=1;
    int j_empty=1;
    int found_empty=0;
    int x,y;

    if(order_G<1 || L<1 || L>order_G || i<1 || j<1 || i>power(order_G,L) || j>power(order_G,L)){
```

```
  printf("error, see line %d", __LINE__);
  exit(1);
}

for(x=0; x<order_G; x++)
   nhoodj[x]=0;

for(x=1; x<=order_G; x++){

   if(what_is_psi(psimatrix, order_G, L, j, x)){//i.e. if vertex x is in psi(T_j)

     j_empty=0;//psi(T_j) is nonempty

     nhoodj[x-1]=1;

     for(y=1; y<=order_G; y++){
        if(what_is(G, order_G, x, y))
          nhoodj[y-1]=1;
     }
   }
}

int changes_made=0;

for(x=1; x<=order_G; x++){

   if(what_is_psi(psimatrix, order_G, L, i, x)){//"if x is in psi(T_i)"

     i_empty=0;//psi(T_i) is nonempty

     if(nhoodj[x-1]==0){                          //"and x is NOT in N_G[psi(T_j)]"
       set_psi(psimatrix, order_G, L, i, x, 0); //"then remove x from psi(T_i)"
       changes_made=1;
     }
   }
}

if(i_empty || j_empty)
  found_empty=1;

if(changes_made==0 && found_empty==0)
  return 0;

if(changes_made==1 && found_empty==0)
  return 1;

if(changes_made==1 && found_empty==1)
  return 2;

if(changes_made==0 && found_empty==1)
  return 3;

printf("something went wrong in alter_psi_value function");
return 4;
```

```
}


int can_w_be_surrounded(char *G, int order_G, int exp, int T_index, int vertex_w){

    /*T is a configuration of "exp" cops on the graph G and
      w is a vertex in G. returns 1 if w can be surrounded
      by cops of T in one move, returns 0 otherwise.*/

    if(vertex_w<1 || vertex_w>order_G || T_index<1 || T_index>power(order_G,exp)){
      printf("error, see line %d", __LINE__);
      exit(1);
    }

    int x,y;

    ///////////////// get positions of cops in T ////////////////

    int *T_cops;

    if((T_cops = (int*) malloc(exp*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit(1);
    }

    for(x=0; x<exp; x++)
       T_cops[x]=0;
                                                    //stores positions of cops in T_cops.
    convert_index_ver2(order_G, exp, T_index, T_cops);//remember each T_i is indexed by the
                                                    //same # as its corresponding psi(T_i)
    ///////////////// get positions of vertices in N_G(w) ////////////////

    int deg_w = get_vertex_degree (G, order_G, vertex_w); //equal to |N_G(w)|
    int *nhood_w;

    if((nhood_w = (int*) malloc(deg_w*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit(1);
    }

    for(x=0; x<deg_w; x++)
       nhood_w[x]=0;

    find_nhood(G, order_G, vertex_w, deg_w, nhood_w);

    ///////////////// create directed graph of flow from s to t ////////////////

    //along rows and cols of flow graph, order source s, sink t, cops in T,
    //vertices in N_G(w) like so: s, cops 1 to exp, vertices 1 to deg_w, t.

    int num_vertices = exp + deg_w + 2;
    int *flow_graph; //directed graph from source s to sink t

    if((flow_graph = (int*) malloc(num_vertices*num_vertices*sizeof(int)))==NULL){
      printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
```

```
      exit(1);
    }

    for(x=0; x<num_vertices*num_vertices; x++)
       flow_graph[x]=0; //most of flow_graph is 0's

    for(x=1; x<=exp; x++)//row# = 0, 1<= col# <= #ofcops = exp
       flow_graph[x]=1;  //there is 1 edge FROM s TO each cop

    for(x=exp+2; x<num_vertices; x++)//now indexing starts at 1, ends at num_vertices
       set_bit_D (flow_graph, num_vertices, x, num_vertices, 1);
       //there is 1 edge FROM each vertex in N_G(w) TO t

    for(x=2; x<exp+2; x++){//rows of cops in T

       for(y=exp+2; y<num_vertices; y++){//cols of vertices in N_G(w)
          //below if statement is asking if cop x can move onto vertex y in G
          if(what_is (G, order_G, T_cops[x-2], nhood_w[y-exp-2]) || T_cops[x-2]==nhood_w[y-exp-2]){
   //recall that cops are indexed from 2 to exp+1 in flow_graph and from 0 to exp-1 in T_cops.
   //vertices are indexed from exp+2 to exp+deg_w+1 in flow_graph and from 0 to deg_w-1 in nhood_w.
            set_bit_D (flow_graph, num_vertices, x, y, 1);
          }         //there is an edge going FROM cop at row x TO vertex at col y
       }
    }

    ///////////////// run Ford-Fulk. alg., decide if w can be surrounded by T ////////////////

    int value=Ford_Fulk(flow_graph, num_vertices, 1, num_vertices);
    int can_be_surrounded;              //find max flow from s to t

    if(value>deg_w){
      printf("error, see line %d", __LINE__);
      exit(1);//edges each have weight of 1 so value>deg_w should be impossible
    }
    if(value==deg_w)   //then (at least) deg_w cops can move onto N_G(w) in 1 move
      can_be_surrounded=1;
    else               //then w cannot be surrounded by T in one move
      can_be_surrounded=0;

    free(T_cops);
    free(nhood_w);
    free(flow_graph);
    return can_be_surrounded;

}


void find_nhood(char *G, int order_G, int v, int deg_v, int *nhood){

     /*nhood must be initialized to the order of deg(v) and to all 0's.
       function will store the #'s identifying vertices in N_G(v) in the
       1-D array nhood. stored in a similar way as convert_index_ver2
       stores positions of cops!*/

     if(v<1 || v>order_G || deg_v<1 || deg_v>=order_G){
```

```
        printf("error, see line %d", __LINE__);
        exit(1);
      }

      int start=1;
      int x,y;

      for(x=0; x<deg_v; x++){//traverse array nhood

         for(y=start; y<=order_G; y++){//traverse V(G) starting from 1
                                       //after last vertex added to nhood
            if(what_is (G, order_G, v, y)){
              nhood[x]=y;
              start=y+1;
              break;
            }
         }
      }
}


char what_is_in_GpowL(char *G, int order_G, int exp, long i, long j){

      long order_ans=power(order_G, exp);
      int a;

      if(i<1 || i>order_ans || j<1 || j>order_ans){
        printf("ensure i and j are valid. see line %d \n", __LINE__);
        exit(1);
      }

      if(i>j)
        return what_is_in_GpowL(G, order_G, exp, j, i);

      if(i==j)
        return 0;

      int *I;
      int *J;

      if((I = (int*) malloc(exp*sizeof(int)))==NULL){
        printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
        exit(1);
      }

      if((J = (int*) malloc(exp*sizeof(int)))==NULL){
        printf("\n\nAllocation of memory failed...line %d\n", __LINE__);
        exit(1);
      }

      for(a=0; a<exp; a++){
         I[a]=0;
         J[a]=0;
      }
```

```
     convert_index_ver2(order_G, exp, i, I);//maps index i to exp-tuple I
     convert_index_ver2(order_G, exp, j, J);//maps index j to exp-tuple J

     for(a=0; a<exp; a++){

        if(what_is(G, order_G, I[a], J[a])==0){
          if(I[a]!=J[a]){
            free(I);
            free(J);
            return 0;
          }
        }
     }

     free(I);
     free(J);
     return 1;

}


char what_is_psi(char *psi_of_T, int order_G, int exp, long i, int j){

   /*returns value in row i column j of (order_G^exp)x(order_G)
     matrix holding psi(T) values*/

     if(i<1 || j<1 || i>power(order_G,exp) || j>order_G ){
       printf("error, see line %d", __LINE__);
       exit(1);
     }                                    //access entry (row,col) of a 2D matrix
                                          //being represented by a 1D array by accessing
     return psi_of_T[(i-1)*order_G + (j-1)];//entry row*(#ofcols)+col in 1D array. note
                                          //that 0<=row<=#ofrows-1 and 0<=col<=#ofcols-1
}//row index, i, must be type long bc order_G^exp=(numberofrows) may be too large for an int


void set_psi(char *psi_of_T, int order_G, int exp, long i, int j, char value){

     /*sets new value to entry in row i, col j of
     (order_G^exp)x(order_G) matrix holding psi(T) values*/

     if(i<1 || j<1 || i>power(order_G,exp) || j>order_G || (value!=0 && value!=1)){
       printf("error, see line %d", __LINE__);
       exit(1);
     }                                     //access entry (row,col) of a 2D matrix
                                           //being represented by a 1D array by accessing
     psi_of_T[(i-1)*order_G + (j-1)] = value;//entry row*(#ofcols)+col in 1D array. note
                                           //that 0<=row<=#ofrows-1 and 0<=col<=#ofcols-1
}//row index, i, must be type long bc order_G^exp=(numberofrows) may be too large for an int


char *build_K_mn_graph(int m, int n){

     /*this function creates and returns an adjascency
       matrix for the following complete bipartite graph:
```

```
    on the left, there are m vertices labeled u_1, u_2,... ,u_m
    on the right, there are n vertices labeled u_m+1, u_m+2,... ,u_m+n
    vertices are then connected in the expected way to create K_m,n*/

  if(m<1 || n<1){
    printf("error, see line %d", __LINE__);
    exit(1);
  }

  int order_G = m + n; // |V(G)| = m + n
  char *G;
  int x,y;

  if((G = (char*) malloc(((order_G*order_G-order_G)/2)*sizeof(char)))==NULL){
    printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
    exit(1);
  }

  for(x=1; x<=order_G; x++){//traverse rows of G
     for(y=x+1; y<=order_G; y++){//traverse columns of G

        if(x<=m && y>m)
          set_bit(G, order_G, x, y, 1);
        else
          set_bit(G, order_G, x, y, 0);
     }
  }

  return G;

}


char *build_Kmn_str_prod_Kmn(int m, int n){

  char *G=build_K_mn_graph(m, n);
  int order_G = m + n;
  char *H=build_str_prod(G, order_G, G, order_G);

  free(G);
  return H;

}


char *build_GP_nk(int n, int k){

  /*creates the generalized petersen graph GP(n,k) based on inputs n and k */

  if(n<3 || k<1 || k>=n){//really should never let k>(n/2)
    printf("pick valid parameters, see line %d", __LINE__);
    exit(1);
  }

  int i,j;
```

```
    char *G;
    int order_G=2*n;

    if((G = (char*) malloc(((order_G*order_G-order_G)/2)*sizeof(char)))==NULL){
      printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
      exit(1);
    }

    for(i=0; i<((order_G*order_G)-order_G)/2; i++)
       G[i]=0;

    for(i=1; i<n; i++)
       set_bit(G, order_G, i, i+1, 1);

    set_bit(G, order_G, 1, n, 1);//outer loop in now created

    for(i=1; i<=n; i++)
       set_bit(G, order_G, i, i+n, 1);//spokes are now created

    for(i=n+1; i<=order_G; i++){//create inner "star"

       j=i+k;

       if(j>order_G)
         j-=n;

       set_bit(G, order_G, i, j, 1);
    }

    return G;

}


char *build_Pm_cartprod_Pn(int m, int n){

    /*builds and returns P_m(cartesian product)P_n
      (def: P_m is a path on m vertices)*/

    if(m<2 || n<2){//cant have a path on 1 vertex
      printf("pick valid parameters, see line %d", __LINE__);
      exit(1);
    }

    char *G;
    int order_G=m*n;
    int x,y;

    if((G = (char*) malloc(((order_G*order_G-order_G)/2)*sizeof(char)))==NULL){
      printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
      exit(1);
    }

    for(x=1; x<=order_G; x++){//traverse rows of adjascency matrix of G
       for(y=x+1; y<=order_G; y++){//traverse columns of adjascency matrix of G
```

```
            if(y==(x+n))//n=number of columns
              set_bit(G, order_G, x, y, 1);
            else
              set_bit(G, order_G, x, y, 0);
          }
        }//creates all vertical edges

        for(x=1; x<=order_G; x++){//NOT traversing adjacency matrix here
           if((x%n) != 0) //i.e. if vertex is NOT in far right column
             set_bit(G, order_G, x, x+1, 1);
        }//creates all horizontal edges

        return G;

}


char *build_Pm_strprod_Pn(int m, int n){

        /*builds and returns P_m(strong product)P_n
          (def: P_m is a path on m vertices)*/

        if(m<2 || n<2){//cant have a path on 1 vertex
          printf("pick valid parameters, see line %d", __LINE__);
          exit(1);
        }

        int x,y;

        char *Pm;
        int order_Pm = m;

        char *Pn;
        int order_Pn = n;

        if((Pm = (char*) malloc(((order_Pm*order_Pm-order_Pm)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
          exit(1);
        }

        if((Pn = (char*) malloc(((order_Pn*order_Pn-order_Pn)/2)*sizeof(char)))==NULL){
          printf("\n\nAllocation of memory failed for GRAPH1...line %d\n", __LINE__);
          exit(1);
        }

        for(x=1; x<=order_Pm; x++){//traverse rows of Pm
           for(y=x+1; y<=order_Pm; y++){//traverse columns of Pm

              if(y==x+1)
                set_bit(Pm, order_Pm, x, y, 1);
              else
                set_bit(Pm, order_Pm, x, y, 0);
           }
        }//make Pm
```

```
      for(x=1; x<=order_Pn; x++){//traverse rows of Pn
         for(y=x+1; y<=order_Pn; y++){//traverse columns of Pn

            if(y==x+1)
              set_bit(Pn, order_Pn, x, y, 1);
            else
              set_bit(Pn, order_Pn, x, y, 0);
         }
      }//make Pn

      char *G=build_str_prod(Pm, order_Pm, Pn, order_Pn);

      free(Pm);
      free(Pn);
      return G;

}


//////// The following subroutines were created by Dr. David Pike ////////
////////  and were altered very slightly for use in this program  ////////


char what_is(char *A, int n, int i, int j){

      if(i<1 || i>n || j<1 || j>n){
        printf("error, see line %d", __LINE__);
        exit(1);
      }

      if(i > j)
        return what_is (A, n, j, i);

      else if(i == j)
             return 0;

      else
        return A[(i-1)*n - (i*(i-1))/2 + (j-i) - 1];

}


void set_bit (char *A, int n, int i, int j, char value){

      if(i<1 || i>n || j<1 || j>n || (value!=0 && value!=1)){
        printf("error, see line %d", __LINE__);
        exit(1);
      }

      if(i > j)
        set_bit (A, n, j, i, value);

      else if(i == j){
```

```
            if(value){
              printf ("\n\nTrying to set A[%d,%d]\n\n", i, j);
              exit (1);
            }

          }

          else
            A[(i-1)*n - (i*(i-1))/2 + (j-i) - 1] = value;

}


int get_vertex_degree (char *A, int n, int vertex){

    if(vertex<1 || vertex>n){
      printf("error, see line %d", __LINE__);
      exit(1);
    }

    int degree = 0;
    int index;

    for(index = 1 ; index <= n ; index ++){
       if(what_is (A, n, vertex, index))
         degree ++;
    }

    return degree;

}


int what_is_D (int *D, int n, int i, int j){

    if(i<1 || i>n || j<1 || j>n){
      printf("error, see line %d", __LINE__);
      exit(1);
    }

    return D[n*(i-1) + j - 1];
        //same as (row)*(#ofcols)+(col)
}


void set_bit_D (int *D, int n, int i, int j, int value){

     if(i<1 || i>n || j<1 || j>n || (value!=0 && value!=1)){
       printf("error, see line %d", __LINE__);
       exit(1);
     }

     D[n*(i-1) + j - 1] = value;
        //same as (row)*(#ofcols)+(col)
}
```

```
char find_xy_path(int *D, int n, int x, int y, path_node **Path){

   path_node *Queue, *temp; //Find a directed path in D from x to y... if such a path exists,
   int *predecessors;       //set Path to it and return 1; else return 0.  Note that, as
   int index;               //recommended, we search for the path with fewest arcs, using a
   char ret_val;            //breadth-first search paradigm.

   if((predecessors = (int *) malloc (n * sizeof (int))) == NULL){
      printf ("\n\nAllocation of memory failed...line %d\n", __LINE__);
      exit (1);
   }

   for(index = 1 ; index <= n ; index ++)
      predecessors [index - 1] = 0;

   Queue = NULL;
   enqueue(&Queue, x);

   while((Queue != NULL) && (predecessors [y - 1] == 0)){

      for(index = 1 ; index <= n ; index ++){

         if(predecessors [index - 1] == 0)

            if(what_is_D (D, n, Queue->vertex, index) > 0){

               /*Then we have found how to get to vertex index...
                 store index's predecessor and add index to the
                 end of the Queue.*/
               predecessors [index - 1] = Queue->vertex;

               enqueue (&Queue, index);

            }

      }

      /* remove the head of the Queue */
      temp = Queue;
      Queue = Queue->next;
      free (temp);

   }/* end while */

   /* free up any memory still allocated to the Queue */

   while (Queue != NULL){
      temp = Queue->next;
      free (Queue);
      Queue = temp;
   }

   ret_val = 0;
```

```
   if(predecessors [y-1] > 0)
      ret_val = 1;

   // ret_val = predecessors [y - 1];

   if(ret_val){

      /* we must build the path from x to y */

      if((*Path = (path_node *) malloc (sizeof (path_node))) == NULL){
         printf ("\n\nAllocation of memory failed...line %d\n", __LINE__);
         exit (1);
      }

      (*Path)->vertex = y;
      (*Path)->next = NULL;

      while((*Path)->vertex != x){

         if((temp = (path_node *) malloc (sizeof (path_node))) == NULL){
            printf ("\n\nAllocation of memory failed...line %d\n", __LINE__);
            exit (1);
         }

         temp->vertex = predecessors [(*Path)->vertex - 1];
         temp->next = *Path;
         *Path = temp;

      }

   }

   free (predecessors);
   return ret_val;

}


void enqueue(path_node **Queue, int value){

   path_node *temp;

   if(*Queue != NULL){

      for(temp = *Queue ; temp->next != NULL ; temp = temp->next);

      if((temp->next = (path_node *) malloc (sizeof (path_node))) == NULL){
         printf ("\n\nAllocation of memory failed...line %d\n", __LINE__);
         exit (1);
      }

      temp->next->vertex = value;
      temp->next->next = NULL;
```

```
   }

   else{

         if((*Queue = (path_node *) malloc (sizeof (path_node))) == NULL){
            printf ("\n\nAllocation of memory failed...line %d\n", __LINE__);
            exit (1);
         }

         (*Queue)->vertex = value;
         (*Queue)->next = NULL;

      }

}


int Ford_Fulk(int *A, int n, int x, int y){

   int index1, index2;
   path_node *Path, *temp, *temp2;
   int edge_connectivity = 0;

   /*From here on we treat D as the residual network described on pages
     593-597 of "Introduction to Algorithms" by Cormen, Leiserson, and
     Rivest (text for CS-466).*/

   Path = NULL;

   while(find_xy_path (A, n, x, y, &Path)){

      edge_connectivity ++;
      temp = Path;

      /* augment D along Path */

      while (temp->next != NULL){

         set_bit_D (A, n, temp->vertex, temp->next->vertex,

                    what_is_D (A, n, temp->vertex, temp->next->vertex) - 1);

         set_bit_D (A, n, temp->next->vertex, temp->vertex,

                    what_is_D (A, n, temp->next->vertex, temp->vertex) + 1);

         temp2 = temp;
         temp = temp->next;
         free (temp2);

      }/* end while */

      free (temp);

      /* prepare for next iteration */
```

```
        Path = NULL;

    }/* end while */

    return edge_connectivity;

}
```