OPEN-LOCATING-DOMINATING SETS

By

Robert Dohner

APPROVED:

Graduate Committee:

_____

Supervisor Dr. Suk Jai Seo (Computer Science)

_____

Dr. Cen Li (Computer Science)

_____

Dr. Salvador E. Barbosa (Computer Science)

_____

Dr. Chrisila Pettey, Chairperson Computer Science Department

_____

Dr. David L. Butler, Dean of the College of Graduate Studies

OPEN-LOCATING-DOMINATING SETS

By

Robert Dohner

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

Middle Tennessee State University

May 2020

ACKNOWLEDGEMENTS

Special thanks to Dr. Seo for providing leadership and vision towards my goals in this project and to my mother for supporting me throughout my academic journey.
Thank you to my thesis committee members who helped refine and add to my thesis.

ABSTRACT

For a graph G, distinguishing sets can be used for detection purposes. Whether it be setting up sensors to detect a thief in a facility or detecting a faulty component in a network of processors, these types of sets can be used to minimize the number of sensors required for the grid or network. Recently, there has been a lot of research into dealing with distinguishing sets, including on open-locating-dominating (OLD) sets, which is the graphical parameter this research focuses on.

A dominating set D is a subset of vertices in G where each of the vertices in G is either in D or adjacent to some vertex in D. An open-locating-dominating set is a special dominating set that is used to detect and pinpoint exact location of a problem vertex in a system. For a given graph G, OLD(G) denotes the minimum cardinality of an OLD set and the problem of determining the value of OLD(G) for an arbitrary graph G is known to be NP-complete.

For our research, programs were developed to find OLD sets and the value of OLD(G) for various classes of graphs. In particular, in order to find the minimum density of an OLD set for the infinite king's graph, an algorithm was created that parallelized the program using a backtracking algorithm and the program has confirmed the previous results. Most importantly we present a proof for the NP-completeness of a fault-tolerant OLD set called a redundant OLD set.

Keywords: dominating set, distinguishing set, open-locating dominating set, redundant OLD set, infinite king’s graph, NP-complete

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

OLD Set – Open-Locating-Dominating Set

OLD_OIND – Open-Independent Open-Locating-Dominating

RED:OLD – Redundant Open-Locating-Dominating

CHAPTER I

INTRODUCTION

A *graph* is a structure of a set of objects that are related to each other. These objects are stated to be *vertices* and the relations between these objects are called *edges*. Using these sets of vertices and edges, we can create a diagram of various relationships between objects, commonly called a graph. There are many uses for graphs. A couple of such uses would be to recreate a floor plan of museum rooms to detect an intruder or to diagram a network of cell towers to cover customers in a certain area.

If we wanted to design a detection array for a museum floor plan, or if we wanted to set up those cell towers in a certain way, then we would have to look at more specific graphs with special properties. For example, imagine setting up cell phone towers. One would ideally like to cover an entire area. One would also like to locate cell phone users through triangulation of towers a user pings when making a cell phone call. When one wants to completely cover an area and be able to uniquely recognize an issue or problem within that area, the best way to solve this problem is through a special type of set of vertices when the network is modeled as a graph. This set is known as a *distinguishing set*, which has the following two properties. One is to cover the entire set of vertices and the other is to uniquely identify a vertex within the set. Distinguishing sets in general are used for detection purposes, whether it be setting up sensors to detect a thief in a facility or detecting a faulty component in a network of processors.

If we were wanting to have coverage of an entire area, such as in the above example where we want our cell phones to cover an entire area, we would look to a *dominating set*. A dominating set D is a subset of vertices in G where every vertex in G

is either in D or adjacent to some vertex in D [6]. This means that every vertex in the graph that is not within the dominating set D must be next to a vertex that belongs to the dominating set. As we see in Figure 1, the two vertices highlighted form a dominating set because all the other vertices that are not highlighted are adjacent to at least one of the highlighted vertices. The symbol γ (G) denotes the minimum size of a dominating set of a graph G. Since we need at least two vertices to dominate the graph G, we have γ (G) =2. Consider another example of a dominating set of the infinite path graph, denoted as P_∞, as shown in the first graph of Figure 2. Note that the label below each vertex indicates which darkened vertex dominates (or covers) the vertex. We observe that at least 1/3 of the entire vertices are needed to dominate the entire graph. This graph is an infinite graph and we state $\gamma\%(P_\infty) = 1/3$.

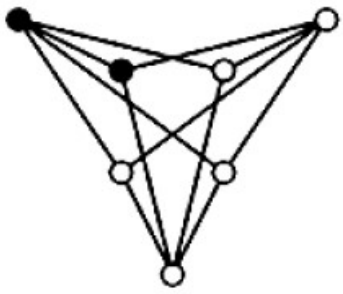


Figure 1: A dominating set of a graph G [7]

As mentioned above, another type of set of vertices we want to look at allows us to uniquely identify a certain vertex by placing a detection device at a subset of vertices. These vertex sets are called distinguishing sets, which include *identifying code* [8], a

locating-dominating set [6], and an *open-locating-dominating set* [3]. There is a bibliography of currently over 420 papers on the topic of distinguishing sets that are maintained by Lobstein [10]. An identifying code is a subset of vertices in G such that any vertex in G can be uniquely identified by examining the vertices around it [8]. Here, the detection device at each of the subset of vertices can identify problems at its neighbors, including itself. In the first graph in Figure 2, the detector at vertex 2 covers three vertices and if an intruder is detected by vertex 2, we don't know exactly which vertex is in trouble. If we consider the second graph in Figure 2, the label set below each of the vertices indicates which darkened vertices dominates the vertex. We observe that these label sets are distinct, and hence the set of darkened vertices can uniquely identify a vertex that is in trouble. In other words, these darkened vertices form an identifying code (IC) of the infinite path. Furthermore, we need at least half of the vertices to uniquely identify a vertex with issue, which is denoted as IC%(P_∞) = 1/2.

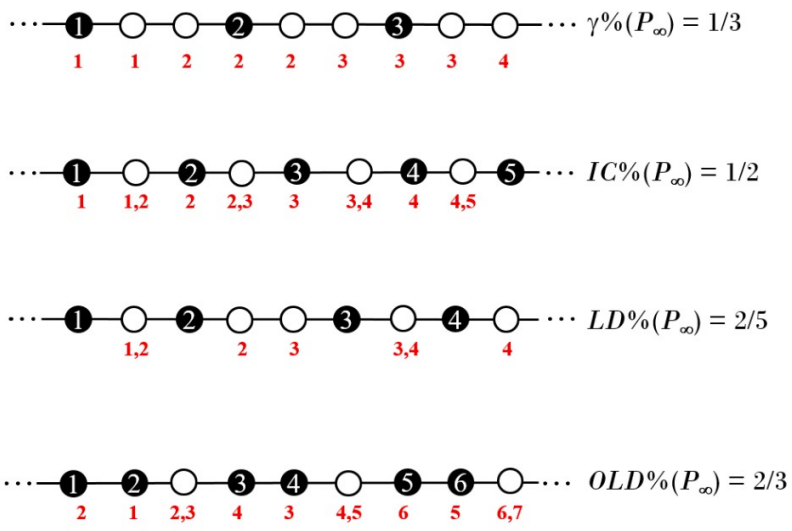


Figure 2: A dominating set, a locating-dominating set, an identifying code, and an open-locating dominating set for the infinite path graphs [10]

Locating-dominating sets are similar to identifying codes, but the detection device has an additional capability of differentiating the issue or problem at itself from an issue or problem at its neighbors. The third line in Figure 2 shows us a locating-dominating set for the infinite path. Similar to the discussion above, the labels below each vertex indicate which vertices dominate the vertex. The highlighted vertices do not have labels below because they identify themselves. Again, we observe that at least 2/5 of the vertices are needed to uniquely identify a vertex with an intruder and hence we have $LD\%(P_\infty) = 2/5$.

There is another type of distinguishing set, which considers a special situation. For example, cell phone towers are notorious for having poor to no signal at the base of the tower. In order for people who are living under a cell phone tower to receive a signal, they would have to be covered by another cell phone tower, not by the closest one. In the case of our distinguishing sets, this means a vertex cannot dominate itself. This leads to the sets we will be mainly looking at, which are open-locating-dominating sets (OLD sets). OLD sets have the same properties of an identifying code, except that a vertex cannot dominate itself [3]. Consider the last graph in Figure 2. Clearly, we see that at least 2/3 of vertices are needed to uniquely identify a vertex with an intruder when the detector cannot detect an issue at itself and hence, we get $OLD\%(P_\infty) = 2/3$. For our research, we will focus on multiple types of these OLD sets.

Another type of OLD set we will focus on is a fault-tolerant OLD set. A fault-tolerant OLD set is an OLD set where if something were to go wrong with a detection device at the vertex, then it still maintains the properties of an OLD set. The problem of

finding an OLD set for an arbitrary graph is known to be *NP-complete* [9], and we believe that the problem of determining a *fault-tolerant OLD-set* is also NP-complete. In this research, we will focus on proving the NP-completeness of the problem. Simply speaking, NP-complete problems are the problems for which an efficient algorithm does not exist. An NP-complete problem is a decision problem which has the property where a candidate solution of a problem can be verified in polynomial time but can only be solved in non-deterministic polynomial time. When we try to find an efficient algorithm for a given problem, but fails to do it, then proving its NP-completeness is considered a significant contribution toward the research on the problem. It is because if we find an efficient algorithm for one NP-complete problem, it is equivalent to finding an efficient algorithm for numerous problems that the researchers from all over the world have struggled to solve efficiently for many years. See Garey and Johnson [4] for details on NP-completeness.

There are two types of graphs we will be looking at when researching our open-locating-dominating sets. The first type of graph is a *finite graph*, which is a graph that has a finite number of vertices and a finite number of edges. The second type of graph we will look at is an *infinite graph*. These graphs go on infinitely with an infinite number of vertices and an infinite number of edges. The types of infinite graphs we will look at typically have a pattern to them.

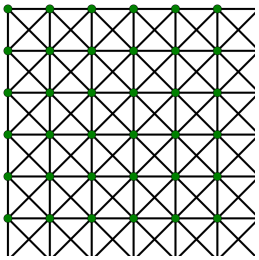


Figure 3: A king's graph [17]

First, we will focus on OLD sets in the *infinite king's graph*. A king's graph is a graph where each vertex is connected to eight other vertices, like a king's movement on a chessboard as shown in Figure 3. This graph will be considered to be non-directional and infinite. Our goal is to find the minimum density OLD set on the infinite king's graph.

Secondly, we will focus on OLD sets on finite graphs. We discuss an algorithm that will find all possible OLD sets on these finite graphs. Finally, we will look at fault tolerant OLD sets. For these sets, we will be focusing on creating a proof that categorizes finding these types of sets as NP-complete.

Specific Aims:

1. To find the minimum density open-locating-dominating set for the infinite king graph.
2. To develop a computer algorithm to find open-locating-dominating sets for finite graphs.
3. To prove that finding a fault tolerant open-locating-dominating set is NP-complete.

The next chapter introduces the results on various distinguishing set concepts and NP-completeness. In Chapter 3, we discuss finding the development of our algorithm, its

implementation, and testing to find the minimum density of an OLD set for the infinite king's graph. In Chapter 4, we discuss an implementation of a program and its testing to find OLD sets for any finite graphs. In Chapter 5, we look over a NP-completeness proof for a fault-tolerant OLD set. Chapter 6 summarizes our findings and goes into possible future works.

CHAPTER II

BACKGROUND

In this chapter, we will present some background information regarding distinguishing sets. First, we will discuss the different types of distinguishing sets for finite and infinite graphs. We will also look at some NP-complete proofs for some of these distinguishing sets.

When dealing with distinguishing sets in a graph, we are interested in determining the minimum number of vertices required to cover the graph. If the graph is infinite, we use the minimum density, which is defined to be a ratio of the number of vertices in the distinguishing set over the number of vertices in the entire graph. First, we will look at identifying codes in an infinite graph.

a. Identifying codes in infinite graphs

Charon et al. showed the minimum and maximum bounds for identifying codes on three types of grids [1]. They proved that these upper and lower bounds can be formed with a variable radius, but for our research, we only care about a radius of 1. With a radius of 1, the lower and upper bounds for the infinite square graph can be placed in between 3/8 and 2/5. For the infinite triangular graph, those bounds are 1/3 and 1/2. Finally, on the hexagonal grid, those bounds are between 3/8 and 2/5.

Theorem 1 [1]: For the infinite square grid SQ, the infinite triangular grid TRI, and the infinite hexagonal grid HEX, the minimum densities are $3/8 \leq IC\%(SQ) \leq 2/5$, $1/3 \leq IC\%(TRI) \leq 1/2$, $3/8 \leq IC\%(HEX) \leq 2/5$, respectively.

Charon et al. also went on to study identifying codes on the infinite king's graph [2]. They were able to narrow both the upper and the lower bound to be the same number. Therefore, the minimum density for an identifying code is 1/4.

Theorem 2 [2]: The minimum density for identifying codes for the infinite king's grid KING is IC%(KING) = 1/4.

b. Locating-dominating sets in infinite graphs

Honkala and Laihonen focuses on finding the smallest locating-dominating set on infinite grids [6]. The solution focuses on four types of grids: a square grid, a king grid, a triangular grid, and a hexagonal mesh grid. They first look at the king grid and made various tests and analysis on that grid. Then, they made a modification to the problem where they arbitrarily choose the divisions of the codes and see what the smallest density of the set would be.

With set divisions on the king grid, they determined that the smallest locating-dominating set would have a density of 1/5 as shown in Figure 4. While Slater showed that the minimum density for a locating-dominating square grid was roughly 30% [12], Honkala and Laihonen were able to bring that estimate down to 2/7 for that grid. As for the infinite triangular grid and hexagonal grid/mesh, they came to the conclusions of roughly 2/9 and 1/3, respectively. Furthermore, Honkala expanded the research into the triangular grid and came up with an even more refined number of 13/57, which was then proven [4].

Theorem 3 [4, 6]: The minimum density for a locating-dominating set for the infinite square, the hexagonal, the triangular, and the king's grid are 2/7, 2/9, 13/57, and 1/5 respectively.

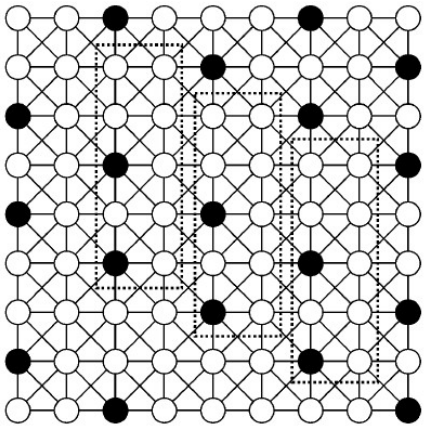


Figure 4: A set showing the minimum density of locating-dominating set of a king's graph [5]

c. Open-locating-dominating sets in finite graphs

Open-locating-dominating sets are similar to identifying codes, except that a vertex cannot dominate itself. This unique property changes the how these sets work in finite and infinite space.

Chellali et al. takes the research into locating-dominating sets and then ponders the implications of a vertex being unable to dominate itself [3]. They begin to focus on finding the OLD set for finite graphs. After defining open-locating-dominating sets, they then define various theorems and propositions for those types of sets.

The authors show the existences of OLD sets within certain graph types and then define which graphs can have OLD sets and which graphs cannot. Furthermore, it shows

that if a graph has a defined OLD, you can add vertices to those graphs and still maintain the same OLD. Figure 5 shows these properties of closed grids having OLD sets. Finally, the paper goes into infinite families of graphs that maintain the same OLD sets.

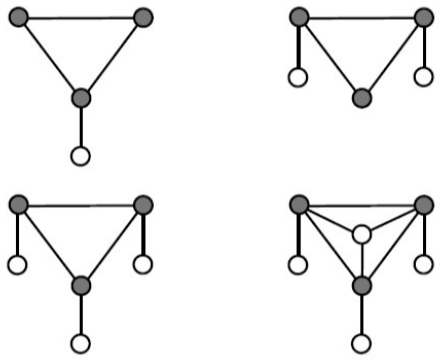


Figure 5: Open-locating-dominating sets on closed graphs [3]

d. Open-locating-dominating sets in infinite graphs

Seo and Slater also investigated how open-locating-dominating sets work in infinite graphs [13]. After proving various theorems and propositions for various finite graphs, they move on to finding OLD sets for infinite states, introducing the concept of OLD% as the minimum density of vertices required for a given infinite grid. Seo and Slater showed that the OLD% is 2/5 and 1/2 for the square grid and the hexagonal grid respectively. For the triangular grid, they showed that OLD% is at most 1/3, but later Kincaid et al. proved that the OLD% is exactly 4/13 [9].

Theorem 4 [8, 11]: The minimum density for an open-locating-dominating set for the infinite square, the hexagonal, and the triangular grid are 2/5, 1/2, and 4/13 respectively.

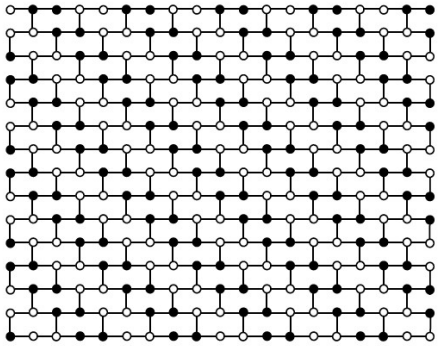


Figure 6: A minimum density OLD set of 1/2 for the infinite hexagonal graph [13]

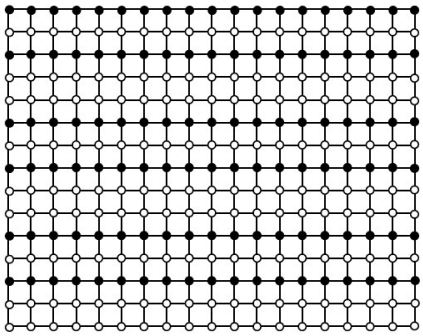


Figure 7: A minimum density OLD set of 2/5 for the infinite square graph [13]

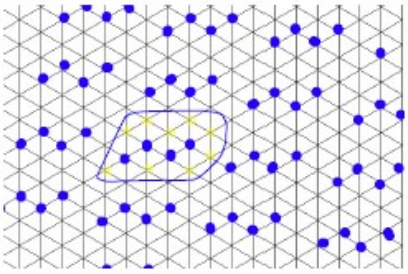


Figure 8: A minimum density OLD set of 4/13 for the infinite triangular graph [9]

Seo and Slater also introduces the concept of *open-independent open-locating-dominating* (OLD_OIND) sets for infinite grids, which expands upon the concepts of OLD% in their previous work [10]. An *independent set* I is a subset of vertices where no

vertex pair in I forms an edge. Since an OLD-set cannot be an independent set, because it is impossible for a vertex to be by itself inside of an OLD set, the concept of *open-independence* in OLD sets stands for two vertices that are connected to each other by an edge but have no other edges to other vertices.

For the infinite square grid, Seo and Slater showed that the OLD_OIND% is 3/7. For the infinite hexagonal grid, they concluded that the OLD_OIND% is 1/2. However, for the OLD_OIND%, Seo and Slater were able to show at best that the OLD_OIND% is less than or equal to 8/25.

Theorem 5 [11]: The minimum density for an open-independent open-locating-dominating set for the infinite square, hexagonal, and triangular grid are 3/7, 1/2, and less than or equal to 8/25 respectively

e. Open-locating-dominating sets in the infinite king's graph

Seo looked into OLD sets in the infinite king's graph [12]. It was discovered that there were multiple graphs showcasing an OLD set with a density of 1/4, as shown in Figures 9, 10, and 11. Using "open-share" arguments, which determines just how much each vertex dominates the tile of the graph without dominating itself, Seo was able to find the theoretical lower bound for OLD sets in the infinite king's graph as 6/25. This number is slightly less than the 1/4 that Seo discovered in her graphs. This means that there is a slight gap between the theoretical and the discovered, which is where our

program will come into play. We will attempt to close this gap with our program. Figures 9, 10, and 11 show the known minimum density OLD sets for the infinite king's graph.

Theorem 6 [12]: The minimum density for an open-locating-dominating set for the infinite king's grid is between 1/4 and 6/25 inclusive.

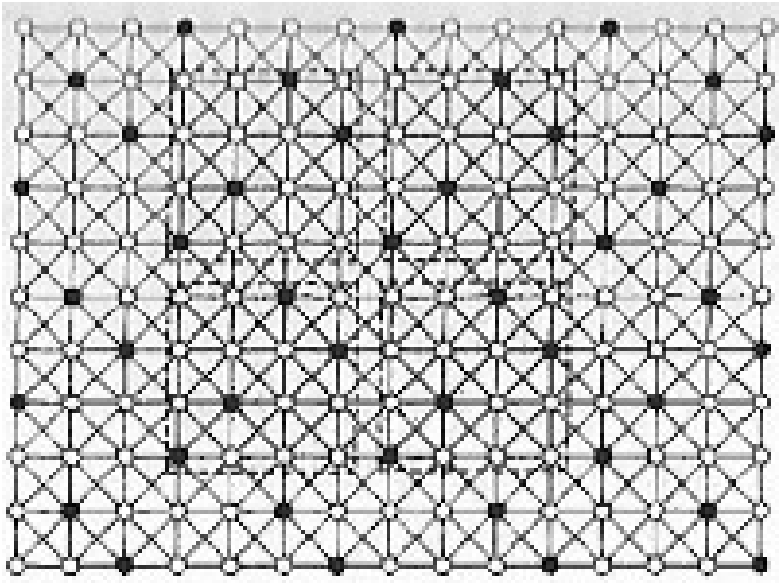


Figure 9: A minimum density OLD set of 1/4 for the infinite king's graph [12]

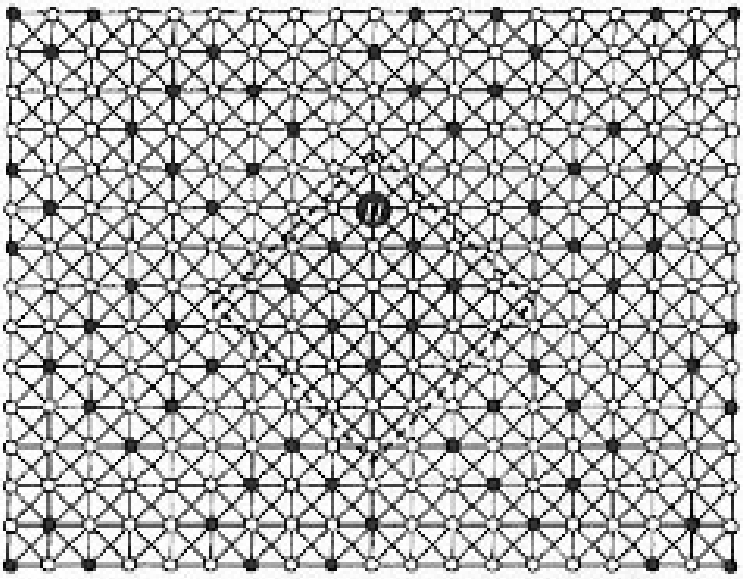


Figure 10: A minimum density OLD set of 1/4 for the infinite king's graph [12]

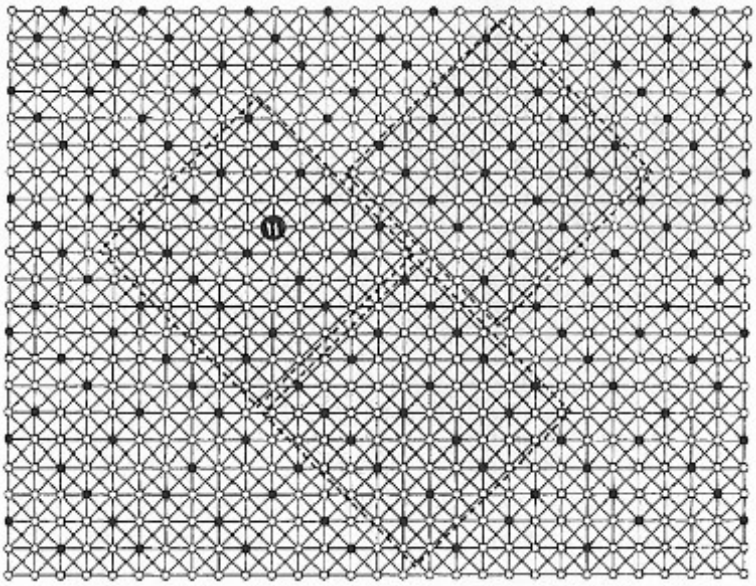


Figure 11: A minimum density OLD set of 1/4 for the infinite king's graph [12]

f. Fault-tolerant open-locating-dominating sets

Fault-tolerant OLD-sets are a type of OLD-set that looks to maintain their OLD properties despite a "fault" within the system. These types of faults could be when a sensor is not working or when a sensor is faulty and sending out a wrong signal [16]. There are different ways to create a fault-tolerant OLD set. The first is through what is called a *redundant OLD set*, which sets out to solve the issue if a sensor is simply not sending any signal [11]. For this type of set, every vertex inside of the graph must be double dominated. Another type of OLD set is the *detector OLD set*, which deals with a faulty sensor falsely sending wrong signal. Figure 12 shows the difference between an OLD set and a fault-tolerant redundant OLD set.

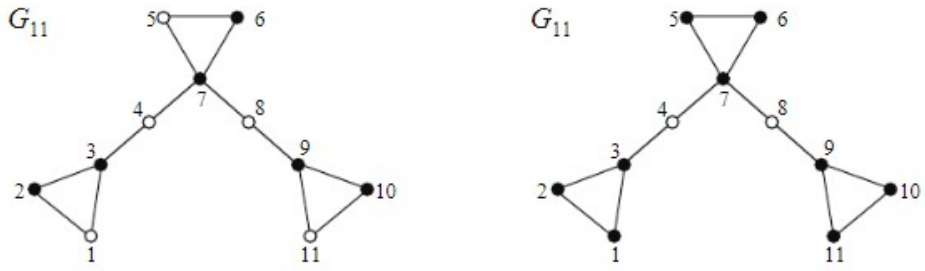


Figure 12: An open-locating-dominating set and a fault-tolerant open-locating-dominating set for G_{11} [11]

g. NP-completeness

NP, which stands for *nondeterministic polynomial time*, describes how long a problem takes to solve. Problems that can be solved "quickly" or in *deterministic time* are considered solvable in polynomial time, or P time. Problems that take a

nondeterministic time to solve are NP problems. There is a special type of problem called *NP-complete*. These problems can have their solutions verified in P time but haven't been proved to solve in P time.

A subclass of NP, the class NP-complete, consists of the hardest problems in the class NP. They have the various characteristics that places these problems within this subclass. These problems have no known polynomial algorithms to solve the problem. There is no proof that the problem cannot be solved by a polynomial time algorithm, but it is widely conjectured that such algorithms do not exist. The study of NP-completeness is very important because if we can find a polynomial time solution to a NP-complete problem, then we can find a polynomial time solution to all NP-complete problems.

In order to determine if a problem A is NP-complete, we need to a known NP-complete problem B, such as the satisfiability problem, to problem A in polynomial time. The satisfiability problem focuses on whether a given statement, which is *conjunction of clauses*, can be evaluated to true or false. Each clause is defined to be a *disjunction of literals*, or *variables*. Each literal can be true or false. In other words, for the satisfiability problem, we are interested in determining whether there is a truth value assignment for each literal which evaluates a conjunction of clauses true. Reducing the satisfiability problem to a given, or new, problem is a sure way to prove the problem is NP-complete.

h. NP-completeness of OLD set problems

Seo and Slater proved that problem of determining the smallest size of an OLD set for an arbitrary graph is an NP-complete problem [14]. They did so by reducing the *3-SAT problem* into the problem OLD in polynomial time. The 3-SAT problem has been proved to be an NP-complete problem. As state above, if a known NP-complete problem can be reduced to a new problem in polynomial time, then the problem is considered to be NP-complete. In the 3-SAT problem, each disjunction, or clause, consists of exactly three literals. The following definitions and the theorem are given in Seo and Slater [14].

3-SAT

INSTANCE: Sets $C = \{c_1, c_2, \ldots, c_M\}$ of clauses on set $U = \{u_1, u_2, \ldots, u_N\}$ such that $|c_i| = 3$ for $1 \leq i \leq M$.

QUESTION: Is there a satisfying truth assignment for C?

Redundant-OLD Set

INSTANCE: Graph $G = (V, E)$ and positive integer $K \leq |V|$

QUESTION: Is $OLD(G) \leq K$

Theorem 7 [12]: Problem OLD is NP-complete.

Proof. First of all, problem OLD belongs to class NP because it can be solved by a non-deterministic polynomial time algorithm and a candidate solution can be verified in a deterministic polynomial time. In order to show problem OLD is NP-complete, we need to show a known NP-complete problem can be reduced to problem OLD in polynomial time and we will do this from the 3-SAT problem.

Let C and U be the set of clauses and the set of variables, respectively. We construct the graph G_i on 21 vertices for each u_i in U and the graph H_i for each clause c_i in C on 7 vertices as shown in Figure 13. Next, we connect each c_i to the three literal vertices. We observe that the graph G constructed in this fashion has 21N + 7M vertices and 27N + 10M edges, where N and M are the number of variables and clauses, respectively. This graphs construction can be done in polynomial time. Now we will show that OLD(*G*) = (10N + 3M) if and only if C has a satisfying truth assignment.

Let D be on an OLD(G) set. Then D must contain the darkened vertices in Figure 13. Furthermore, because each v_i vertex has to be dominated by either u_i or u_i', D must contain exactly one vertex from $\{u_i, u_i'\}$ for each G_i and |D| = (9N + 3M) + N = (10N + 3M). Because D is an OLD(G) set, each vertex c_i has to be dominated by at least one of the three literal vertices. We can see that the set of literal vertices that belong to D form a satisfying truth assignment for C.

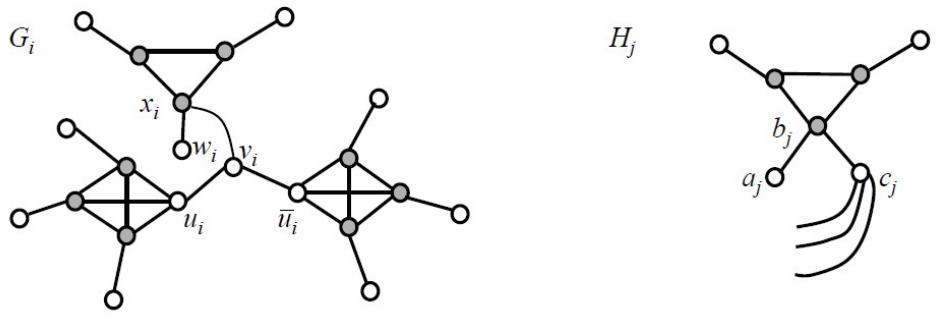


Figure 13: A graph proving that OLD set problem is NP-complete [14]

Now suppose there is a satisfying truth assignment for each c_i. We first form a set D containing 9 vertices from each variable graph G_i and 3 vertices from each clause graph H_i. Next, for each variable graph G_i, we add to D the vertex u_i if it is true. Otherwise, we add the vertex u_i'. The resulting set D will contain $(9N + 3M) + N = (10N + 3M)$ vertices and each v_i will be dominated by one vertex in D and each vertex c_j will be dominated by at least one literal vertex, whichever one is assigned true.

We see that we have OLD (G) = (10N + 3M) if and only if C has a satisfying truth assignment. Since the reduction from the 3-SAT problem to OLD problem can be done in polynomial time, we conclude that OLD problem is NP-complete.

Seo and Slater also proved that problem OLD_OIND is an NP-complete problem by reducing the 3-SAT problem to the OLD_OIND problem in polynomial time [13]. As described above, each component in an OLD_OIND set of a graph forms a pair of vertices connected to each other. The following definitions and the theorem are given in Seo and Slater [13]. We present our own proof for the theorem based on results [13].

XOIOLD (existence of an open-independent, open-locating-dominating set)

INSTANCE: A graph G.

QUESTION: Does G have an OLDOIND-set?

Theorem 8 [11]: Problem OLD_OIND is NP-complete.

Proof. First, problem OLD_OIND belongs to class NP because it can be solved by a non-deterministic polynomial time algorithm and a candidate solution can be verified in

deterministic polynomial time. We show problem OLD_OIND is NP-complete by reducing from the 3-SAT problem.

Given a collection of clauses C and a collection of literals U, we can create a graph of six vertices for each u_i in U and construct a graph of three vertices for each c_j in C as shown in Figure 14. Then, we connect each c_j to three literal vertices accordingly.

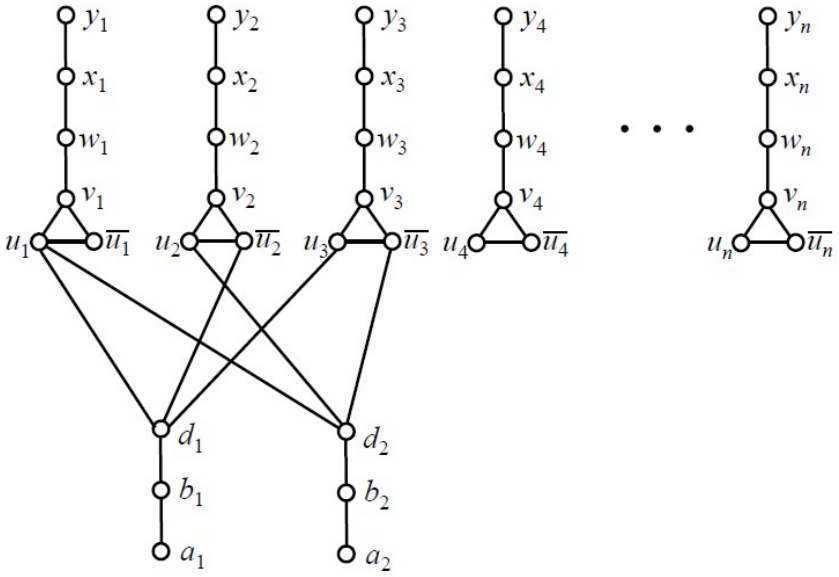


Figure 14: A graph proving that OLD_OIND set problem is NP-complete [13]

Suppose there is a satisfying truth assignment of C. We first form a set D containing vertices x_i, y_i, v_i for each u_i, and vertices a_j and b_j for each c_j. Now for each literal, we add the vertex u_i if it is true. Otherwise, we add the vertex u_i'. Now, each d_i vertex will be dominated by at least one other vertex from the set of three literal vertices

because there is a satisfying truth assignment of C. We see that D is an OLD_OIND set of G.

Now suppose there is an OLD_OIND set D for G. We must have $\{x_i, y_i\}$ be a subset of D and w_i not be an element in D for each literal. We also must have $\{a_j, b_j\}$ be a subset of D and d_j not be an element in D for each clause. We also see that we must have v_i be an element of D for each literal because w_i and y_i are otherwise not distinguished. Because we must have v_i be an element of D, the set D must hold either u_i or u_i', but not both. Since each vertex b_i dominates both a_i and d_i, in order to distinguish the two vertices a_i and d_i, each d_i must be dominated by at least one vertex from the set of three literal vertices. In other words, there is at least one literal which is assigned true, and hence there is a satisfying truth assignment for C.

We have shown the reduction of 3-SAT to the OLD_OIND problem and it can be done in polynomial time. We conclude that there is an OLD_OIND set for G if and only if there is a satisfying assignment of C.

CHAPTER III

OLD SETS IN THE INFINITE KING'S GRAPH

In this chapter, we will describe the algorithm for finding an OLD set for the infinite king's graph. We will discuss the algorithm in full detail and go over the steps the algorithm takes in finding an OLD set. Finally, we will discuss some of the issues pertaining towards the algorithm and future steps that could be taken.

a. Methods

As stated in earlier chapters, Seo [11] has found a few OLD sets with a 1/4 density for the infinite king's graph. This means that the OLD% for the infinite king's graph is at most 1/4. Seo has also shown that the theoretical minimum density is 6/25, which is just slightly under 1/4 [11]. Using a computer program, we attempted to find a minimum density open-locating-dominating set with a density of 6/25.

For our program, we accepted three inputs. The first two are the number of rows and columns in the tile of the graph. While the graph is infinite, the tiles of the graph are simply repeated infinitely over the graph, so we need the dimensions of the tile. The third input is the number of selected vertices in the tile and this number represents the size of an OLD set. For example, if the user inputs 4 6 5, then the program will try to find OLD sets with the cardinality of 5 for a 4 by 6 tile in the infinite king's graph.

Using the number of vertices given by the user, the program goes through all possible combinations of vertices and test whether the set is an OLD set. We create a matrix using the first set of vertices, and if that matrix does not create an OLD set, we

will iterate the last vertex. Once that last vertex has been iterated completely, then we will backtrack to the second to last vertex and iterate that vertex. In this fashion, the program tries every possible vertex combination within a tile and tests whether that combination constitutes an OLD set. Obviously, if we try all possibilities, the solution space will be huge and the program becomes intractable for the input of a large size. For example, if the three inputs are 4 4 4, then the program needs to check 43,680 cases. Instead of blindly checking all possibilities, our program utilizes two promising heuristics.

Before we test if the set is an OLD set, we first determine if a set of vertices is promising. We do this by checking to see if there is a singleton vertex within the set. If there is a singleton vertex within the set that is not connected to any other vertices, then there is no possible way for the set to create an OLD set. Then, we check to see if there is a vertex that has two private neighbors and no other neighbors. A private neighbor is a vertex that is connected to one other vertex. If a vertex has two private neighbors, then there is no way for that set to be able to create an OLD set simply because those two private neighbors are not distinguished.

The program then goes through two phases. The first is to check the property of open-domination and the second is the property of locating or distinguishing. If the set of vertices is open-dominating, then we check to see if the set of vertices is open-dominating, and thus, using the laws of symmetry, of the entire graph. We do this by creating a second matrix of the same size that is initialized with all 0s. Using the first matrix, we find every vertex within that matrix. We then fill the second matrix with a

series of 1s in every box around the vertex in the first matrix. If every element in the second matrix is a 1, then the set is an open-dominating set. Figure 15 shows a set that is an open-dominating set and the matrix it produces. Figure 16 shows the pseudocode for the check for open-domination.

0 0 1 0
0 0 0 1
0 1 0 0
1 0 0 0

1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1

Set is an open-dominating set.

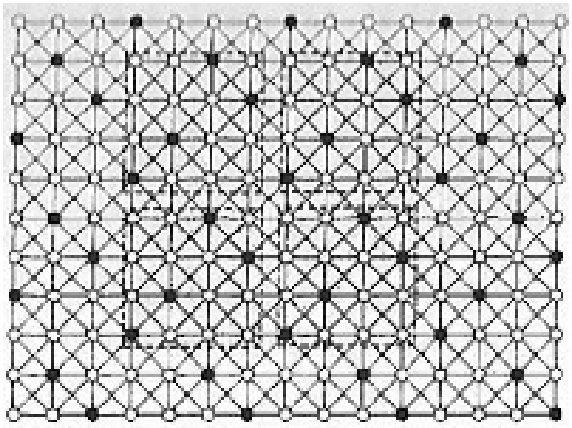


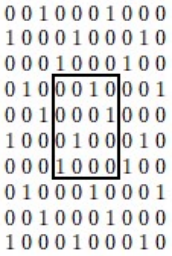
Figure 15: A test of an OLD set for open-dominating and the graphical representation of the matrix [12]

Algorithm 1: Open Dominating Property Check

```
create a new tile of the same size;
for each vertex in the set do
    set a 1 for each neighboring vertex;
if every vertex is a 1 then
    return true;
else
    return false;
```

Figure 16: Pseudocode for open dominating property check

Secondly, we determine if the set has locating or distinguishing properties. In order to determine this, we need to expand the matrix we created by three in each direction. The reason for this expansion is in order to ensure that the vertices are the edges can be located properly. Figure 16 shows the expanded matrix for the open-dominating set shown in Figure 15. Using this newly expanded matrix, we can now find the locating codes for all the vertices in this tile. A locating code is every vertex that is within the set that is around the vertex in question. We do this for every vertex in the tile, whether it is a part of the set or not. We store all these locating codes within an array of vectors. Figure 17 shows the pseudocode for the distinguishing check for these vertices.



1,1	0,2 1,0	3,1	4,2	5,1	4,2 5,0	7,1	8,2
1,2	0,2 2,3	3,2	2,3 3,1 4,2	5,2	4,2 6,3	7,2	6,3 7,1 8,2
1,3	0,2 1,4 2,3	3,3	2,3 4,2	5,3	4,2 5,4 6,3	7,3	6,3 8,2
1,4	2,3	3,4	2,3 3,5	5,4	6,3	7,4	6,3 7,5
1,5	0,6 1,4	3,5	4,6	5,5	4,6 5,4	7,5	8,6
1,6	0,6 2,7	3,6	2,7 3,5 4,6	5,6	4,6 6,7	7,6	6,7 7,5 8,6
1,7	0,6 1,8 2,7	3,7	2,7 4,6	5,7	4,6 5,8 6,7	7,7	6,7 8,6
1,8	2,7	3,8	2,7 3,9	5,8	6,7	7,8	6,7 7,9
2,1	1,0 3,1	4,1	3,1 4,2 5,0	6,1	5,0 7,1	8,1	7,1 8,2 9,0
2,2	2,3 3,1	4,2	3,1	6,2	6,3 7,1	8,2	7,1
2,3	1,4	4,3	4,2 5,4	6,3	5,4	8,3	8,2 9,4
2,4	1,4 2,3 3,5	4,4	3,5 5,4	6,4	5,4 6,3 7,5	8,4	7,5 9,4
2,5	1,4 3,5	4,5	3,5 4,6 5,4	6,5	5,4 7,5	8,5	7,5 8,6 9,4
2,6	2,7 3,5	4,6	3,5	6,6	6,7 7,5	8,6	7,5
2,7	1,8	4,7	4,6 5,8	6,7	5,8	8,7	8,6 9,8
2,8	1,8 2,7 3,9	4,8	3,9 5,8	6,8	5,8 6,7 7,9	8,8	7,9 9,8

Figure 17: A test for locating properties of an OLD set with all the locating codes

Algorithm 2: Distinguishing Property Check

```
for each vertex in the tile do
    find all neighboring vertices in set;
    check to see if neighbor list is unique;
    if not unique then
        return false;
return true;
```

Figure 18: Pseudocode for distinguishing property check

Once we have all the identifying codes in this array, we need to perform a check on each element of the array. That check is to ensure that the identifying code is unique. We check this vector against every vector below it to ensure that it does not match another vector. If both checks are okay for every element of the vector, we then can declare that these vertices are a distinguishing set for this tile.

Since the complexity of this algorithm is exponential in the worst case, we have parallelized the algorithm using a server cluster utilizing a protocol known as MPI. MPI is a protocol that allows commands, or "messages" to be passed from one server to another. This way, we can leverage multiple servers while reducing the time it takes to backtrack through a very large amount of data. In this server cluster, the first server, server 0, acts as the master, sending information to all the worker servers. The master server handles the selecting a candidate OLD set, which is a set of vertices, while the

worker servers handle checking to see if the set of vertices is an OLD set. Figure 19 shows the pseudocode for the MPI structure.

Algorithm 3: MPI Structure

```
if master rank then
    while backtracking through vertices do
        if vertices are promising then
            send vertices to servant rank;
            wait to receive results from servant rank;
            if OLD set found then
                record results;
else
    wait for master rank to send vertices;
    receive vertices from master rank;
    if vertices have Open Dominating Property then
        if vertices have Distinguishing Property then
            return true;
        else
            return false;
    else
        return false;
```

Figure 19: Pseudocode for MPI structure

b. Results

Since we are looking for minimum densities of 1/4 and 6/25, there are certain configurations we want to look at. When running our program on the server cluster, for a tile that is size 4 x 4 with a density of 1/4, it took 0.34 seconds to check every single possible vertex configuration. We did find an OLD configuration, but it was a

configuration that was already found and confirmed previously by Seo [11]. No other OLD configurations were found for these sets of data.

While running our program on a 5 x 5 tile with a density of 6/25, it took 43.7 seconds to check every single possible vertex configuration. We did not find any OLD configurations for this set of input values. This means that if there is a set with a minimum density of 6/25, it would have to be in a much larger tile size.

Finally, when running our program on a 6 x 6 tile with a density of 1/4, it took over an hour and we were unable to continue running the program on the server cluster that we were using due to time limit constraints. Unfortunately, we were unable to go through the entire 6 x 6 tile set within a reasonable time frame, nor were we able to find any OLD sets during the time frame. There were other tile size configurations that we did check, such as a 4 x 16 tile set, but again, due to the size of the tile set and the number of vertices required to reach the desired minimum densities, we were unable to get through the all the possible vertex combinations.

c. Discussion

We were able to confirm that we could find an OLD set using our algorithm. We were also able to confirm that there is only one possible configuration of an OLD set for a 4 x 4 tile set with 4 vertices. All other possible configurations are symmetric to that configuration. We were also able to confirm that there are no possible configurations for an OLD set for a 5 x 5 tile set with 6 vertices. Unfortunately, the backtracking approach

could only handle so much, as the larger tile sets with more vertices were too large for the methodology.

The reason why the program worked for a 4 x 4 tile set with 4 vertices compared to a 6 x 6 tile set with 9 vertices is simply the number of points the algorithm would have to go through. A 4 x 4 tile with 4 vertices has 43,680 (${}_{16}P_4$) possible permutations to go through, which is easily done with our program. However, in comparison to the 6 x 6 tile with 9 vertices, the number of permutations the algorithm would have to go through is 34,162,713,446,400 (${}_{36}P_6$). There is a steep gap between the two tile sets that our promising heuristic and our parallelization could not bridge.

In order to find OLD sets on much larger tile sets, we would have to change the approach of how vertices are generated. The first possible method would be to create a better promising heuristic that would check to see if a vertex set is promising before checking to see if it is indeed an OLD set. One such possible better promising heuristic may involve vertex shares. The theory of vertex shares is that vertices take up a certain amount of area around them and share that area with other vertices. The trick would be to maximize the vertex share between vertices in order to find the optimal spacing between vertices. This theory of vertex shares was what helped create the theoretical value for the minimum density for OLD sets in the infinite king's graph [11]

The second possible method would be to start off with a random set of vertices and then move vertices around based on how close the set is to an OLD set. Again, vertex shares would help to move these vertices around to the best configuration. In this

method, we would find which vertices are violating the OLD set principles and then move them based on the optimal vertex shares.

CHAPTER IV

OLD SETS IN A FINITE GRAPH

In this section, we will go over an algorithm to find OLD sets for a finite graph. We will look at the methods the algorithm uses to find these OLD sets. We will also look at some of the complications that have arisen while searching for OLD sets on finite graphs.

a. Methods

To develop an algorithm for finite graphs, we first took our algorithm from our infinite king's graph. However, we had to modify this algorithm to work for a closed, finite graph. First, we had to accept any type of closed finite graph. Secondly, we had to use such a graph to check and verify if the set is indeed an OLD set on that graph.

Our program first accepts two variables. These variables are the name of a file and the number of vertices in the graph. The data file contains an adjacency list of edges, which is a commonly used representation of a graph. The program takes this list of edges and then creates a series of C++ vectors to represent the graph. Figure 17 shows a Petersen graph representation and the input file list for the graph.

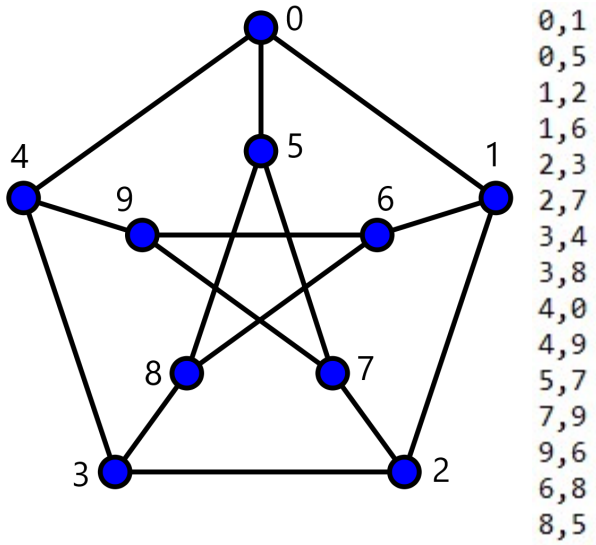


Figure 20: A Petersen graph and its representation in a file [18]

The program once again utilizes the backtracking methodology to generate all the possible vertices that exist in the graph. We leverage the same MPI interface that we used for the previous algorithm to once again help with some of the load of backtracking through all the possibilities. The same verification methods discussed in the previous chapter exist here, except instead of checking for every vertex that surrounds a vertex in the infinite king's graph, instead, the program will check every edge that connects the vertex that we are currently verifying.

Once we have verified if a set is an OLD set, we report the result and create an output message detailing what set of vertices is an OLD set. Once we have gone through every possible vertex combination, we output the final results of how many sets we have found in total.

b. Results

Utilizing our algorithm, we were able to go through many different types of graphs, such as the Petersen graph and the cubic graph, and find OLD sets for those graphs. The algorithm seems to perform very rapidly, given that the number of vertices is not too large and the closed graphs are not too overwhelming.

```
mpiexec -f mpihosts -n 12 ./a.out petersen.dat 5
Open-locating-dominating set found!
pointItems contains:  1 0 2 4 3
Open-locating-dominating set found!
pointItems contains:  1 0 7 2 5
Open-locating-dominating set found!
pointItems contains:  1 0 4 6 9
Open-locating-dominating set found!
pointItems contains:  1 8 0 5 6
Open-locating-dominating set found!
pointItems contains:  0 3 4 5 8
Open-locating-dominating set found!
pointItems contains:  0 7 4 5 9
Open-locating-dominating set found!
pointItems contains:  1 8 2 3 6
Open-locating-dominating set found!
pointItems contains:  1 2 9 6 7
Open-locating-dominating set found!
pointItems contains:  2 9 3 4 7
Open-locating-dominating set found!
pointItems contains:  2 3 5 7 8
Open-locating-dominating set found!
pointItems contains:  3 4 6 8 9
Open-locating-dominating set found!
pointItems contains:  5 6 7 8 9
Found 12 OLD sets.
Time taken: 0.06s
Program has reached the end.
Press ENTER to continue.
```

Figure 21: An OLD test of the Petersen graph for five vertices

The results above list all possible combinations for OLD sets in a Petersen graph for five vertices. We can see it took 0.06 seconds to run when utilizing 12 MPI servers. It found 12 results and listed all the possible results.

c. Discussion

As long as the closed graphs are not too large, then the program is able to easily backtrack through all the vertices and find every possible OLD set that exists within that graph for that number of vertices. However, once again, we are limited by size. In the infinite king's graph, every vertex has eight edges. It is possible in these closed graphs that there are even more edges connecting to the vertices.

Another factor that could hamper this program is the number of vertices. Backtracking through all of the vertices is a very costly and timely algorithm. However, unlike the previous problem, where we were trying to the minimum density, this algorithm is attempting to find every possible set. This means we would have to check every possible vertex, since we are not satisfied with merely one answer. Therefore, using something like vertex shares to modify the program may help in terms of the promising algorithm, but would not help in any other instance.

CHAPTER V

NP-COMPLETENESS OF FAULT-TOLERANT OLD SETS

In this chapter, a proof of the fault-tolerant OLD set problems is NP-complete is shown. As introduced by Seo and Slater [11], we say that an OLD-set S is a redundant OLD set (RED:OLD set) if, for each v that is an element of S, the set S - {v} is an OLD set, and we let RED:OLD(G) denote the minimum cardinality of a RED:OLD set for G. A redundant OLD set S of G with RED:OLD(G) = |S| is called a RED:OLD(G) set. We will show the proof using the reduction from the 3-SAT problem. Proving the RED:OLD set problem is NP-complete will be a significant contribution to the NP-Complete community and to the Graph Theory community.

3-SAT

INSTANCE: Sets $C = \{c_1, c_2, \ldots, c_M\}$ of clauses on set $U = \{u_1, u_2, \ldots, u_N\}$ such that $|c_i| = 3$ for $1 \leq i \leq M$.

QUESTION: Is there a satisfying truth assignment for C?

Redundant-OLD Set

INSTANCE: Graph $G = (V, E)$.

QUESTION: Does G have a RED:OLD set with RED:OLD(G) ≤ K?

Theorem 8: Problem RED:OLD set is NP-complete.

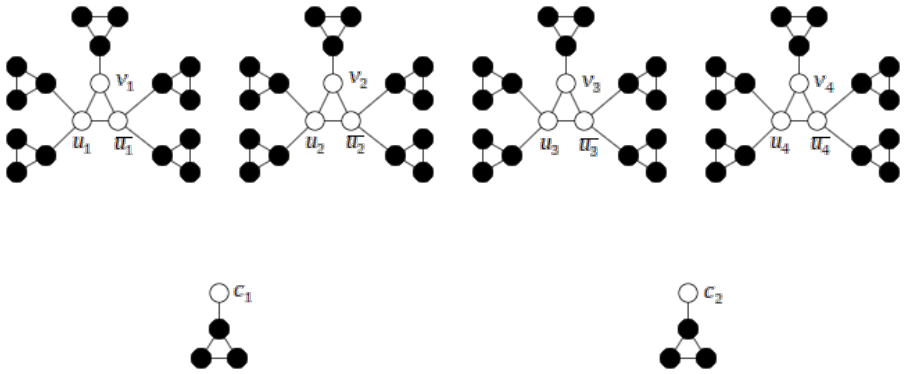


Figure 22: Variable and clause graphs

Proof. First, Problem RED:OLD belongs to class NP because it can be solved by a non-deterministic polynomial time algorithm and a candidate solution can be verified in deterministic polynomial time. We will show a polynomial time reduction from 3-SAT to RED:OLD. Given U and C, for each u_i, construct the graph G_i on 18 vertices as shown in Figure 18. For each clause c_j, construct the graph H_j on 4 vertices, also shown in Figure 18. To complete the creation of the graph G, for $1 \leq i \leq M$ if clause Cj = $\{u_{j,1}, u_{j,2}, u_{j,3}\}$, let clause vertex c_j be adjacent to variable vertices $u_{j,1}$, $u_{j,2}$, and $u_{j,3}$. We see that G has 18N + 4M vertices and can be constructed from C in polynomial time.

Assume that S, a subset of V(G), is a RED:OLD(G)-set. Then S is required to contain every darkened vertex in Figure 18. In addition, in order for each v_i to be double dominated, either u_i or u_i' must be a member of the set S. Now S contains 16N+3M vertices. Since S is a RED:OLD set, each c_i must be dominated by at least one vertex from the set of three literal vertices. This implies there is a satisfying truth assignment for clause c_i.

Suppose there is a satisfying truth assignment of C. We first form a set S containing all of the darkened vertices in Figure 18. Now for each literal, we add to S u_i if u_i is true. Otherwise, we add u_i'. We observe that each c_i vertex will be dominated by at least one other vertex from the set of three literal vertices because there is a satisfying truth assignment of C. We see that S is a RED:OLD set of G containing 16N+3M vertices.

We have shown the reduction of 3-SAT to the RED:OLD problem and it can be done in polynomial time. We conclude that there is a RED:OLD set for G with 16N+3M vertices if and only if there is a satisfying assignment of C.

CHAPTER VI

CONCLUSION AND FUTURE WORKS

In this chapter, we discuss the future directions of the research on the topic of OLD sets. We recognize that there is plenty of work to do in the future stemming from this research.

One of the aims of this research was to find the minimum density OLD set for the infinite king's graph. We have made significant progress toward the problem by developing a program, running it in parallel, and creating a promising heuristic for the algorithm. However, as we have seen, we were unable to narrow the gap between 1/4 and 6/25 for the OLD set in the infinite king's graph. While we were able to eliminate some possibilities, we were unable to pinpoint whether a graph exists on 6/25 or if 1/4 is truly the minimum.

A couple ways we could find this answer is through improving our algorithm. We can improve our algorithm by implementing a better promising heuristic that involves vertex shares. We could also improve the way we generate vertices by using vertex shares to find optimal vertex placements within the graph itself. We plan to continue to work on finding the answer in a closed form.

For many classes of graphs, the minimum cardinality of OLD sets or the upper and lower bounds on it have been found. However, there has not been much work done on finding an algorithm to determine an OLD set for a given graph, including trees, which is considered to be a relatively simple class of graphs. If we use a brute-force approach, we may be able to find a solution of the OLD set problem for a graph with a

relatively small number of vertices. However, when the number of vertices of a graph becomes too large, the complexity of such an algorithm would become prohibitive, and hence the problem would turn out to be intractable. We have implemented a simple algorithm and tested on several graphs. Our program has not been tested on a graph with an arbitrary number of vertices and edges and we plan to improve our program so that it can determine an OLD set efficiently for an arbitrary graph or for some special classes of graphs.

As discussed in earlier chapters, there are three types of fault-tolerant OLD sets. We have proved that the simplest type of these three, the RED:OLD set problem is NP-complete. RED:OLD set problem is also called redundant-distinguishing OLD set problem. The other two types are called detection-distinguishing OLD sets and error-correcting OLD sets. We conjecture that the problems of finding the minimum size of these two fault-tolerant sets belong to the NP-complete class, but there are no known proofs for these two problems. Creating a proof for the two problems would be considered a significant contribution to the research communities of Theory of NP as well as to the research community of Graph Theory.

Since the OLD set problem was introduced in 2010 [14], many researchers have discovered solutions for various classes of graphs. However, there are still other classes of graphs to look at. The fault-tolerant OLD set problem is a relatively new topic, which was introduced in 2014 [11]. Plenty of work could be done with fault-tolerant OLD sets, which include finding solutions in infinite graphs.

BIBLIOGRAPHY

[1] Charon I, Honkala I, Hudry O, Lobstein A. General bounds for identifying codes in some infinite regular graphs. The Electronic Journal of Combinatorics. 2001 Nov 14:R39-.

[2] Charon I, Honkala I, Hudry O, Lobstein A. The minimum density of an identifying code in the king lattice. Discrete Mathematics. 2004 Feb 6;276(1-3):95-109.

[3] Chellali, M., Rad, N. J., Seo, S. J., and Slater, P. J. On open neighborhood locating dominating in graphs. Electronic Journal of Graph Theory and Applications (EJGTA), 2(2):87–98, 2014.

[4] Garey M.R. and Johnson D.S.. Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman. 1979.

[5] Honkala, I. An optimal locating-dominating set in the infinite triangular grid. Discrete Mathematics, 306(21):2670–2681, 2006.

[6] Honkala, I. and Laihonen, T. On locating–dominating sets in infinite grids. European Journal of Combinatorics, 27(2):218–227, 2006.

[7] Inpoonjai, P. and Jiarasuksakun, T. Dominating and locating sets in the multiplication of a graph. Indian Journal of Science and Technology, 8.

[8] Karpovsky MG, Chakrabarty K, Levitin LB. On a new class of codes for identifying vertices in graphs. IEEE Transactions on Information Theory. 1998 Mar:44(2):599-611.

[9] Kincaid, R., Oldham, A., and Yu, G. Optimal open-locating-dominating sets in infinite triangular grids. Discrete Applied Mathematics, 193:139–144, 2015.

[10] Lobstein A. Watching systems, identifying, locating-dominating, and discriminating codes in graphs. https://www.lri.fr/~lobstein/debutBIBidetlocdom.pdf

[11] Seo, Suk J. and Slater, P. J. Fault Tolerant Detectors for Distinguishing Sets in Graphs. Discussiones Mathematicae Graph Theory. 35. 10.7151/dmgt.1838, 2015.

[12] Seo, S. J. Open-locating-dominating sets in the infinite king grid. Journal of Combinatorial Mathematics and Combinatorial Computing, 104:31–47, 2018.

[13] Seo, S. J. and Slater, P. J. Open-independent, open-locating-dominating sets. Electronic Journal of Graph Theory and Applications, 5(2):179–193, 2017.

[14] Seo, S. J. and Slater, P. J. Open neighborhood locating dominating sets. Australasian J. Combinatorics, 46:109–120, 2010.

[15] Slater, P. J. Domination and location in acyclic graphs. Networks, 17(1):55–64, 1987.

[16] Slater, P. J. Fault-tolerant locating-dominating sets. Discrete Mathematics, 249(1- 3):179–189, 2002.

[17] Wikipedia contributors,. King's graph — Wikipedia, the free encyclopedia, 2019. [Online; accessed 9-April-2019].

[18] Wikipedia contributors,. Petersen graph — Wikipedia, the free encyclopedia, 2020. [Online; accessed 2-February-2020].

APPENDICES

APPENDIX A

SOURCE CODE FOR OLD SETS IN INFINITE KING'S GRAPH

OLDSetsKings.h

```
using namespace std;

#pragma once
#include <iostream>
#include <fstream>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <unordered_set>
#include <set>
#include <vector>
#include <algorithm>
#include "mpi.h"

struct a_hash {

inline size_t operator()(const pair<int, int> & v) const {
      return v.first * 31 + v.second;
}
};

struct customCompare {

    bool operator()(const pair<int, int>& lhs, const pair<int,
int>& rhs) const {

        if (get<0>(lhs) == get<0>(rhs))
            return get<1>(lhs) < get<1>(rhs);
        else
            return get<0>(lhs) < get<0>(rhs);
    }
};

// matrix is the current matrix of the thread, domMatrix is used
by isOpenDominating, locMatrix
// is used by isLocating, OLDMatrix2 is used in
isOpenLocatingDominating
int** matrix;
int** domMatrix;
int** locMatrix;
int** OLDMatrix2;

// pointArray is used to send the points from rank 0 to the other
ranks, rows/cols/points
// are the inputs, pointItems are the points in set format, sets
are used to ensure
```

```
// that the points are not duplicated, locate is used by
isLocating
int* pointArray;
int* loadPointArray;
int loadCount;
int rows, cols, points;
int willLoad;
set<pair<int, int>, customCompare> pointItems;
vector<pair<int, int>> *locate;

// printEverything is an old  oolean used to print stuff,
isWorking is used to tell all
// the ranks when to stop
bool printEverything;
bool isWorking;

// numRanks is the number of ranks used, curRank is the current
rank of the server, status
// and request are used in sends and receives
int numRanks;
int curRank;
MPI_Status status;
MPI_Request request;

// Counters used to count various things, like number of OLD sets
found
int counter;
int OLDCounter;
int promisingCounter;

void createMatrices();

// Intializes everything that needs to be initialized
void initialize();

// Prints the matrix that is sent to it, used for debugging
void printMatrix (int** mat);

// Prints the locMatrix, used for debugging
void printLocMatrix (int** mat);

// Prints the point items in the set, used for debugging
void printPointItems (set<pair<int, int>, customCompare>
pointItems2);

// Fixes the row, ensures that rows stay in scope
int fixRow (int num);

// Fixes the col, ensures that cols stay in scope
int fixCol (int num);

// The main driver function for rank zero
void backtrack (int count, pair<int, int> start);

// The main driver function for non zero ranks
```

```
bool isOpenLocatingDominating();

// Checks to see if a matrix is open dominating
bool isOpenDominating (set<pair<int, int>, customCompare>
pointItems2);

// Checks to see if a matrix is locating
bool isLocating (int** matrix2);

// Checks to see if a matrix is promising before doing
calculations
bool isPromising();

// Adds a point to a matrix
void addPoint (pair<int, int> point, int** matrix2);

// Removes a point from a matrix
void removePoint (pair<int, int> point, int** matrix2);

// Gets the next point from the previous one, returns (-1, -1) if
that point is out
// of scope
pair<int, int> nextPoint (pair<int, int> point);

// Gets the previous point from the current point, returns (-1, -
1) if that point is
// out of scope
pair<int, int> previousPoint (pair<int, int> point);

// Gets the point that is X away from the point provided, returns
(-1, -1) if that
// point is out of scope
pair<int, int> nextXPoint (pair<int, int> point, int x);

// Changes a pair set of points to an array of points
void pairSetToArray (set<pair<int, int>, customCompare>
pointItems2);

// Changes an array of points back to a pair set of points,
returns a false
// if one of those points are negative
bool arrayToPairSet (int* pointArray2);

bool arrayToFile (int* pointArray2);

bool fileToArray (int* pointArray2);

// Checks to see if a point is next to another point
int findNeighbors (pair<int, int> point, set<pair<int, int>,
customCompare> pointItems2);
unordered_set<pair<int, int>, a_hash> getNeighbors (pair<int, int>
point, set<pair<int, int>, customCompare> pointItems2);
pair<int, int> checkNeighbor (pair<int, int> point, set<pair<int,
int>, customCompare> pointItems2, int zeroAdjust, int oneAdjust);
```

```
pair<int, int> checkPointInSet (pair<int, int> point,
set<pair<int, int>, customCompare> pointItems2);
```

OLDSetsKings.cpp

```
#include "OLDSetsKings.h"

int main(int argc, char** argv) {

// Alarm set so that I don't get yelled at, can adjust for need
alarm (3000);

// Initialize the mpi and get the ranks and current rank
MPI_Init (NULL, NULL);
MPI_Comm_size (MPI_COMM_WORLD, &numRanks);
MPI_Comm_rank (MPI_COMM_WORLD, &curRank);

printEverything = false;
isWorking = true;

rows = atoi (argv[1]);
cols = atoi (argv[2]);
points = atoi (argv[3]);
willLoad = atoi (argv[4]);

if (curRank == 0) {

      createMatrices();
      initialize();

      counter = 0;

      clock_t tStart = clock();

      if (willLoad == 1) {

            if (fileToArray(loadPointArray)) {

                  pair<int, int> nextPair = make_pair
(loadPointArray[0], loadPointArray[1]);
                  loadCount = 2;
                  backtrack (1, previousPoint (nextPair));

            } else {

                  initialize();
                  backtrack(1, make_pair(0, -1));
            }

      } else {

            // Call the backtrack function from rank 0
            backtrack(1, make_pair(0, -1));
      }
```

```
      // When the backtrack function is done, tell the other
ranks to stop working
      isWorking = false;
      for (int i = 0; i < numRanks - 1; i++) {

            MPI_Send (&isWorking, 1, MPI_INT, i + 1, 999,
MPI_COMM_WORLD);

      }

      cout << "Found " << OLDCounter << " OLD sets." << endl;
      printf("Time taken: %.2fs\n", (double)(clock() - tStart) /
CLOCKS_PER_SEC);

      cout << "Program has reached the end." << endl;
      cout << "Press ENTER to continue.";
      cin.ignore();
      cin.get();

      MPI_Abort (MPI_COMM_WORLD, -1);

} else {

      createMatrices();
      initialize();

      bool isOLD;

      // Set up the ability to receive isWorking whenever
      MPI_Irecv(&isWorking, 1, MPI_INT, 0, 999, MPI_COMM_WORLD,
&request);

      while (isWorking) {

            // Receive the point array from rank 0
            MPI_Recv (pointArray, points * 2, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

            // Turn that point array into a pair set
            bool isNeg = false;
            isNeg = arrayToPairSet (pointArray);


            // As long as there are no negatives in the pair set,
check to see if OLD
            // Otherwise, return false
            if (!isNeg && isPromising()) {

                  for (const pair<int, int>& curPoint :
pointItems) {

                        addPoint (curPoint, matrix);
                  }
                  isOLD = isOpenLocatingDominating();
```

```
            } else {

                  isOLD = false;
            }

            // Send the results back to rank 0
            int sendOLD = isOLD;
            MPI_Send (&sendOLD, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD);

            // Re-initalize and barrier
            initialize();
            MPI_Barrier (MPI_COMM_WORLD);
      }
}

return 0;
}

void createMatrices() {

matrix = new int*[rows];
for (int i = 0; i < rows; i++)
      matrix[i] = new int[cols];

OLDMatrix2 = new int*[rows];
for (int i = 0; i < rows; i++)
      OLDMatrix2[i] = new int[cols];

pointArray = new int [points * 2];
loadPointArray = new int [points * 2];
}

// Intializes everything that needs to be initialized
void initialize() {

for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++)
            matrix[i][j] = 0;

for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++)
            OLDMatrix2[i][j] = 0;

pointItems.clear();

for (int i = 0; i < points * 2; i++)
      pointArray[i] = 0;

for (int i = 0; i < points * 2; i++)
      loadPointArray[i] = 0;

OLDCounter = 0;
}
```

```
// Prints the matrix that is sent to it, used for debugging
void printMatrix (int** mat) {

cout << endl;
for (int i = 0; i < rows; i++) {

      for (int j = 0; j < cols; j++)
            cout << mat[i][j] << " ";
      cout << endl;
}
cout << endl;
}

// Prints the locMatrix, used for debugging
void printLocMatrix (int** mat) {

cout << endl;
for (int i = 0; i < rows + 6; i++) {

      for (int j = 0; j < cols + 6; j++)
            cout << mat[i][j] << " ";
      cout << endl;
}
cout << endl;
}

// Prints the point items in the set, used for debugging
void printPointItems (set<pair<int, int>, customCompare>
pointItems2) {

cout << "pointItems contains:  ";
for (const pair<int, int>& curPoint : pointItems2) {

      cout << " " << get<0>(curPoint) << "," << get<1>(curPoint);
}
cout << endl;
}

// Fixes the row, ensures that rows stay in scope
int fixRow (int num) {

if (num == -1) {

      return rows - 1;
}

if (num == rows) {

      return 0;
}

return num;
}
```

```
// Fixes the col, ensures that cols stay in scope
int fixCol (int num) {

if (num == -1) {

      return cols - 1;
}

if (num == cols) {

      return 0;
}

return num;
}

// The main driver function for rank zero, uses recurrsion to
backtrack
// through the graph and check every point
void backtrack (int count, pair<int, int> start) {

// The ending condition to the recurrsion, means the point is no
longer
// in scope and thus, needs to backtrack
if (start == make_pair (-1, -1)) {

      return;
}

pair<int, int> newPoint = start;
pair<int, int> tempPoint;
int numAdvance = 0;

while (nextPoint (newPoint) != make_pair (-1, -1)) {

      newPoint = nextPoint (newPoint);
      tempPoint = newPoint;
      pointItems.insert (newPoint);

      // Call again if not enough points, otherwise, do work
      if (count != points) {

            if (willLoad == 1) {

                  pair<int, int> nextPair = make_pair
(loadPointArray[loadCount], loadPointArray[loadCount + 1]);
                  loadCount += 2;
                  if (loadCount == (points * 2)){

                        willLoad = 0;
                  }
                  backtrack (count + 1, previousPoint
(nextPair));
                  numAdvance = 0;
```

```
            } else {

                  backtrack (count + 1, newPoint);
                  numAdvance = 0;
            }

      } else if (isPromising()) {
      //} else {

            // Go through all the ranks and send them their
points to check
            for (int i = 0; i < numRanks - 1; i++) {

                  pointItems.erase (tempPoint);
                  tempPoint = nextXPoint (newPoint, i);
                  pointItems.insert (tempPoint);

                  // Turn the points into an array so we can send
them using mpi
                  pairSetToArray (pointItems);
                  MPI_Send (pointArray, points * 2, MPI_INT, i +
1, 0, MPI_COMM_WORLD);
            }

            // Gather up the information from the other ranks
            for (int i = 0; i < numRanks - 1; i++) {

                  int isOLD;
                  MPI_Recv (&isOLD, 1, MPI_INT, i + 1, 0,
MPI_COMM_WORLD, &status);
                  if (isOLD == 1) {

                        OLDCounter++;
                  }
            }

            // Barrier to ensure everyone is on the same page
            MPI_Barrier (MPI_COMM_WORLD);
            numAdvance = numRanks - 2;
            counter++;
            pointItems.erase (tempPoint);
            pointItems.insert (newPoint);
            if (counter >= 100000) {
                  printPointItems(pointItems);
                  pairSetToArray (pointItems);
                  if (arrayToFile (pointArray)) {

                        counter = 0;
                  }
            }

      } else {

            numAdvance = 0;
      }
```

```
      // Erase the point and then jump forward to the next
unchecked point
      pointItems.erase (newPoint);
      if (count == points) {

            newPoint = nextXPoint (newPoint, numAdvance);
      }
}
}


// The main driver function for non zero ranks
bool isOpenLocatingDominating() {

for (int i = 0; i < rows; i++)
      for (int j = 0; j < rows; j++)
            OLDMatrix2[i][j] = matrix[i][j];

bool isOD = isOpenDominating (pointItems);
bool isL;

// Only have to check if isLocating if the set is open dominating
if (isOD) {

      isL = isLocating (OLDMatrix2);

      if (isL) {

            // Print everything out if OLD is found
            cout << "Open-locating-dominating set found!" <<
endl;
            printMatrix (OLDMatrix2);
            return true;

      } else {

            return false;
      }
} else {

      return false;
}
}

// Checks to see if a matrix is open dominating
bool isOpenDominating (set<pair<int, int>, customCompare>
pointItems2) {

domMatrix = new int*[rows];
for (int i = 0; i < rows; ++i)
      domMatrix[i] = new int[cols];

for (int i = 0; i < rows; i++)
      for (int j = 0; j < rows; j++)
```

```
            domMatrix[i][j] = 0;

// For every point in the matrix, surround that point with 1s,
but do not
// put a 1 where the point is
for (const pair<int, int>& curPoint : pointItems2) {

      for (int i = get<0>(curPoint) - 1; i <= get<0>(curPoint) +
1; i++) {

            for (int j = get<1>(curPoint) - 1; j <=
get<1>(curPoint) + 1; j++) {

                  if ((get<0>(curPoint) != i) &&
(get<1>(curPoint) != j))

                        domMatrix[fixRow (i)][fixCol (j)] = 1;
            }
      }
}

// If the entire matrix is filled with 1s, the set is open
dominating, if
// not, return false
for (int i = 0; i < rows; i++)

      for (int j = 0; j < cols; j++)

            if (domMatrix[i][j] == 0) {

                  for (int m = 0; m < rows; m++) {

                        delete[] domMatrix[m];
                  }
                  delete[] domMatrix;
                  return false;
            }

for (int m = 0; m < rows; m++) {
      delete[] domMatrix[m];
}
delete[] domMatrix;

return true;
}

// Checks to see if a matrix is locating
bool isLocating (int** matrix2) {

// Vector required to check to ensure that every point in the
array
// has a unique locating code
locate = new vector<pair<int, int>>[(rows + 4) * (cols + 4)];
int count = 0;
```

```
locMatrix = new int* [rows + 6];
for (int i = 0; i < rows + 6; ++i)
      locMatrix[i] = new int[cols + 6];

bool found;

// Using a for loop so that we can check offset matrices
for (int k = 0; k < cols; k++) {

      found = true;

      for (int i = 0; i < (rows + 4) * (cols + 4); i++) {

            locate[i].clear();
      }
      count = 0;

      // Initialize the locMatrix
      for (int i = 0; i < rows + 6; i++)
            for (int j = 0; j < cols + 6; j++)
                  locMatrix[i][j] = 0;

      // Setting up the offsets and filling up the locMatrix
      for (int i = 3; i < rows + 3; i++)
            for (int j = 0; j < cols + 6; j++) {

                  int temp = (j + k) % cols;
                  locMatrix[i][j] = matrix2[i - 3][temp];
            }

      for (int i = 0; i < 3; i++)
            for (int j = 0; j < cols + 6; j++)
                  locMatrix[i][j] = locMatrix[i + rows][j];

      for (int i = rows + 3; i < rows + 6; i++)
            for (int j = 0; j < cols + 6; j++)
                  locMatrix[i][j] = locMatrix[i - rows][j];

      if (printEverything)
            printLocMatrix (locMatrix);

      // Go through the locMatrix and find each point's
identifying code
      for (int x = 1; x < rows + 5; x++) {
            for (int y = 1; y < cols + 5; y++) {

                  if (printEverything)
                        cout << x << "," << y << endl;

                  for (int i = x - 1; i <= x + 1; i++) {
                        for (int j = y - 1; j <= y + 1; j++) {
                              if ((i != x) || (j != y)) {
                                    if (locMatrix[i][j] == 1) {
```

```
                                                locate[count].push_back
(make_pair (i, j));
                                          }
                                    }
                              }
                        }

                        sort(locate[count].begin(),
locate[count].end());
                        if (printEverything) {

                              for (const pair<int, int>& curPoint :
locate[count]) {
                                    cout << " " << get<0>(curPoint) <<
"," << get<1>(curPoint);
                              }
                              cout << endl << endl;
                        }
                        count++;
                  }
            }

            // Checks to see if a locating code is either zero or has a
duplicate.
            // If either of these are true, then isLocating is false.
            int i = 0;
            while (found && (i < ((rows + 4) * (cols + 4)))) {
                  if (locate[i].empty()) {

                        found = false;
                  }

                  for (int j = i + 1; j < ((rows + 4) * (cols + 4));
j++) {

                        if (locate[i] == locate[j]) {

                              found = false;
                        }
                  }
                  i++;
            }

            if (found) {

                  for (int m = 0; m < rows + 6; m++) {
                        delete[] locMatrix[m];
                  }
                  delete[] locMatrix;
                  delete[] locate;
                  return true;
            }
}

for (int m = 0; m < rows + 6; m++) {
```

```
      delete[] locMatrix[m];
}
delete[] locMatrix;
delete[] locate;
return false;
}

bool isPromising() {

//printPointItems(pointItems);
int neighborCount = 0;

for (const pair<int, int>& curPoint : pointItems) {

      neighborCount = findNeighbors (curPoint, pointItems);
      //cout << "Point:  " << get<0>(curPoint) << "," <<
get<1>(curPoint) << "  Neighbors = " << neighborCount << endl;

      if (neighborCount == 0) {

            //cout << "Is not Promising" << endl;
            return false;
      }

      if (neighborCount > 1) {

            unordered_set<pair<int, int>, a_hash> neighborSet =
getNeighbors (curPoint, pointItems);
            int privateNeighbor = 0;
            for (const pair<int, int>& neighbor : neighborSet) {

                  if (findNeighbors (neighbor, pointItems) == 1)
{

                        privateNeighbor++;
                  }
            }
            //cout << "Private = " << privateNeighbor << endl;
            if (privateNeighbor > 1) {

                  return false;
            }
      }
}

return true;
}

// Adds a point to a matrix
void addPoint (pair<int, int> point, int** matrix2) {

matrix2[get<0>(point)][get<1>(point)] = 1;
}

// Removes a point from a matrix
```

```
void removePoint (pair<int, int> point, int** matrix2) {

matrix2[get<0>(point)][get<1>(point)] = 0;
}

// Gets the next point from the previous one, returns (-1, -1) if
that point is out
// of scope
pair<int, int> nextPoint (pair<int, int> point) {

if (get<0>(point) < 0) {

      return (make_pair(-1, -1));
}
if ((get<1>(point) + 1) != cols) {

      return (make_pair (get<0>(point), get<1>(point) + 1));
}
else if ((get<0>(point) + 1) != rows) {

      return (make_pair (get<0>(point) + 1, 0));
}
else {

      return (make_pair (-1, -1));
}
}

pair<int, int> previousPoint (pair<int, int> point) {

if (get<0>(point) < 0) {

      return (make_pair(-1, -1));
}
if ((get<1>(point) - 1) != -1) {

      return (make_pair (get<0>(point), get<1>(point) - 1));
}
else if ((get<0>(point) - 1) != -1) {

      return (make_pair (get<0>(point) - 1, cols - 1));
}
else {

      return (make_pair (-1, -1));
}
}

// Gets the point that is X away from the point provided, returns
(-1, -1) if that
// point is out of scope
pair<int, int> nextXPoint (pair<int, int> point, int x) {

pair<int, int> currentPoint = point;
for (int i = 0; i < x; i++) {
```

```
      currentPoint = nextPoint (currentPoint);
}

return currentPoint;
}

// Changes a pair set of points to an array of points
void pairSetToArray (set<pair<int, int>, customCompare>
pointItems2) {

int pointCounter = 0;
for (const pair<int, int>& point : pointItems2) {

      pointArray[pointCounter * 2] = get<0>(point);
      pointArray[(pointCounter * 2) + 1] = get<1>(point);
      pointCounter++;
}
}

// Changes an array of points back to a pair set of points,
returns a false
// if one of those points are negative
bool arrayToPairSet (int* pointArray2) {

bool isNeg = false;
pointItems.clear();
for (int i = 0; i < points; i++) {

      if (pointArray2[i * 2] < 0) {

            isNeg = true;
      }
      pointItems.insert (make_pair (pointArray2[i * 2],
pointArray2[(i * 2) + 1]));
}
//cout << endl;

return isNeg;
}

bool arrayToFile (int* pointArray2) {

for (int i = 0; i < points * 2; i++) {

      if (pointArray2[i] < 0) {

            return false;
      }
}
ofstream arrayFile ("tempOLD.txt");
if (arrayFile.is_open()) {

      for (int i = 0; i < points * 2; i++) {
```

```
            arrayFile << pointArray2[i] << " ";
      }
      arrayFile.close();
      return true;
    }

    return false;
}

bool fileToArray (int* pointArray2) {

ifstream arrayFile ("tempOLD.txt");
if (arrayFile.is_open()) {
      for (int i = 0; i < points * 2; i++) {

            arrayFile >> pointArray2[i];
      }
}

cout << endl;
}

int findNeighbors (pair<int, int> point, set<pair<int, int>,
customCompare> pointItems2) {

pair<int, int> negativePair = make_pair(-1, -1);
int neighbors = 0;

if (checkNeighbor (point, pointItems2, -1, 0) != negativePair) {

      neighbors++;
}
if (checkNeighbor (point, pointItems2, 1, 0) != negativePair) {

      neighbors++;
}
if (checkNeighbor (point, pointItems2, 0, -1) != negativePair) {

      neighbors++;
}
if (checkNeighbor (point, pointItems2, 0, 1) != negativePair) {

      neighbors++;
}

if (checkNeighbor (point, pointItems2, -1, -1) != negativePair) {

      neighbors++;
}
if (checkNeighbor (point, pointItems2, 1, -1) != negativePair) {

      neighbors++;
}
if (checkNeighbor (point, pointItems2, -1, 1) != negativePair) {
```

```
      neighbors++;
}
if (checkNeighbor (point, pointItems2, 1, 1) != negativePair) {

      neighbors++;
}

return neighbors;
}

unordered_set<pair<int, int>, a_hash> getNeighbors (pair<int, int>
point, set<pair<int, int>, customCompare> pointItems2) {

pair<int, int> negativePair = make_pair(-1, -1);
unordered_set<pair<int, int>, a_hash> neighborSet;
if (checkNeighbor (point, pointItems2, -1, 0) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, -1,
0));
}
if (checkNeighbor (point, pointItems2, 1, 0) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, 1,
0));
}
if (checkNeighbor (point, pointItems2, 0, -1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, 0, -
1));
}
if (checkNeighbor (point, pointItems2, 0, 1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, 0,
1));
}

if (checkNeighbor (point, pointItems2, -1, -1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, -1,
-1));
}
if (checkNeighbor (point, pointItems2, 1, -1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, 1, -
1));
}
if (checkNeighbor (point, pointItems2, -1, 1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, -1,
1));
}
if (checkNeighbor (point, pointItems2, 1, 1) != negativePair) {

      neighborSet.insert (checkNeighbor (point, pointItems2, 1,
1));
```

```
}

return neighborSet;
}

pair<int, int> checkNeighbor (pair<int, int> point, set<pair<int,
int>, customCompare> pointItems2, int zeroAdjust, int oneAdjust)
{

pair<int, int> tempPoint = make_pair (fixRow (get<0>(point) +
zeroAdjust), fixCol (get<1>(point) + oneAdjust));
return checkPointInSet (tempPoint, pointItems2);
}

pair<int, int> checkPointInSet (pair<int, int> point,
set<pair<int, int>, customCompare> pointItems2) {

if (pointItems2.count (point)) {

      return point;

} else {

      return make_pair (-1, -1);
}
}
```

APPENDIX B

SOURCE CODE FOR OLD SETS IN FINITE GRAPHS

OLDSetsFinite.h

```
using namespace std;

#pragma once
#include <iostream>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <unordered_set>
#include <vector>
#include <sstream>
#include <algorithm>
#include <fstream>
#include "mpi.h"

struct Edge {

int src, dest;
};

class Graph {

public:
	vector<vector<int>> adjList;

	void createGraph (vector<Edge> const &edges, int N) {

		adjList.resize(N);

		for (auto &edge: edges) {

			adjList[edge.src].push_back(edge.dest);
		}
	}
};

// pointArray is used to send the points from rank 0 to the other
ranks, points
// are the inputs, graphItems are all the points in the graph and
their connections,
// pointItems are the points in set format, sets are used to
ensure
// that the points are not duplicated, locate is used by
isLocating
int* pointArray;
int points;
unordered_set<int> pointItems;
vector<int> *locate;
```

```
// printEverything is an old boolean used to print stuff,
isWorking is used to tell all
// the ranks when to stop
bool printEverything;
bool isWorking;

// numRanks is the number of ranks used, curRank is the current
rank of the server, status
// and request are used in sends and receives
int numRanks;
int curRank;
MPI_Status status;
MPI_Request request;

// Counters used to count various things, like number of OLD sets
found
int counter;
int OLDCounter;

vector<Edge> forwardEdgeVector;
vector<Edge> backwardEdgeVector;
Graph forwardGraph;
Graph backwardGraph;
int vertexCount;

// Intializes everything that needs to be initialized
void initialize();

// Prints the unordered set, used for debugging
void printUnorderedSet (unordered_set<int> currentSet);

void setUpGraph (string filename);

// Prints the graph
void printGraph (Graph const &graph);

// The main driver function for rank zero
void backtrack (int count, int start);

// The main driver function for non zero ranks
bool isOpenLocatingDominating();

// Checks to see if a point set is locating
bool isLocating ();

// Gets the next point from the previous one, returns -10 if that
point is out
// of scope
int nextPoint (int point);

// Gets the point that is X away from the point provided, returns
-10 if that
// point is out of scope
int nextXPoint (int point, int x);
```

```
// Changes a pair set of points to an array of points
void setToArray (unordered_set<int> pointItems2);

// Changes an array of points back to a pair set of points,
returns a false
// if one of those points are negative
bool arrayToSet(int* pointArray2);
```

OLDSetsFinite.cpp

```
#include "OLDSetsFinite.h"

int main(int argc, char** argv) {

// Alarm set so that I don't get yelled at, can adjust for need
alarm (3000);

// Initialize the mpi and get the ranks and current rank
MPI_Init (NULL, NULL);
MPI_Comm_size (MPI_COMM_WORLD, &numRanks);
MPI_Comm_rank (MPI_COMM_WORLD, &curRank);

printEverything = false;
isWorking = true;

setUpGraph (argv[1]);
points = stoi (argv[2]);

initialize();

if (curRank == 0) {

      clock_t tStart = clock();

      // Call the backtrack function from rank 0
      backtrack (1, -1);

      // When the backtrack function is done, tell the other
ranks to stop working
      isWorking = false;
      for (int i = 0; i < numRanks - 1; i++) {

            MPI_Send (&isWorking, 1, MPI_INT, i + 1, 999,
MPI_COMM_WORLD);
      }

      cout << "Found " << OLDCounter << " OLD sets." << endl;
      printf("Time taken: %.2fs\n", (double)(clock() - tStart) /
CLOCKS_PER_SEC);

      cout << "Program has reached the end." << endl;
      cout << "Press ENTER to continue.";
      cin.ignore();
      cin.get();
```

```
      MPI_Abort (MPI_COMM_WORLD, -1);

} else {

      bool isOLD;

      MPI_Irecv(&isWorking, 1, MPI_INT, 0, 999, MPI_COMM_WORLD,
&request);

      while (isWorking) {

            // Receive the point array from rank 0
            MPI_Recv (pointArray, points, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

            // Turn that point array into a set
            bool isNeg = false;
            isNeg = arrayToSet (pointArray);

            // As long as there are no negatives in the set,
check to see if OLD
            // Otherwise, return false
            if (!isNeg) {

                  int counter = 0;

                  isOLD = isOpenLocatingDominating();

            } else {

                  isOLD = false;
            }

            // Send the results back to rank 0
            int sendOLD = isOLD;
            MPI_Send (&sendOLD, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD);

            // Re-initalize and barrier
            initialize();
            MPI_Barrier (MPI_COMM_WORLD);
      }
}
}


// Intializes everything that needs to be initialized
void initialize() {

pointItems.clear();

pointArray = new int [points];

counter = 0;
OLDCounter = 0;
```

```
}

// Prints the point items in the set, used for debugging
void printUnorderedSet (unordered_set<int> currentSet) {

cout << "pointItems contains:  ";
for (const int& curPoint : currentSet) {

      cout << curPoint << " ";
}
cout << endl;
}

void printGraph (Graph const &graph) {

for (int i = 0; i < vertexCount; i++) {

      cout << i << "--> ";

      for (int v : graph.adjList[i]) {

            cout << v << " ";
      }

      cout << endl;
}
}

void setUpGraph (string filename) {

ifstream file (filename);
string str;
unordered_set<int> uniquePoints;

while (getline (file, str)) {

      stringstream ss (str);
      Edge forwardEdge, backwardEdge;
      string first, second;
      getline (ss, first, ',');
      getline (ss, second, ',');
      uniquePoints.insert (stoi (first));
      forwardEdge.src = stoi (first);
      forwardEdge.dest = stoi (second);
      backwardEdge.dest = stoi (first);
      backwardEdge.src = stoi (second);
      forwardEdgeVector.push_back (forwardEdge);
      backwardEdgeVector.push_back (backwardEdge);
}

vertexCount = uniquePoints.size();
forwardGraph.createGraph (forwardEdgeVector, vertexCount);
backwardGraph.createGraph (backwardEdgeVector, vertexCount);
}
```

```
// The main driver function for rank zero, uses recurrsion to
backtrack
// through the graph and check every point
void backtrack (int count, int start) {

// The ending condition to the recurrsion, means the point is no
longer
// in scope and thus, needs to backtrack
if (start < -1) {

      return;
}

int newPoint = start;
int tempPoint;
int numAdvance = 0;

while (nextPoint (newPoint) > -1) {

      newPoint = nextPoint (newPoint);
      tempPoint = newPoint;
      pointItems.insert (newPoint);

      // Call again if not enough points, otherwise, do work
      if (count != points) {

            backtrack (count + 1, newPoint);
            numAdvance = 0;

      } else {

            // Go through all the ranks and send them their
points to check
            for (int i = 0; i < numRanks - 1; i++) {

                  pointItems.erase (tempPoint);
                  tempPoint = nextXPoint (newPoint, i);
                  pointItems.insert (tempPoint);
                  //printUnorderedSet (pointItems);

                  // Turn the points into an array so we can send
them using mpi
                  setToArray (pointItems);
                  MPI_Send (pointArray, points, MPI_INT, i + 1,
0, MPI_COMM_WORLD);
            }

            // Gather up the information from the other ranks
            for (int i = 0; i < numRanks - 1; i++) {

                  int isOLD;
                  MPI_Recv (&isOLD, 1, MPI_INT, i + 1, 0,
MPI_COMM_WORLD, &status);
                  if (isOLD == 1) {
```

```
                        OLDCounter++;
                  }
            }

            // Barrier to ensure everyone is on the same page
            MPI_Barrier (MPI_COMM_WORLD);
            numAdvance = numRanks - 2;

      }

      // Erase the point and then jump forward to the next
unchecked point
      pointItems.erase (tempPoint);
      if (count == points) {

            newPoint = nextXPoint (newPoint, numAdvance);
      }

}
}

// The main driver function for non zero ranks
bool isOpenLocatingDominating() {

//bool isOD = isOpenDominating (pointItems);
bool isOD = true;
bool isL;

// Only have to check if isLocating if the set is open dominating
if (isOD) {

      isL = isLocating ();

      if (isL) {

            // Print everything out if OLD is found
            cout << "Open-locating-dominating set found!" <<
endl;
            printUnorderedSet (pointItems);
            return true;

      } else {

            return false;
      }
} else {

      return false;
}
}

// Checks to see if a point set is locating
bool isLocating () {
```

```
// Vector required to check to ensure that every point in the
array
// has a unique locating code
locate = new vector<int>[vertexCount];
int count = 0;

bool found;

// Using a for loop so that we can check offset matrices

found = true;

for (int i = 0; i < vertexCount; i++) {

      locate[i].clear();
}

// Go through the locMatrix and find each point's identifying
code
for (int y = 0; y < vertexCount; y++) {

      if (printEverything)
            cout << y << endl;

      for (int v : forwardGraph.adjList[y])
            if (pointItems.count (v) == 1)
                  locate[y].push_back (v);

      for (int v : backwardGraph.adjList[y])
            if (pointItems.count (v) == 1)
                  locate[y].push_back (v);

      sort(locate[y].begin(), locate[y].end());

      if (printEverything) {

            for (const int& curPoint : locate[y]) {

                  cout << curPoint <<  ", ";
            }
            cout << endl << endl;
      }
      count++;
}

// Checks to see if a locating code is either zero or has a
duplicate.
// If either of these are true, then isLocating is false.
int i = 0;
while (found && (i < vertexCount)) {
      if (locate[i].empty()) {

            found = false;
      }
```

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

		if (locate[i] == locate[j]) {

			found = false;
		}
	}
	i++;
}

if (found) {

	delete[] locate;
	return true;
}

delete[] locate;
return false;
}

// Gets the next point from the previous one, returns -10 if that
point is out
// of scope
int nextPoint (int point) {

if (point < -1) {

	return -10;

} else if ((point + 1) < vertexCount) {

	return point + 1;

} else {

	return -10;
}
}

// Gets the point that is X away from the point provided, returns
-10 if that
// point is out of scope
int nextXPoint (int point, int x) {

int currentPoint = point;
for (int i = 0; i < x; i++) {

	currentPoint = nextPoint (currentPoint);
}

return currentPoint;
}

// Changes a pair set of points to an array of points
void setToArray (unordered_set<int> pointItems2) {
```

```
int counter = 0;
for (const int& point : pointItems2) {

      pointArray[counter] = point;
      counter++;
}
}

// Changes an array of points back to a pair set of points,
returns a false
// if one of those points are negative
bool arrayToSet (int* pointArray2) {

bool isNeg = false;
pointItems.clear();
for (int i = 0; i < points; i++) {

      if (pointArray2[i] < 0) {

            isNeg = true;
      }
      pointItems.insert (pointArray2[i]);
}

return isNeg;
}
```