Isolation in Synchronized Drone Formations

Andrew P. Brunner
University of Denver

Follow this and additional works at: https://digitalcommons.du.edu/etd

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
https://digitalcommons.du.edu/etd/1013
ISOLATION IN SYNCHRONIZED DRONE FORMATIONS

A Thesis

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Andrew P. Brunner

August 2015

Advisor: Mario A. Lopez
Abstract

This paper expands on a theoretical model that is used for aerial robots that are working cooperatively to complete a task. In certain situations, such as when multiple robots have catastrophic failures, the surviving robots could become isolated so that they never again communicate with another robot. We prove some properties about isolated robots flying in a grid formation, and we present an algorithm that determines how many robots need to fail to isolate at least one robot. Finally, we propose a strategy that eliminates the possibility of isolation altogether.
Acknowledgements

Andy Brunner would like to thank Dr. Mario Lopez for giving him the opportunity to work on this project and for his guidance throughout the project. In addition, he would like to thank Professors J.M. Díaz-Bañez and S. Bereg for their discussions related to the project, as well as E. Caraballo for his input on Chapter 5. He would also like to thank the Oral Defense Committee for their time and feedback. Finally he would like to thank the faculty and staff of the Department of Computer Science at the University of Denver for all their instruction and support over the years.
### Table of Contents

Chapter One: Introduction ........................................................................................................... 1

Chapter Two: Background ......................................................................................................... 2

Chapter Three: Distance Between Robots .................................................................................. 5

Chapter Four: Properties of $n \times m$ grids ............................................................................. 8
  Proof that an $n \times m$ grid has $\gcd(n, m)$ rings ................................................................. 8
  Resilience of an $n \times m$ grid ............................................................................................... 12

Chapter Five: Algorithm for 1-resilience ................................................................................... 18
  Preliminaries .......................................................................................................................... 18
  The Algorithm ...................................................................................................................... 20
  Proof of Correctness .............................................................................................................. 22
  Theoretical Running Time ..................................................................................................... 23
  Test Cases .............................................................................................................................. 27
  Results ..................................................................................................................................... 28

Chapter Six: Modified Tree Strategy ......................................................................................... 34
  Proof that sectors do not starve in the tree strategy .............................................................. 34
  Modified Tree Strategy ......................................................................................................... 35

Chapter Seven: Conclusions and Future Work ........................................................................ 37

Bibliography .............................................................................................................................. 38

Appendix A: Source Code ....................................................................................................... 39
List of Figures

Figure 1: A simple synchronized formation. ................................................................. 3
Figure 2: An example of failure. ..................................................................................... 4
Figure 3: A 2x2 grid. ......................................................................................................... 5
Figure 4: A ring segment. ................................................................................................. 6
Figure 5: A ring in a 4x4 grid that hits each wall exactly once. ....................................... 9
Figure 6: The two rings merge into one when a link is added. ....................................... 10
Figure 7: A 2x4 grid with a 4x4 grid below it. After adding the 4 links, it becomes a 6x4 grid. ................................................................................................................................. 11
Figure 8: Two drones in the same row meet. .................................................................. 15
Figure 9: If there is a robot on a ring at position p, then in t time, there will be a robot on the ring at position p+t. ......................................................................................................... 19
Figure 10: Partial execution of the 1-resilience algorithm. ............................................. 22
Figure 11: Pseudocode for the 1-Resilience algorithm ..................................................... 23
Figure 12: An example of a hanging graph. ................................................................... 25
Figure 13: A chain graph .................................................................................................. 26
Figure 14: A simple tree. ................................................................................................. 27
Figure 15: A line. ............................................................................................................... 28
Figure 16: Runtime of the algorithm tested on lines....................................................... 29
Figure 17: Runtime of the algorithm tested on n x n grids............................................ 29
Figure 18: Runtime of the algorithm tested on hanging graphs ..................................... 30
Figure 19: Log runtime of the algorithm tested on hanging graphs. ............................. 31
Figure 20: Runtime of the algorithm tested on chains graphs. .............................. 32

Figure 21: Log runtime of the algorithm tested on chains graphs. .......................... 33
Chapter One: Introduction

Aerial robots (ARs) flying in coordination can be used for a variety of tasks such as construction, area patrol, or any task that requires the ARs to communicate frequently. This paper is an expansion of the simplified theoretical model proposed in [1]. In particular, we consider cases where some ARs on the team fail. We investigate scenarios in which failures cause the team becomes so sparse that at least one robot will never again communicate with another robot on the team. We call this starvation. We investigate the flight path that such a robot would take, and how many failures would be required for such a scenario to occur. We prove some general properties of the theoretical model, and some properties that apply specifically to grid formations. We propose an algorithm that determines how many robots must fail for at least one robot to starve. Finally, we propose a strategy under which it would be impossible for a robot to starve unless every other robot on the team had failed.

This thesis is organized as follows: Chapter 2 explains the theoretical model and what has already been proven about it. Chapter 3 presents a proof of the distance between robots in the system. Chapter 4 proves some properties of formations consisting of an $n \times m$ grid. Chapter 5 shows an algorithm to determine how many robots must fail for a single robot to starve. Chapter 6 presents a new strategy to eliminate the possibility of starvation. Lastly, Chapter 7 discusses future work to be done.
Chapter Two: Background

The simplified theoretical model described in [1] is composed of the following assumptions. The robots are flying on a 2D plane and each robot has its own trajectory that it continuously patrols. See Figure 1. Each trajectory is a circle of unit radius, and each robot flies along the circumference of its circle. Every vehicle flies at the same speed and has a fixed communication range. If one robot is within the communication range of another, then the two robots can communicate. A communication link exists between two paths if the minimum distance between the paths is less than the communication range. We say that two circles are tangent if they share a communication link. With these assumptions, we can create a communication graph, in which the trajectories are the nodes of the graph, and the communication links are the edges. We also assume that the communication graph is planar. If, in the course of flying, each robot meets another robot every time it comes to a tangency point, then the system is synchronized. In this paper, we only consider formations that start out synchronized.

In [1], a series of theorems about the theoretical model were proved, first considering the case where all robots are flying in the same direction (i.e. all clockwise or all counter-clockwise), and then exploring the case when some robots are flying clockwise and others are flying counter-clockwise. An important insight was that a team
of robots can be synchronized if and only if the underlying communication graph is bipartite, which applied to both the single and bi-directional scenarios.

![Figure 1: A synchronized formation at three different moments in time where neighboring robots are flying in opposite directions.](image)

One focus of [1] was the possibility that as the robots are carrying out their task, one or more of the robots leave the system. This could be due to need to refuel, mechanical failure, or other unforeseen hazards. One way to compensate for a robot’s failure is to have a neighboring robot take over the empty trajectory. When all of the robots are flying in a single direction, this could pose a problem as a robot would have to reverse direction in midair when it crosses into a new trajectory. This may not be possible for an AR to do while also maintaining synchronization. When neighboring robots are flying in opposite directions however, a robot can smoothly enter a neighboring trajectory without much change in speed or direction. See Figure 2. The circles in Figure 2 are drawn apart to make the communication links easy to see, but in reality the distance between trajectories would be very small, or could be made up for by speeding up slightly. For modeling purposes, we assume that robots cross communication links instantaneously. [1] also proved that “In a synchronized system
allowing swaps in case of failures, each path is occupied by at most one AR.” So if a robot comes to a communication link, and there is not a robot on the other side, it knows the other trajectory is empty, so it will cross over into it. Each robot will continue to fly in this way: switching trajectories each time it does not see another robot across the communication link.

![Diagram of robot trajectories](image.png)

*Figure 2: In this example, the robot in the top right trajectory has failed. When the robot in the bottom right trajectory sees that there is no robot in the top right trajectory, it crosses over to the empty trajectory.*

[1] proposed this strategy for covering the path of a missing neighbor, but it did not investigate how far the possibility of failures could be pushed. This paper explores this possibility and proposes a very failure tolerant solution.
Chapter Three: Distance Between Robots

Definition 3.1. A robot *starves* if, after some point in time, it never meets another robot at a tangency point.

Definition 3.2. Let $l_1$ and $l_2$ be two communication links on a trajectory, $t$, where $t$ has no other links between $l_1$ and $l_2$. A *sector* is the part of $t$ between $l_1$ and $l_2$.

Definition 3.3. A *ring* is the path a robot takes if it is starving. See Figure 3. A ring is made of a series of sectors $S_1, S_2, \ldots, S_m$ where a starving robot will traverse the sectors in the order $S_1, S_2, \ldots, S_m, S_1, S_2, \ldots$

![Figure 3](image.png)

*Figure 3: This formation has two rings, one colored in red, and the other colored in blue. The two drones on the red ring will continually miss each other, thus they are both starving and will continue to circle the red ring.*

Theorem 3.1. If robot $r_p$ is on a ring, $R$, and robot $r_q$ is on the same ring, and $r_p$ and $r_q$ are synchronized, then $r_q$ is $2m\pi$ ahead of $r_p$, for some $m \in \mathbb{N}$.

Proof. Let the segment of the ring between $r_p$ and $r_q$ be called $S$. See Figure 4.
Let $k$ be the number of sectors that $S$ traverses (including the sectors $r_p$ and $r_q$ are on).

Starting at $r_p$ and following $S$, rename the robots such that robot $r_i$ is on the trajectory that contains the $i^{th}$ sector that $S$ traverses (e.g. $r_p$ becomes $r_1$, $r_q$ becomes $r_k$). Let $t_i$ be the amount of time it takes the robot to traverse the $i^{th}$ sector, and let $\phi_{a,b}$ be the location on trajectory $a$ that is linked to trajectory $b$. We claim that $r_j$ will be at $\phi_{j,j+1}$ in $\sum_{i=1}^{j} t_i$ time. Proof by induction:

**Base Case:** By definition, $r_1$ will be at $\sum_{i=1}^{j_1} t_i$ in $t_1$ time.

**Inductive Step:** Suppose $r_{j-1}$ will be at $\phi_{j-1,j}$ in $\sum_{i=1}^{j-1} t_i$ time.
Since $r_{j-1}$ is at $\phi_{j-1,j}$ at time $\sum_{i=1}^{j-1} t_i$, then $r_j$ is at $\phi_{j,j-1}$ at time $\sum_{i=1}^{j-1} t_i$ as well since they are synchronized. Thus, $r_j$ must be at $\phi_{j,j+1}$ in $t_j + \sum_{i=1}^{j-1} t_i = \sum_{i=1}^{j} t_i$ time.

By the above claim, $r_{k-1}$ will be at $\phi_{k-1,k}$ in $\sum_{i=1}^{k-1} t_i$ time. Since $r_{k-1}$ and $r_k$ are synchronized, $r_k$ must be at $\phi_{k,k-1}$ in $\sum_{i=1}^{k-1} t_i$ time as well. $r_k$ is $t_k$ ahead of $\phi_{k,k-1}$ by definition, so it must be at $\phi_{k,k-1}$ again at time $2\nu\pi - t_k$, $\forall \nu \in \mathbb{N}$. Thus, $\exists m \in \mathbb{N}$ such that $\sum_{i=1}^{k-1} t_i = 2m\pi - t_k$, thus, $\sum_{i=1}^{k} t_i = 2m\pi$. Since $\sum_{i=1}^{k} t_i$ is the length of $S$, the distance between $r_1$ and $r_k$ is $2m\pi$, thus, the distance between $r_p$ and $r_q$ is $2m\pi$. 
Chapter Four: Properties of \( n \times m \) grids

*Proof that an \( n \times m \) grid has \( \gcd(n, m) \) rings*

First, we prove a lemma about \( n \times n \) grids. Consider a grid \( G \) consisting of \( n \) rows and \( n \) columns, occupied by \( n^2 \) ARs operating in fully synchronized fashion. Label the nodes so that the bottom left trajectory has coordinates \((0, 0)\) and the top right has coordinates \((n - 1, n - 1)\). Let us say that a ring, \( R \), hits the bottom wall of a grid at \((i, 0)\), \(0 \leq i \leq n - 1\), if \( R \) covers the bottom half of the trajectory that is at position \((i, 0)\). Define hitting the left, right, and top walls similarly. See Figure 5.

**Lemma 4.1.** In an \( n \times n \) grid, if a ring, \( R \), hits one wall, then \( R \) hits every wall exactly once.

*Proof.* Without loss of generality, suppose ring \( R \) hits the bottom wall of an \( n \times n \) grid at trajectory \((i, 0)\), \(0 \leq i \leq n - 1\). Following the ring clockwise, the ring will repeatedly move one trajectory to the left, and one trajectory up until it hits the left wall. Therefore, the ring goes to trajectories \((i, 0) \to (i - 1, 0) \to (i - 1, 1) \to (i - 2, 1) \to (i - 2, 2) \to \cdots \to (0, i - 1) \to (0, i)\). Thus, the ring hits the left wall. Similar proofs show the ring hits the top wall at \((n - i - 1, n - 1)\), the right wall at \((n - 1, n - i - 1)\), and the bottom wall at \((0, i)\), finishing where it started. Thus, a ring that hits one wall hits each wall exactly once.
Figure 5: A ring in a 4x4 grid that hits each wall exactly once.

Lemma 4.2. Let $G$ be a graph with $j \geq 2$ rings. Let $R_1$ and $R_2$ be distinct rings in $G$. Let $s_1$ and $s_2$ be sectors on $R_1$ and $R_2$ respectively. Adding a link, $l$, between $s_1$ and $s_2$ creates a new graph, $G'$ with $j - 1$ rings.

Proof. Link $l$ splits both $s_1$ and $s_2$ into two new sectors. See Figure 6. Let $s_1'$ be part of $s_1$ that flows into $l$, and let $s_1''$ be the part of $s_1$ that flows away from $l$. Let $s_2'$ and $s_2''$ be similarly defined. Consider the ring, $R'$ that covers $s_1'$ in $G'$. It must flow into $l$, and start to cover $s_2''$. Now $R'$ starts to cover the sectors in $G'$ that $R_2$ covered in $G$. Thus, $R'$ must traverse exactly the same sectors $R_2$ did (except $s_2$ which is not in $G'$). Then it will
eventually, come back to \( s'_2 \) cross \( l \), and cover \( s''_1 \). Then, similarly, it will traverse the sectors \( R_1 \) did in \( G \). Finally it will come back to \( s'_1 \), completing the ring. Thus, \( R' \) traverses exactly the paths \( R_1 \) and \( R_2 \) did, and nothing more. So when \( l \) was added, \( R_1 \) and \( R_2 \) merged into \( R' \). Thus \( G' \) has one less ring than \( G \).

**Figure 6:** The two rings merge into one when a link is added.

**Lemma 4.3.** Consider a system that consists of an \( n \times m \) grid with \( j \geq 1 \) rings, and an \( m \times m \) grid below it. If the grids are connected with \( k \) links, then the resulting system has \( j + m - k \) rings.

**Proof.** By induction. See Figure 7.

**Base Case:** \( k = 0 \). The \( n \times m \) grid has \( j \) rings and the \( m \times m \) grid has \( m \) rings, thus, the entire system has \( j + m = j + m - k \) rings.

**Inductive Step:** Assume the grids have \( k - 1 \) links and the system has \( j + m - (k - 1) \) rings. Pick a trajectory, \( t_1 \), from the bottom row of the \( n \times m \) grid that does not have a link to the \( m \times m \) grid. It has a corresponding trajectory, \( t_2 \), on the top.
row of the $m \times m$ grid. Let $R_1$ be the ring that covers the bottom half of $t_1$. Let $R_2$ be the ring that covers the top half of $t_2$. $R_1$ and $R_2$ are distinct rings by Lemma 4.1. Now add a link between $t_1$ and $t_2$. By Lemma 4.2, the resulting graph has $j + m - (k - 1) - 1 = j + m - k$ rings.

**Lemma 4.4.** An $n \times m$ grid has the same number of rings as an $(n + m) \times m$ grid.

*Proof.* Suppose the $n \times m$ grid has $j$ rings. An $(n + m) \times m$ grid is an $n \times m$ grid with an $m \times m$ grid below it, connected by $m$ links. Thus, by Lemma 4.3, the grid has $j + m - m = j$ rings.

**Corollary 4.5.** If $n > m$, then an $n \times m$ grid has the same number of rings as an $(n - m) \times m$ grid.

*Proof.* Suppose $n > m$. Let $k = n - m$. By Lemma 4.4, a $k \times m$ grid has the same number of rings as a $(k + m) \times m$ grid. Since $k + m = n$, an $n \times m$ grid has the same number of rings as an $(n - m) \times m$ grid.
Theorem 4.6. The number of rings of an $n \times m$ grid is the greatest common divisor of $n$ and $m$, $\gcd(n, m)$.

Proof. From Euclid, we know that when $n > 0$, and $m > 0$, the $\gcd(n, m)$ can be defined as

$$\gcd(n, m) = \begin{cases} n, & n = m \\ \gcd(n - m, m), & n > m \\ \gcd(n, m - n), & m > n \end{cases}$$

Suppose that $\text{numRings}(n, m)$ is a function that returns the number of rings in an $n \times m$ grid. Then clearly, $\text{numRings}(n, m) = \text{numRings}(m, n)$. From this fact and Corollary 4.5, we know that $\text{numRings}(n, m)$ can be defined as

$$\text{numRings}(n, m) = \begin{cases} n, & n = m \\ \text{numRings}(n - m, m), & n > m \\ \text{numRings}(n, m - n), & m > n \end{cases}$$

Thus, when $n > 0$, and $m > 0$, $\gcd$ and $\text{numRings}$ are the same function. Therefore, the number of rings of an $n \times m$ grid is the greatest common divisor of $n$ and $m$, $\gcd(n, m)$.

Resilience of an $n \times m$ grid

Consider a grid $G$ consisting of $m$ rows and $n$ columns, occupied by $nm$ ARs (robots) operating in fully synchronized fashion. Label the nodes so that the bottom left trajectory has coordinates $(0,0)$ and the top right has coordinates $(n - 1, m - 1)$.

Definition 4.1. The $k$-resilience of $G$ is the minimum number of robots whose removal results in the starvation for at least $k$ of the surviving robots.

We show that the 1-resilience of a $G$ is $n + m - 2$. We also show that $G$ cannot have more than $\min(n, m)$ starving robots, and that the system can enter full starvation
mode after \(nm - \min(n, m)\) failures, a number that corresponds to the \(\min(n, m)\)-resilience of \(G\).

**Lemma 4.7.** If at any time \(t\), a robot \(r\) is the only robot in its row and column, then \(r\) will not meet another robot at the next link it comes to.

*Proof.* Two possibilities arise.

*Case 1:* The next link \(r\) encounters is a link north or south. Since \(r\) is the only robot in its column, it will not encounter another robot at this link.

*Case 2:* The next link \(r\) encounters is a link east or west. Since \(r\) is the only robot in its row, it will not encounter another robot at this link.

**Lemma 4.8.** If at any time \(t\), a robot \(r\) is the only robot in its row and column and \(r\) crosses a link, then \(r\) remains the only robot in its row and column.

*Proof.* Suppose a robot, \(r\), is the only robot in its row and column. Suppose \(r\) is at the north link of a node in row \(i\). Then each robot in row \(i + 1\) must be at the south link of its node due to synchronization. Since \(r\) is the only robot in its column, it crosses the link north into row \(i + 1\). Since \(r\) is the only robot in row, all the robots in row \(i + 1\) cross south into row \(i\). Thus, \(r\) remains the only robot in its row. Similar proofs show that \(r\) remains the only robot in its row if it crosses a link south, and remains the only robot in its column if it crosses a link east, or west.

**Lemma 4.9.** If at any time \(t\), robot \(r\) is the only robot in its row and column, then \(r\) is starving.
Proof. By induction. After encountering $i$ links, $r$ will be the only robot in its row and column and it will not have met any other robots.

**Base Case:** The case $i = 0$ is true by the hypotheses of the theorem.

**Inductive Step:** Assume that after encountering $i - 1$ links, $r$ is the only robot in its row and column and it has not met any other robots. By Lemma 4.7, $r$ will not meet a robot at the next link (i.e. the $i^{th}$ link). By Lemma 4.8, $r$ will be the only robot in its row and column after crossing the $i^{th}$ link. Thus, after encountering $i$ links, $r$ is the only robot in its row and column and it has not met any other robots. Since $r$ will not meet another robot at any link, it is starving.

**Lemma 4.10.** If two robots are in the same row or column, then they are not both starving.

**Proof.** Suppose two robots $p$ and $q$ are in the same row with $p$ located west of $q$.

Suppose by way of contradiction that they are both starving. We claim that $p$ and $q$ will always be in the same row. To see this, suppose $p$ is at the south link of a node. If the system is synchronized, the $q$ must be at the south link of its node as well. Since they are both starving, they both cross south. Similarly, if $p$ crosses north, $q$ must cross north as well. Thus, $p$ and $q$ will always be in the same row.

Move the system forward until $p$ is at a link going east or west. Since the system is synchronized, $q$ must be at a link going east or west as well.
Case 1: \( p \) is at a link going east, and \( q \) is at a link going west. See Figure 8.

*Induction:* If there are \( 2i \) nodes in between \( p \) and \( q \), not including the nodes they are currently on, then \( p \) and \( q \) will meet.

*Base case:* If \( i = 0 \), they are at the same link.

*Inductive Step:* Assume that \( p \) and \( q \) will meet if there are \( 2(i - 1) \) nodes between them. Suppose \( p \) and \( q \) have \( 2i \) nodes between them. In \( \pi \) time, \( p \) will have moved one column east, and \( q \) will have moved one column west. Thus, they are \( 2(i - 1) \) nodes apart. Thus, they meet.

Case 2: \( p \) is at a link going east, and \( q \) is at a bride going east.
Robot $q$ will continue east. Consider when $q$ is at the link going east into the east-most column. After $2\pi$ time, $q$ will be at a link going west out of the east-most column. Now $p$ is at a link going east, and $q$ is at a link going west. This is Case 1, thus, $p$ and $q$ will meet.

*Case 3:* $p$ is at a link going west, and $q$ is at a link going west.

A proof similar to the proof of Case 2 shows that $p$ and $q$ will meet.

*Case 4:* $p$ is at a link going west, and $q$ is at a link going east.

If $p$ gets to the west-most column, and $q$ gets to the east-most column at the same time, they will both change direction simultaneously. That is, $p$ will encounter a link going east and $q$ will encounter a link going west. This is Case 1, thus, they meet. If $p$ gets to the west-most column before $q$ gets to the east-most column, $p$ gets turned around and will come to a link going east. This is Case 2, thus, they meet. If $q$ gets to the east-most column before $p$ gets to the west-most column, then $q$ gets turned around and will come to a link going west. This is Case 3, thus, they meet.

In every case, $p$ and $q$ meet each other. Contradiction. Therefore, if two robots are in the same row, then they are not both starving. A similar proof shows two starving robots cannot be in the same column.

**Lemma 4.11.** There can be at most $\min(n, m)$ starving robots in the grid.

*Proof.* Lemma 4.9 allows us to force a starving robot in each row and column. By Lemma 4.10, there can be at most one starving robot in each row and column. Thus, there can be at most $\min(n, m)$ starving robots in the grid.
**Theorem 4.12.** Let $G$ be an $n \times m$ grid. For $k \in \{1, 2, ..., \min(n, m)\}$, the $k$-resilience of $G$ is $k(n + m - k - 1)$.

**Proof.** Without loss of generality, assume $n \leq k$. To cause any given robot to starve, it is necessary and sufficient to make it the sole occupant of its row and column at some time $t$. Thus, removing $(n - 1) + (m - 1)$ robots in the same row or column forces the given robot to starve. A second starving robot can be made to starve by removing $(n - 2) + (m - 2)$ additional robots, and so on, up to the $k^{th}$ robot whose starvation is obtained by removing $(n - k) + (m - k)$ robots, for $1 \leq k \leq \min(n, m)$. Note that when $\min(n, m)$ robots are starving the system is in full starvation mode. Adding across all values of $k$, the total number of robots removed is $kn + km - k(k + 1) = k(n + m - k - 1)$, as claimed.

**Corollary 4.13.** Let $G$ be an $n \times m$ grid. The 1-resilience of $G$ is $n + m - 2$ and the full-starvation $\min(n, m)$-resilience of $G$ is $nm - \min(n, m)$. 
Chapter Five: Algorithm for 1-resilience

We have developed an algorithm that will find the 1-resilience of any planar, bipartite graph $G$.

Preliminaries

Definition 5.1. The minimum number of robots that need to fail to make robot $r$ starve is the resilience of $r$.

In the sequel, the length of a ring $R$ will be denoted by $L_R$.

Theorem 5.1. Suppose a robot $r$ is at position $p$ of ring $R$. Then $t$ units of time later (assuming no failures occur), $R$ will have a robot, not necessarily $r$, at position $p + t$.

Proof. Let $k \in \mathbb{N}$ be the number of links on $R$ between $p$ and $p + t$. Let $t_k$ be the distance on $R$ from $p$ to the $k^{th}$ link on $R$ after $p$. Then $\forall t < t_k$, there will be a robot at $p + t$ in $t$ units of time. Proof by induction.

Base Case: $k = 0$. Then $r$ will be at $p + t$ in $t$ time.

Inductive Step: Suppose $\forall t < t_{k-1}$, there will be a robot $r'$ at $p + t$ in $t$ units of time. Thus, $r'$ will be at $p + t_{k-1}$ in $t_{k-1}$ time. When $r'$ comes to the $(k - 1)^{th}$ link, two cases arise (see Figure 9):
Figure 9: If there is a robot on a ring at position p, then in t time, there will be a robot on the ring at position p+t.

Case 1: $r'$ does not meet another robot at the link.

Then $r'$ will cross the link and continue along $R$ until it reaches the $k^{th}$ link. Thus, $\forall t < t_k$, there will be a robot at $p + t$ in $t$ units of time.

Case 2: $r'$ meets another robot $r''$ at the link.

Then $r''$ communicates with $r'$ across the link, and will continue along $R$ reaches the $k^{th}$ link. Thus, $\forall t < t_k$, there will be a robot at $p + t$ in $t$ units of time.

So no matter how many links are between $p$ and $p + t$, if there is a robot at $p$, then in $t$ time, there will be a robot at $p + t$.

Note that while Theorem 5.1 is true for rings, the same is not necessarily true for trajectories. If a robot $r$ is at position $p$ of a trajectory $T$, then $t$ units of time later, $T$ may not have a robot at position $p + t$ (for example if $r$ came across an empty link and left $T$, and no robot entered $T$).

Definition 5.2. A robot or link $a$ is some distance $d$ behind another robot or link $b$ if $a$ and $b$ are on the same ring, and following the direction of the ring, the distance from $a$ to $b$ is $d$. 
**Definition 5.3.** Suppose robot \( r \) is distance \( d \) behind some communication link \( \ell \), and a different robot \( r' \) is also distance \( d \) behind \( \ell \). Then \( r' \) prevents \( r \) from starving because if \( r \) follows its ring, it will meet a robot at \( \ell \) because of \( r' \). Notice \( r \) starves if and only if all the robots that prevent \( r \) from starving fail.

**Theorem 5.2.** All robots on the same ring have the same resilience.

*Proof:* Suppose robot \( r \) is on ring \( R \). Consider a robot \( r' \) that is also on \( R \). Suppose \( r' \) is \( 2k\pi, k \in \mathbb{N} \) behind \( r \). Then for every robot \( s \), if \( s \) prevents \( r \) from starving, then the robot \( 2k\pi \) behind \( s \) prevents \( r' \) from starving by Theorem 5.1. Thus, for every robot that prevents \( r \) from starving, there is a robot that prevents \( r' \) from starving. Therefore, \( r \) and \( r' \) have the same resilience. Since any pair of robots on the same ring have the same resilience, all robots on the same ring have the same resilience.

**The Algorithm**

In order to determine the 1-resilience of a graph \( G \) with \( n \) trajectories, we determine the resilience of one robot on every ring of \( G \). If we take the minimum of these resiliences, that is the 1-resilience of \( G \).

Consider the system at some time \( t \), before any robots have failed. Without loss of generality, suppose \( t = 0 \). Consider a robot \( r \) on a ring \( R \) and a link \( \ell \) on \( R \). We want to determine how many robots must fail in order for \( r \) to starve. Suppose \( r \) is distance \( d \) behind \( \ell \), and that \( S \) is the other ring on this link (note that \( S \) and \( R \) may be the same ring). By Theorem 5.1, we know if there is a robot \( r' \) on \( S \) that is \( d \) behind \( \ell \), it would prevent \( r \) from starving. Thus, two cases arise:
Case 1: Robot \( r' \) on \( S \) that is \( d \) behind \( \ell \) has not been removed yet.

Since \( r' \) prevents \( r \) from starving, we remove it. Note that because \( r \) is \( d \) behind \( \ell \), it is also \( m \cdot L_R + d \), \( \forall m \in \mathbb{N} \), behind \( \ell \). Thus, any robot on \( S \) other than \( r \) (recall that \( R \) and \( S \) may be the same ring) that is \( m \cdot L_R + d \) behind \( \ell \) prevents \( r \) from starving. So we remove every robot that is \( m \cdot L_R + d \) behind \( \ell \), where \( m < j \) and \( j \) is the smallest number such that \((j \cdot L_R + d) \mod L_S = d\). At this point, we can stop because if we go back further, we will be visiting positions of robots we have already removed. So we have removed every robot that could make a robot meet \( r \) at that link.

Case 2: Robot \( r' \) on \( S \) that is \( d \) behind \( \ell \) has been removed already.

If \( r' \) was already removed, then we must have also removed all the robots that were \( m \cdot L_R \) behind it. Thus, we have already removed every robot that could make a robot meet \( r \) at this link.

We repeat this process for every link in \( R \). Note that if \( R \) crosses the same link twice, then that link will go through this process twice, once from each direction. After this has been done, we know that \( r \) will not see another robot at any link on \( R \), thus, it is starving, and we know the resilience of \( r \) (the number of robots we removed). We replace the removed robots and repeat this process for one robot on every ring in the system. The minimum resilience of those robots is the 1-resilience of \( G \). See Figure 11 for the pseudocode.

\[^{1}\text{We include 0 in this set.}\]
**Proof of Correctness**

The 1-resilience of a graph $G$ is the minimum number of failures that could result in one robot starving. By Theorem 5.2 we know that all robots on the same ring have the same resilience. Thus, we need only calculate the resilience of one robot on each ring to determine the resilience of every robot in the system. The minimum of these calculated resiliences is the 1-resilience of $G$.

---

**Figure 10**: (a) shows the complete system with rings. The drone circled in green is the one that we attempt to make starve. (b) shows that the drone in the top right trajectory is not viable. (c) shows that the robot on the bottom trajectory is not viable, because it will cause the drone circled in green to meet another robot the second time it comes to that link.
Figure 11: Pseudocode for the One-Resilience algorithm

Theoretical Running Time

The for-loop on line 2 runs once for every ring, and the for-loop on line 5 runs once for every visit to a communication link that ring has (note a ring may visit a communication link twice). Each communication link is made of two (not necessarily distinct) rings. Thus, combined, the two for-loops run twice for every edge on the graph. If \( n \) is the number of trajectories in a graph, then we know that a connected graph has at least \( n - 1 \) edges, and by Euler’s formula, we know that a planar bipartite graph has at most \( 2n - 4 \) edges, where \( n \) is the number of nodes in the graph. So the code inside both for-loops (but outside the do-while loop) will run at least \( 2(n - 1) \) times, and at most \( 4n - 8 \) times, i.e. in \( \Theta(n) \) time. Note that for each time we calculate the resilience of a
robot, the code inside the do-while loop starting on line 13 can run at most $n - 1$ times total. The if-statement on line 11 prevents any robot from being removed more than once, and at most $n - 1$ robots can be removed (every robot except $r$). So calculating the resilience of a robot takes $O(n)$ time. Since we only calculate the resilience for one robot on each ring, the worst case running time for this algorithm is $O(rn)$ where $r$ is the number of rings in the graph.

Note that for any grid or line, the length of the rings in that system are all equal. Thus, every time the do-while loop is encountered, it runs only once. So the two for-loops determine the running time of the algorithm in these cases. Since the two for-loops combined run $\Theta(n)$ times as mentioned above, grid and lines run in $\Theta(n)$ time.

Grids and lines are, up to a constant, best cases for the algorithm. The algorithm takes much longer on other types of graphs. We call one such type of graph a hanging graph. It is based on an $m \times m$ grid, but each column $i$ has a line of length $i$ added to the top of it. See Figure 12.
These lines add a different number of trajectories to each ring in the $m \times m$ grid, making each ring have a different length. This guarantees that the do-while loop will run multiple times each time it is encountered (remember the do-while loop only runs once each time it is encountered in a normal grid). So there are $m^2$ trajectories in the grid portion, and $\sum_{i=0}^{m-1} i = \frac{1}{2} (m - 1) m$ trajectories total in the lines. Thus, the graph has $n = m^2 + \frac{1}{2} m^2 - \frac{1}{2} m$ trajectories total. We also know that the graph has $m$ rings because the $m \times m$ grid had $m$ rings by Theorem 4.6, and adding the lines did not change
this by Lemma 4.1. Since $m = \Theta(\sqrt{n})$, we know that the algorithm takes $\Theta(n\sqrt{n}) = \Theta(n^{1.5})$ time on hanging graphs. Consequently, the worst-case is $\Omega(n^{1.5})$.

There are graphs that perform even worse than hanging graphs. We found a type of graph that has $\Theta(n)$ rings. We call it a chain graph. See Figure 13.

![Figure 13: A chain graph](image)

A chain graph is multiple $2 \times 2$ grids chained together. We can see that there is one unique ring on each $2 \times 2$ grid, and one more that chains them all together, thus there are $\frac{n}{4} + 1 = \Theta(n)$ rings in the graph. The length of each of the small rings is 2 circles.
(i.e. \( O(1) \)). The big ring touches every node in the graph, so we know it has length \( \Theta(n) \).

We must calculate the resilience of a robot on each of the small rings, and each time we do that, the do-while loop has to execute \( \Theta(n) \) times. Since there are \( \Theta(n) \) rings in the grid, this whole process takes \( \Theta(n^2) \) time.

Thus, the best case running time for the 1-resilience algorithm is \( \Theta(n) \), and the worst case running time is \( \Theta(n^2) \).

**Test Cases**

We implemented the 1-resilience algorithm in Java (see Appendix A for source code) and tested the correctness of the algorithm on the following graphs: \( 2^n \times 2^n \) grids, \( 2 \leq n \leq 8 \), a simple tree (Figure 14), the graph in Figure 10, and lines of length \( 2^n \), \( 2 \leq n \leq 16 \), (Figure 15). The program can also be used to test the 1-resilience of any \( n \times m \) grid and lines of any length.

To empirically test the running time of the algorithm, we applied it to the following graphs: lines of length \( 2^n \), \( 2 \leq n \leq 16 \), \( 2^n \times 2^n \) grids, \( 2 \leq n \leq 8 \), hanging grids where \( 2 \leq m \leq 50 \), and chain grids of size \( 8i + 4 \), \( 0 \leq i \leq 300 \).

*Figure 14: A simple tree.*
Results

The results given by the algorithm are correct as can be calculated by hand. See Table 1. Notice that the 1-resilience of the grids matches the theoretical result given in Chapter Four.

Table 1

<table>
<thead>
<tr>
<th>Graph</th>
<th>1-Resilience (from the Java program)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^n \times 2^n$ grid $2 \leq n \leq 8$</td>
<td>$2^{n+1} - 2$</td>
</tr>
<tr>
<td>4-node tree from Figure 14</td>
<td>2</td>
</tr>
<tr>
<td>6-node graph from Figure 10</td>
<td>4</td>
</tr>
<tr>
<td>$2^n$ node line $2 \leq n \leq 16$</td>
<td>$2^n - 1$</td>
</tr>
</tbody>
</table>

In order to empirically verify our theoretical running times, we tested the algorithm on many graphs and recorded the number of times the do-while loop ran in total. The runtimes for the test cases of the lines and grids are given in Figure 16 and Figure 17. The runtime is linear, which matches our theoretical analysis.
Figure 16: Runtime of the algorithm tested on lines.

Figure 17: Runtime of the algorithm tested on $n \times n$ grids
Next we ran the algorithm on hanging graphs, $2 \leq m \leq 50$. See Figure 18.

![1-Resilience Algorithm Running Time for Hanging Graphs](image)

Figure 18: Run time of the algorithm tested on hanging graphs.

The running time looks non-linear, but it is difficult to see from this plot what the growth rate is, so we take the log of both axes. The slope of the line in this line should show us the running time. See Figure 19.

The slope is 1.503, indicating that the algorithm runs in $\Theta(n^{1.5})$ time for hanging graphs, which matches the theoretical result.
Finally, we tested chain graphs (of size $8i + 4$, $0 \leq i \leq 300$) and examined their running times$^2$. See Figure 20.

---

$^2$ Only testing graphs of size $8i + 4$ made for easier coding. Theoretically we would not expect chain graphs of other sizes to have noticeably different runtimes than these.
Figure 20: The run time of the algorithm tested on chains.
The growth rate is clearly non-linear this time. To get a better idea of what the growth rate is, we again take the log of both sides of the graph. The slope of the resulting line will be the exponent of the run time. See Figure 21. The slope of the line is 1.965. This verifies our theoretical result that chain graphs run in $\Theta(n^2)$ time.

Figure 21: Log runtime of the algorithm tested on chains.
Chapter Six: Modified Tree Strategy

J.M. Díaz-Báñez, M.A. Lopez, and S. Bereg proposed a new strategy concerning when robots should cross communication links. In the original strategy, a robot crosses a communication link if it does not see another robot on the other side of the link. In the new strategy, robots only switch when they do not see a robot and the link between trajectories is on a pre-specified spanning tree of the graph, \( G \). This is called the tree strategy. Suppose we have a bipartite connected graph \( G \), and a spanning tree, \( S \), of \( G \).

**Proof that sectors do not starve in the tree strategy**

**Lemma 6.1.** If sector, \( S_i \) starves, then every sector starves.

**Proof.** By induction:

*Base case:* Sector \( S_i \) starves by assumption.

*Inductive Step:* If sector \( S_i \) starves, then sector \( S_{i-1} \) must be starving.

Assume that \( S_i \) is starving. Assume by way of contradiction that \( S_{i-1} \) is not starving. Then at some point, there will be a robot, \( r_j \) on \( S_{i-1} \). When the robot comes to the link between \( S_i \) and \( S_{i-1} \), then either it will meet another robot at the link, or it will not.

*Case 1:* \( r_j \) does not meet another robot at the link.

Then \( r_j \) will cross the link and traverse \( S_i \), thus, \( S_i \) was not starving. Contradiction.

*Case 2:* \( r_j \) meets another robot, \( r_k \), at the link between \( S_i \) and \( S_{i-1} \).
Then \( r_k \) will not cross the link because it saw \( r_j \), and thus \( r_k \) traverses \( S_i \). Thus \( S_i \) was not starving. Contradiction.

Thus, if one sector starves, then every sector starves, thus, there are no robots in the system. Thus, if there is at least one robot in the system, then no sector starves.

**Corollary 6.2.** If the tree strategy is used, and there is at least one robot in the system, then no sector starves.

**Modified Tree Strategy**

**Definition 6.3.** The modified tree strategy is exactly the same as the tree strategy with the following exception: Pick any leaf node of \( S \), and call this node the hole. If a robot sees that the hole is empty, the robot will enter the hole as usual, but once a robot is in the hole, it cannot traverse the link out of the hole unless it has been in the hole for \( 2n/\pi \) time without talking to another robot. If this happens, then the robot in the hole knows it is only robot left in the system. Thus, it leaves the hole and traverses the rest of the system by itself.

**Definition 6.4.** Let \( l_h \) be the communication link between the hole and the hole's parent on the spanning tree.

**Lemma 6.3.** If the hole is empty and there is at least one robot in the system, then the hole will be filled after some amount of time.

**Proof:** Consider the tree, \( S' \), that is \( S \) with the hole cut off. By the claim above, we know that no sector of \( S' \) starves. Thus, every sector of the hole's parent is traversed. Thus, some robot will see that the hole is empty and then cross \( l_h \) into it.
**Theorem 6.4.** If there is a robot, $r_h$, in the hole, and there is at least one other robot in the system, then no one starves.

*Proof:* Assume a robot, $r_h$, is in the hole. Now assume by way of contradiction that some robot, $r_i$, is starving, and $r_i$ is not $r_h$. If $r_h$ is not at $l_h$, go forward in time until $r_h$ is at $l_h$. If we cut off the hole from $S$, we still have a tree, $S'$. Since crossing a communication link is instantaneous, we can think of $r_h$ being on the hole’s parent at $l_h$. Thus, we know that $r_h$ and $r_i$ are on the same ring since $S'$ only has one ring. Since they are on the same ring, we know that $r_h$ is $2m\pi$ ahead of $r_i$. After $2\pi$ time, $r_h$ is in the same place, and $r_i$ is $2\pi$ closer to $r_h$, so they are now $2(m - 1)\pi$ away. After $2(m - 1)\pi$ more time, they will be distance 0 away, thus, they are communicating, so, neither $r_h$ nor $r_i$ are starving. Contradiction. Thus, no robot in the system is starving.
Chapter Seven: Conclusions and Future Work

In this paper, we expanded on the previously proposed theoretical model of cooperative drone formations. We identified how robots can become isolated and proved properties about isolation. We implemented an algorithm that finds how many failures are necessary for starvation to occur, and presented a new strategy to prevent isolation.

There is much to be expanded upon with this work. Is there an efficient algorithm to find the $k$-resilience of any graph, for any $2 \leq k$? What is the $k$-resilience for useful graphs other than grids? This is only the simplified theoretical model. Do some of these results still hold when we take away some of the base assumptions? For example, what happens when the trajectories are circles with different radii, or if the trajectories are not circles at all? These are only a few of the possible directions in which to continue this research.
Bibliography

Appendix A: Source Code

/*
 * OneResilience.java
 * Andy Brunner
 * June 2015
 * Calculates the 1-resilience of any graphs
 */
package com.abrunner;

import java.io.PrintWriter;

public class OneResilience {

    // returns the 1-resilience, given the number of nodes in the
    // ring structure
    public static int oneResilience(int n, Ring[] rings) {
        int steps = 0;
        int max = 0;

        // find how many robots must fail for a robot on ring q to fail
        for (int q = 0; q < rings.length; q++) {
            Ring R = rings[q];

            // finds first connection after robot
            int i = 0;
            while (R.cons[i].myRingDist > R.robots[0].distanceOnRing) {
                i++;
            }

            // distTrav is the distance from the robot's initial
            // position to the
            // link we are currently looking at
            int distTrav = 0;
            for (int j = 0; j < R.cons.length; j++) {
                // adds the distance between the connection the robot
                // is at and
                // the next connection
                if (j == 0) {
                    distTrav += trueMod(

        // if the robot is further than the farthest
        // the next connection the robot comes to will be
        R.cons[0]
            if (i == R.cons.length) {
                i = 0;
                break;
            }
        }

        // if the robot is further than the farthest
        // the next connection the robot comes to will be
        R.cons[0]
    }

}
R.cons[i].myRingDist -
R.robots[0].distanceOnRing),
        R.length);
    } else {
        if (i - 1 == -1) {
            // System.out.println("Adding " +
            // trueMod((R.cons[i].myRingDist -
            // R.cons[R.cons.length-1].myRingDist),
            R.length));
            distTrav += trueMod(
                R.cons[i].myRingDist -
            R.cons[R.cons.length - 1].myRingDist),
            R.length);
        } else {
            // System.out.println("Adding " +
            // trueMod((R.cons[i].myRingDist -
            // R.cons[{i-1}].myRingDist), R.length));
            distTrav += trueMod(
                (R.cons[i].myRingDist - R.cons[({i -
            1}]).myRingDist),
                R.length);
        }
    }

    // removes robot on the other ring
    // System.out.println("Looking at " +
    // trueMod((R.cons[i].otherRingDist - distTrav),
    // rings[R.cons[i].otherRingIndex].length) + " on ring
    " +
    // R.cons[i].otherRingIndex);
    rings[R.cons[i].otherRingIndex].removeRobotAt(trueMod(
        (R.cons[i].otherRingDist - distTrav),
        rings[R.cons[i].otherRingIndex].length));

    // origDist is the conflicting robot's initial position
    // on the other ring
    int origDist = trueMod((R.cons[i].otherRingDist -
        distTrav),
        rings[R.cons[i].otherRingIndex].length);
    int count = 1;

    // aDist is the R.length further back from origDist
    // int aDist = trueMod((origDist - count*R.length) ,
    // rings[R.cons[i].otherRingIndex].length);
    int aDist = origDist;
    if
    (rings[R.cons[i].otherRingIndex].isRobotAt(origDist)) {
        do {
            steps++;
            aDist = trueMod((origDist - count * R.length),
                R.length))
rings[R.cons[i].otherRingIndex].length);  
    // System.out.println("Looking at " + 
    trueMod(aDist ,  
    // rings[R.cons[i].otherRingIndex].length) + " 
    on ring "  
    // + R.cons[i].otherRingIndex);
    
    rings[R.cons[i].otherRingIndex].removeRobotAt(trueMod( 
    aDist ,  
    rings[R.cons[i].otherRingIndex].length));  
    count++;  
    } while (aDist != origDist);
    i = trueMod(i + 1, R.cons.length);
}

// find the robots on each ring that are still viable
int endCount = 0;
for (Ring S : rings) {
    endCount += S.getNumViableRobots();
}

// System.out.println("It takes " + (n-endCount) + 
// " failures to make a robot on ring " + q + " starve.");
if (endCount > max) {
    max = endCount;
}

for (Ring S : rings) {
    for (Robot r : S.robots) {
        r.viable = true;
    }
}

System.out.println(n + " " + steps);
return (n - max);

// a modulus operator that always returns a positive integer
public static int trueMod(int a, int b) {
    return (a % b + b) % b;
}

public static int helper(int a, int b) {
    int sum = a;
    int count = 1;
    while (trueMod(sum, b) != 0) {
        sum += a;
        count++;
    }
    return count;
}
public static void linesByTwo(int power) {
    int n = 2;
    PrintWriter writer = null;
    try {
        writer = new PrintWriter("lineResults.txt");
    } catch (Exception e) {
        e.printStackTrace();
    }

    long prevTime = 1;
    for (int i = 0; i < power; i++) {
        Ring[] theRing = TestGraphs.line(n);
        long startTime = System.nanoTime();
        int a = oneResilience(n, theRing);
        long endTime = System.nanoTime();
        // writer.println(n + " " + a + " " + (endTime - startTime));
        // System.out.println((endTime - startTime)/(double)prevTime);
        prevTime = (endTime - startTime);
        writer.println(n + " " + a + " " + (endTime - startTime));
        n = 2 * n;
    }
    writer.close();
}

public static void doublingGrids(int power) {
    int n = 2;
    PrintWriter writer = null;
    try {
        writer = new PrintWriter("squareGridResults.txt");
    } catch (Exception e) {
        e.printStackTrace();
    }

    for (int i = 0; i < power; i++) {
        Ring[] theRing = TestGraphs.grid(n, n);
        int a = 0;
        long startTime = System.nanoTime();
        a = oneResilience(n * n, theRing);
        long endTime = System.nanoTime();
        // System.out.println((endTime - startTime)/(double)prevTime);
        writer.println(n + "x" + n + " " + a + " " + (endTime - startTime));
        // System.out.println(i);
        n = 2 * n;
    }
    writer.close();
}

public static void main(String[] args) {
    /*
System.out.println("The 1-resilience of a 2x2 grid is ")
    oneResilience(4, TestGraphs.twoByTwo());
    System.out.println("The 1-resilience of the 4-node tree is ")
    oneResilience(4, TestGraphs.fourTree());
    System.out.println("The 1-resilience of the 6 node graph is ")
    oneResilience(6, TestGraphs.sixNodes());
    System.out.println("The 1-resilience of a 2-node line is ")
    oneResilience(2, TestGraphs.line(2));
    System.out.println("The 1-resilience of a 4-node line is ")
    oneResilience(4, TestGraphs.line(4));
    System.out.println("The 1-resilience of a 256-node line is ")
    oneResilience(256, TestGraphs.line(256));
*/

// linesByTwo(16);
// doublingGrids(10);
/*
int n=3; int max = 50; int[] ns = new int[max-2]; for(int i = 2; i <
    max; i++) { ns[i-2] = i; }
    */
System.out.println(oneResilience(n*n,TestGraphs.grid(n,n)));
for(int k = 0; k < ns.length; k++) { n= ns[k]; int sum = 0;
    for(int i = 0; i < n; i++) { sum += i; }
    oneResilience(n*n+sum,TestGraphs.hangingGrid(n)); }
*/

int m = 300;
for (int i = 0; i < m; i++) {
    int n = 8 * i + 4;
    oneResilience(n, TestGraphs.nRings(n));
}

/*
* Rings.java
* Andy Brunner
* June 2015
* A Connection is the communication link that bridges two nodes. It
* holds how far along it is on each ring it connects.
* Note that what is a communication link in the graph actually
* consists of two Connections.
*/
package com.abrunner;

public class Connection {
    int otherRingIndex;
    int myRingDist;
    int otherRingDist;

    public Connection() {
        this.otherRingDist = -1;
        this.myRingDist = -1;
        this.otherRingDist = -1;
    }

    public Connection(int ringIndex, int myRingDist, int otherRingDist) {
        this.otherRingIndex = ringIndex;
        this.myRingDist = myRingDist;
        this.otherRingDist = otherRingDist;
    }

    public String toString() {
        return "OtherRingIndex: " + otherRingIndex + "   myRingDist: "
              + myRingDist + "   otheRingDist: " + otherRingDist;
    }
}

package com.abrunner;

public class Robot {
    boolean viable;
    int distanceOnRing;

    public Robot(boolean aCrossedOff, int aDistanceOnRing) {
        viable = aCrossedOff;
        distanceOnRing = aDistanceOnRing;
    }
}

package com.abrunner;

public class Position {
    public boolean isHorz;
}
public int row;
public int col;
public boolean down;
public boolean right;
public int dist;
public static boolean[] covered;

public Position(boolean isHorz, int row, int col, boolean down, boolean right) {
    this.isHorz = isHorz;
    this.row = row;
    this.col = col;
    this.down = down;
    this.right = right;
    this.dist = 2;
}

public Position() {
    isHorz = true;
    row = 0;
    col = 0;
    down = true;
    right = true;
    dist = 2;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + col;
    result = prime * result + dist;
    result = prime * result + (down ? 1231 : 1237);
    result = prime * result + (isHorz ? 1231 : 1237);
    result = prime * result + (right ? 1231 : 1237);
    result = prime * result + row;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Position other = (Position) obj;
    if (col != other.col)
        return false;
    if (down != other.down)
return false;
if (isHorz != other.isHorz)
    return false;
if (right != other.right)
    return false;
if (row != other.row)
    return false;
return true;
}

public void print() {
    System.out.println((isHorz ? "horz" : "vert") + " row: " + row + " col: " + col + " down: " + down + " right: " + right);
}

public Position getNext(int n, int m) {
    Position result = new Position();
    Position p = this;

    // down right horz
    if (p.down && p.right && p.isHorz) {
        // normal
        if (p.row < n - 1) {
            result.row = p.row;
            result.col = p.col + 1;
            result.isHorz = false;
            result.down = true;
            result.right = true;
            result.dist = 2;
        }
    }
    // hit bottom wall
    else if (p.row == n - 1 && p.col < m - 2) {
        result.row = p.row;
        result.col = p.col + 1;
        result.isHorz = true;
        result.down = false;
        result.right = true;
        result.dist = 4;
    }
    // hit bottom right corner
    else if (p.row == n - 1 && p.col == m - 2) {
        result.row = p.row - 1;
        result.col = p.col + 1;
        result.isHorz = false;
        result.down = false;
        result.right = false;
        result.dist = 6;
    }
else {
   System.out.println("SOMETHING WENT WRONG 1");
}

} // up right horz
else if (!(p.down && p.right && p.isHorz)) {
   // normal
   if (p.row > 0) {
      result.row = p.row - 1;
      result.col = p.col + 1;
      result.isHorz = false;
      result.down = false;
      result.right = true;
      result.dist = 2;
   }

   // hit top wall
   else if (p.row == 0 && p.col < m - 2) {
      result.row = p.row;
      result.col = p.col + 1;
      result.isHorz = true;
      result.down = true;
      result.right = true;
      result.dist = 4;

      covered[col + 1] = true;
   }

   // hit top right corner
   else if (p.row == 0 && p.col == m - 2) {
      result.row = p.row;
      result.col = p.col + 1;
      result.isHorz = false;
      result.down = true;
      result.right = false;
      result.dist = 6;

      covered[col + 1] = true;
   }

   else {
      System.out.println("SOMETHING WENT WRONG 2");
   }
}

} // down left horz
else if (p.down && !p.right && p.isHorz) {

}
// normal
if (p.row < n - 1) {
    result.row = p.row;
    result.col = p.col;
    result.isHorz = false;
    result.down = true;
    result.right = false;
    result.dist = 2;
}

// hit bottom wall
else if (p.row == n - 1 && p.col > 0) {
    result.row = p.row;
    result.col = p.col - 1;
    result.isHorz = true;
    result.down = false;
    result.right = false;
    result.dist = 4;
}

// hit bottom left corner
else if (p.row == n - 1 && p.col == 0) {
    result.row = p.row - 1;
    result.col = p.col;
    result.isHorz = false;
    result.down = false;
    result.right = true;
    result.dist = 6;
}
else {
    System.out.println("SOMETHING WENT WRONG 3");
}

// down right vert
else if (p.down && p.right && !p.isHorz) {
    // normal
    if (p.col < m - 1) {
        result.row = p.row + 1;
        result.col = p.col;
        result.isHorz = true;
        result.down = true;
        result.right = true;
        result.dist = 2;
    }
    // hit right wall
    else if (p.row < n - 2 && p.col == m - 1) {
        result.row = p.row + 1;
        result.col = p.col;
result.isHorz = false;
result.down = true;
result.right = false;
result.dist = 4;
}

// hit bottom right corner
else if (p.row == n - 2 && p.col == m - 1) {
    result.row = p.row + 1;
    result.col = p.col - 1;
    result.isHorz = true;
    result.down = false;
    result.right = false;
    result.dist = 6;
}
else {
    System.out.println("SOMETHING WENT WRONG 4");
}

// up left horz
else if (!p.down && !p.right && p.isHorz) {
    // normal
    if (p.row > 0) {
        result.row = p.row - 1;
        result.col = p.col;
        result.isHorz = false;
        result.down = false;
        result.right = false;
        result.dist = 2;
    }

    // hit top wall
    else if (p.row == 0 && p.col > 0) {
        result.row = p.row;
        result.col = p.col - 1;
        result.isHorz = true;
        result.down = true;
        result.right = false;
        result.dist = 4;

        covered[col] = true;
    }

    // hit top left corner
    else if (p.row == 0 && p.col == 0) {
        System.out.println("THIS CASE SHOULD NOT HAVE BEEN
REACHED!");
        result.row = p.row;
        result.col = p.col;
    }
result.isHorz = false;
result.down = true;
result.right = true;
result.dist = 6;

covered[0] = true;
}
else {
  System.out.println("SOMETHING WENT WRONG 5");
}

// up right vert
else if (!p.down && p.right && !p.isHorz) {
  // normal
  if (p.col < m - 1) {
    result.row = p.row;
    result.col = p.col;
    result.isHorz = true;
    result.down = false;
    result.right = true;
    result.dist = 2;
  }

  // hit right wall
  else if (p.row > 0 && p.col == m - 1) {
    result.row = p.row - 1;
    result.col = p.col;
    result.isHorz = false;
    result.down = false;
    result.right = false;
    result.dist = 4;
  }

  // hit top right corner
  else if (p.row == 0 && p.col == m - 1) {
    result.row = p.row;
    result.col = p.col - 1;
    result.isHorz = true;
    result.down = true;
    result.right = false;
    result.dist = 6;

    covered[col] = true;
  }

  else {
    System.out.println("SOMETHING WENT WRONG 6");
  }
}
else if (p.down && !p.right && !p.isHorz) {
    // normal
    if (p.col > 0) {
        result.row = p.row + 1;
        result.col = p.col - 1;
        result.isHorz = true;
        result.down = true;
        result.right = false;
        result.dist = 2;
    }
    // hit left wall
    else if (p.row < n - 2 && p.col == 0) {
        result.row = p.row + 1;
        result.col = p.col;
        result.isHorz = false;
        result.down = true;
        result.right = true;
        result.dist = 4;
    }
    // hit bottom left corner
    else if (p.row == n - 2 && p.col == 0) {
        result.row = p.row + 1;
        result.col = p.col;
        result.isHorz = true;
        result.down = false;
        result.right = true;
        result.dist = 6;
    }
    else {
        System.out.println("SOMETHING WENT WRONG 7");
    }
}
else if (!p.down && !p.right && !p.isHorz) {
    // normal
    if (p.col > 0) {
        result.row = p.row;
        result.col = p.col - 1;
        result.isHorz = true;
        result.down = false;
        result.right = false;
        result.dist = 2;
    }
    // hit left wall
else if (p.row > 0 && p.col == 0) {
    result.row = p.row - 1;
    result.col = p.col;
    result.isHorz = false;
    result.down = false;
    result.right = true;
    result.dist = 4;
}

// hit top left corner
else if (p.row == 0 && p.col == 0) {
    result.row = p.row;
    result.col = p.col;
    result.isHorz = true;
    result.down = true;
    result.right = true;
    result.dist = 6;

    covered[col] = true;
}
else {
    System.out.println("SOMETHING WENT WRONG 8");
}

return result;
}

public static void main(String[] args) {
    covered = new boolean[4];
    Position p = new Position();
    for (int i = 0; i < 10; i++) {
        p = p.getNext(2, 4);
        p.print();
    }
}

/*
 * TestGraphs.java
 * Andy Brunner
 * June 2015
 * Gives the ring structures of a series of test graphs
 */
package com.abrunner;

import java.util.ArrayList;
import java.util.Arrays;
public class TestGraphs {

    public static Ring[] nRings(int n) {
        // bigRingCons = 3, 2,
        // 4, 4, 2, 2, 2, 2, 2, 4, 4, 2, 2, 2, 4, 4, 2, 2, 2, 4, 4, 2, 2, 2, 4, 4, ...
        int circLength = 8;
        int[] bigRingConLocs = new int[n + 2 * (n / 4 - 1)];
        bigRingConLocs[0] = 2;
        bigRingConLocs[1] = 4;
        int count = 0;
        int increment = 0;
        int length = bigRingConLocs[1];
        int i = 0;
        for (i = 0; i < 3 * (n / 4 - 1); i++) {
            switch (count % 6) { 
            case 0:
            case 1:
                increment = 4;
                break;
            case 2:
            case 3:
            case 4:
            case 5:
                increment = 2;
                break;
            default:
                System.out.println("Invalid increment");
                break;
            }
            count++;

            length += increment;
            bigRingConLocs[i + 2] = length;
        }

        increment = 6;
        length += increment;
        bigRingConLocs[i + 2] = length;
        i++;

        if (n % 8 == 4)
        {
            increment = 2;
            length += increment;
            bigRingConLocs[i + 2] = length;
            i++;
        }
    }
}
count = 0;
for (int j = 0; j < 3 * (n / 4 - 1); j++) {
    if (n % 8 == 4) {
        switch (count % 6) {
            case 0:
            case 1:
                increment = 4;
                break;
            case 2:
            case 3:
            case 4:
            case 5:
                increment = 2;
                break;
            default:
                System.out.println("Invalid increment");
                break;
        }
    }
    /* else if(n % 8 == 0) { switch(count % 6) { case 0: case 1: case 3: increment = 2; break; case 4: case 5: increment = 4; break; default: System.out.println("Invalid increment"); break; } */
    count++;
    length += increment;
    bigRingConLocs[i + j + 2] = length;
}
/* for(int j = 0; j < bigRingConLocs.length; j++) {
    System.out.println(bigRingConLocs[j]); }
*/

Connection[] bigRingCons = new Connection[bigRingConLocs.length];
Ring[] theRings = new Ring[n / 4 + 1];
for (i = 0; i < n / 4; i++) {
    Connection[] theCons = new Connection[4];
    theCons[0] = new Connection(0, 2, bigRingConLocs[3 * i]);
    bigRingCons[3 * i] = new Connection(i + 1, bigRingConLocs[3 * i], 2);

    theCons[1] = new Connection(0, 4,
        bigRingConLocs[bigRingConLocs.length - 3 * i - 2 + 1]);
bigRingCons[bigRingConLocs.length - 3 * i - 2 + 1] = new Connection( 
    i + 1, 
    bigRingConLocs[bigRingConLocs.length - 3 * i - 2 + 1], 4);

theCons[2] = new Connection(0, 10, 
    bigRingConLocs[bigRingConLocs.length - 3 * i - 2]);
bigRingCons[bigRingConLocs.length - 3 * i - 2] = new Connection( 
    i + 1, bigRingConLocs[bigRingConLocs.length - 3 * i - 2], 
    10);

theCons[3] = new Connection(0, 12, bigRingConLocs[1 + 3 * i]);
bigRingCons[1 + 3 * i] = new Connection(i + 1, 
    bigRingConLocs[1 + 3 * i], 12);

Robot[] theRobs = new Robot[2];
if (((i + 1) % 2) == 1) {
    for (int j = 0; j < theRobs.length; j++) {
        theRobs[j] = new Robot(true, 1 + 8 * j);
    }
} else if (((i + 1) % 2) == 0) {
    for (int j = 0; j < theRobs.length; j++) {
        theRobs[j] = new Robot(true, 7 + 8 * j);
    }
}

theRings[i + 1] = new Ring(theRobs, theCons, 16);

int bigRingLength = 8 * n - (n / 4 * 16);
for (int j = 0; j < n / 4 - 1; j++) {
    bigRingCons[2 + 3 * j] = new Connection(0, 
        bigRingConLocs[2 + 3 * j], 
        bigRingConLocs[bigRingConLocs.length - (2 + 3 * j) - 1]);
    bigRingCons[bigRingConLocs.length - (2 + 3 * j) - 1] = new Connection( 
        0, bigRingConLocs[bigRingConLocs.length - (2 + 3 * j) - 1], 
        bigRingConLocs[2 + 3 * j]);
}
Robot[] bigRobs = new Robot[bigRingLength / 8];
for (int j = 0; j < bigRingLength / 8; j++) {
    bigRobs[j] = new Robot(true, 1 + 8 * j);
}

Ring bigRing = new Ring(bigRobs, bigRingCons, 8 * n - (n / 4 * 16));
theRings[0] = bigRing;
System.out.println("big ring length is " + bigRing.length);
/*
 * for(i = 0; i < bigRing.cons.length; i++) { System.out.println(bigRing.cons[i]); } for(i = 0; i <
 * bigRing.robots.length; i++) { System.out.println("big robot at " +
 * bigRing.robots[i].distanceOnRing); }
 * */
*/

for(i = 0; i < n/4; i++) { Ring smRing = theRings[i+1]; System.out.println("Ring" + (i+1) + " has length " +
smRing.length);
 * for(int j = 0; j < smRing.cons.length; j++) { System.out.println(smRing.cons[j]); } for(int j = 0; j <
 * smRing.robots.length; j++) { System.out.println("sm robot at " +
 * smRing.robots[j].distanceOnRing); }
 * */

return theRings;
}

public static Ring[] hangingGrid(int n) {
// the horizontal links in the grid
Link[][] horz = new Link[n][n - 1];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n - 1; j++) {
        horz[i][j] = new Link();
    }
}

// the vertical links in the grid
Link[][] vert = new Link[n - 1][n];
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n; j++) {
        vert[i][j] = new Link();
    }
}

int numConsSet = 0;
int thisRing = -1;

// ArrayList<ArrayList<Connection>> uberCons = new
// ArrayList<ArrayList<Connection>>();
ArrayList<Ring> someRings = new ArrayList<Ring>();
Position p = new Position();
Position.covered = new boolean[n];
for (int i = 0; i < n; i++) {
    Position.covered[i] = false;
}
ArrayList<Connection> theCons = new ArrayList<Connection>();
while (numConsSet < 2 * ((n - 1) * n + (n - 1) * n)) {

    for (int i = 0; i < n; i++) {
        if (!Position.covered[i]) {
            thisRing++;
            if (i == n - 1 && i % 2 == 0) {
                p = new Position(false, 0, n - 1, true, false);
                p.dist = 4;
            } else if (i % 2 == 1) {
                p = new Position(true, 0, i - 1, true, false);
                p.dist = 2;
            } else if (i % 2 == 0) {
                p = new Position(true, 0, i, true, true);
                p.dist = 2;
            }
            break;
        }
    }

    int ringDist = p.dist;
    Position origP = p;
    // System.out.println("orgp is ");
    // origP.print();
    do {
        // System.out.println("p is ");
        // p.print();

            if (p.isHorz) {
                if (!horz[p.row][p.col].firstSet) {
                    horz[p.row][p.col].firstRing = thisRing;
                    horz[p.row][p.col].firstCon.myRingDist =
                    ringDist;
                horz[p.row][p.col].firstSet = true;
                theCons.add(horz[p.row][p.col].firstCon);
            } else {
                horz[p.row][p.col].secCon.otherRingIndex =
                horz[p.row][p.col].secCon.otherRingDist =
                horz[p.row][p.col].firstRing;
                horz[p.row][p.col].secCon.myRingDist =
                horz[p.row][p.col].secCon.otherRingDist =
                horz[p.row][p.col].firstCon.myRingDist;
                horz[p.row][p.col].firstCon.otherRingDist =
                thisRing;
                horz[p.row][p.col].firstCon.otherRingDist =
                ringDist;
                theCons.add(horz[p.row][p.col].secCon);
            }
        } else {
            if (!vert[p.row][p.col].firstSet) {
                vert[p.row][p.col].firstRing = thisRing;
vert[p.row][p.col].firstCon.myRingDist = ringDist;
vert[p.row][p.col].firstSet = true;
theCons.add(vert[p.row][p.col].firstCon);
} else {
  vert[p.row][p.col].secCon.otherRingIndex =
vert[p.row][p.col].firstRing;
  vert[p.row][p.col].secCon.myRingDist =
ringDist;
  vert[p.row][p.col].secCon.otherRingDist =
vert[p.row][p.col].firstCon.myRingDist;
  vert[p.row][p.col].firstCon.otherRingIndex =
thisRing;
  vert[p.row][p.col].firstCon.otherRingDist =
ringDist;
  theCons.add(vert[p.row][p.col].secCon);
}
}
numConsSet++;

p = p.getNext(n, n);
ringDist += p.dist;
} while (!p.equals(origP));

// uberCons.add(theCons);
// System.out.println(theCons);

// System.out.println(theCons.size());
ArrayList<Robot> someRobs = new ArrayList<Robot>();

if (someRings.size() == n - 1 && (((n - 1) % 2) == 0)) {
  ringDist -= 4;
  /*
   * for(int i = 0; i*8 < ringDist;i++) {
   * Robot(true, i*8 +7); }
   */
} else {
  ringDist -= 2;
}
// System.out.println(ringDist);
for (int i = 0; i * 8 < ringDist; i++) {
  someRobs.add(new Robot(true, i * 8 +1));
  // System.out.println("Adding robot at " + (i*8+1));
}

Robot[] theRobs = someRobs.toArray(new
Robot[someRobs.size()]);

someRings.add(new Ring(theRobs, null, ringDist));
int a = theCons.size() / someRings.size();
for (int i = 0; i < someRings.size(); i++) {
    Connection[] cons = theCons.subList(i * a, (i + 1) * a).toArray(
        new Connection[a]);
    someRings.get(i).cons = cons;
}

for (int i = 0; i < someRings.size(); i++) {
    int oldConLength = someRings.get(i).cons.length;
    Connection[] newCons = new Connection[oldConLength + 2 * i];
    for (int j = 0; j < oldConLength; j++) {
        newCons[j] = someRings.get(i).cons[j];
    }
    for (int j = 0; j < i; j++) {
        newCons[oldConLength + j] = new Connection(i,
            someRings.get(i).length - 1 + 8 * i - 4 * j,
            someRings.get(i).length - 1 + 4 * j);
        newCons[oldConLength + 2 * i - j - 1] = new Connection(i,
            someRings.get(i).length - 1 + 4 * j,
            someRings.get(i).length - 1 + 8 * i - 4 * j);
    }
    someRings.get(i).cons = newCons;
    
    /*
     * ArrayList<Robot> newRobs = new ArrayList<Robot>(Arrays.asList(someRings.get(i).robots));
     * int lastRobPos = newRobs.get(newRobs.size() - 1).distanceOnRing;
     * for(int j = 0; j < i; j++) { newRobs.add(new Robot(true,
     * lastRobPos +8*i)); }
     */
    Robot[] newRobs = new Robot[someRings.get(i).robots.length + i];
    int lastRobPos =
    someRings.get(i).robots[someRings.get(i).robots.length - 1].distanceOnRing;
    for (int j = 0; j < someRings.get(i).robots.length; j++) {
        newRobs[j] = someRings.get(i).robots[j];
    }
    for (int j = 0; j < i; j++) {
        newRobs[someRings.get(i).robots.length + j] = new Robot(true,
            lastRobPos + 8 * (j + 1));
    }
someRings.get(i).robots = newRobos;
someRings.get(i).length = someRings.get(i).length + 8 * i;
}
return someRings.toArray(new Ring[someRings.size()]);
} public static Ring[] grid(int n, int m) {
// the horizontal links in the grid
Link[][] horz = new Link[n][m - 1];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m - 1; j++) {
        horz[i][j] = new Link();
    }
}
// the vertical links in the grid
Link[][] vert = new Link[n - 1][m];
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < m; j++) {
        vert[i][j] = new Link();
    }
}
int numConsSet = 0;
int thisRing = -1;

// ArrayList<ArrayList<Connection>> uberCons = new
// ArrayList<ArrayList<Connection>>();
ArrayList<Ring> someRings = new ArrayList<Ring>();
Position p = new Position();
Position.covered = new boolean[m];
for (int i = 0; i < m; i++) {
    Position.covered[i] = false;
}
ArrayList<Connection> theCons = new ArrayList<Connection>();
while (numConsSet < 2 * ((n - 1) * m + (m - 1) * n)) {
    for (int i = 0; i < m; i++) {
        if (!Position.covered[i]) {
            thisRing++;
            if (i == m - 1 && i % 2 == 0) {
                p = new Position(false, 0, m - 1, true, false);
                p.dist = 4;
            } else if (i % 2 == 1) {
                p = new Position(true, 0, i - 1, true, false);
                p.dist = 2;
            } else if (i % 2 == 0) {
                p = new Position(true, 0, i, true, true);
                p.dist = 2;
            }
        }
        break; 60
```java
int ringDist = p.dist;
Position origP = p;
// System.out.println("orgp is ");
// origP.print();
do {
    // System.out.println(someRings.size());
    // p.print();
    if (p.isHorz) {
        if (!horz[p.row][p.col].firstSet) {
            horz[p.row][p.col].firstRing = thisRing;
            horz[p.row][p.col].firstCon.myRingDist = ringDist;
            horz[p.row][p.col].firstSet = true;
            theCons.add(horz[p.row][p.col].firstCon);
        } else {
            horz[p.row][p.col].secCon.otherRingIndex = 
            horz[p.row][p.col].firstRing;
            horz[p.row][p.col].secCon.myRingDist = ringDist;
            horz[p.row][p.col].secCon.otherRingDist = 
            horz[p.row][p.col].firstCon.myRingDist;
            horz[p.row][p.col].secCon.otherRingIndex = 
            thisRing;
            horz[p.row][p.col].firstCon.otherRingDist = 
            ringDist;
            theCons.add(horz[p.row][p.col].secCon);
        }
    } else {
        if (!vert[p.row][p.col].firstSet) {
            vert[p.row][p.col].firstRing = thisRing;
            vert[p.row][p.col].firstCon.myRingDist = ringDist;
            vert[p.row][p.col].firstSet = true;
            theCons.add(vert[p.row][p.col].firstCon);
        } else {
            vert[p.row][p.col].secCon.otherRingIndex = 
            vert[p.row][p.col].firstRing;
            vert[p.row][p.col].secCon.myRingDist = ringDist;
            vert[p.row][p.col].secCon.otherRingDist = 
            vert[p.row][p.col].firstCon.myRingDist;
            vert[p.row][p.col].secCon.otherRingIndex = 
            thisRing;
            vert[p.row][p.col].firstCon.otherRingDist = 
            ringDist;
            theCons.add(vert[p.row][p.col].secCon);
        }
    }
    // System.out.println(someRings.size());
    // p.print();
}
```
numConsSet++;

p = p.getNext(n, m);
ringDist += p.dist;
} while (!p.equals(origP));

// uberCons.add(cons);
// System.out.println(cons);

ArrayList<Robot> someRobs = new ArrayList<Robot>();

if (someRings.size() == m - 1 && ((m - 1) % 2) == 0) {
    ringDist -= 4;
    /*
     * for(int i = 0; i*8 < ringDist;i++) {
     *     someRobs.add(new Robot(true, i*8 +7)); }
     */
} else {
    ringDist -= 2;
}
// System.out.println(ringDist);
for (int i = 0; i * 8 < ringDist; i++) {
    someRobs.add(new Robot(true, i * 8 + 7));
    // System.out.println("Adding robot at " + (i*8+1));
}

Robot[] theRobs = someRobs.toArray(new Robot[someRobs.size()]);

someRings.add(new Ring(theRobs, null, ringDist));

int a = theCons.size() / someRings.size();
for (int i = 0; i < someRings.size(); i++) {
    Connection[] cons = theCons.subList(i * a, (i + 1) * a).toArray(new Connection[a]);
    someRings.get(i).cons = cons;
}

return someRings.toArray(new Ring[someRings.size()]);

// returns the ring structure of the graph from Figure 9 in the thesis
public static Ring[] sixNodes() {
// Ring 0 starts at the top of the top left node
// Ring 1 starts at the top of the top right node
// 2x2 grid
Connection[] cons0 = new Connection[4];
cons0[0] = new Connection(1, 25, 25);
cons0[1] = new Connection(1, 50, 350);
cons0[2] = new Connection(1, 125, 325);
cons0[3] = new Connection(1, 150, 50);

Connection[] cons1 = new Connection[8];
cons1[0] = new Connection(0, 25, 25);
cons1[1] = new Connection(0, 50, 150);
cons1[2] = new Connection(1, 100, 300);
cons1[3] = new Connection(1, 150, 250);
cons1[4] = new Connection(1, 250, 150);
cons1[5] = new Connection(1, 300, 100);
cons1[6] = new Connection(0, 325, 125);
cons1[7] = new Connection(0, 350, 50);

Robot[] robots0 = new Robot[2];
robots0[0] = new Robot(true, 1);
robots0[1] = new Robot(true, 101);

Robot[] robots1 = new Robot[4];
robots1[0] = new Robot(true, 1);
robots1[1] = new Robot(true, 101);
robots1[2] = new Robot(true, 201);
robots1[3] = new Robot(true, 301);

Ring[] rings = new Ring[2];
rings[0] = new Ring(robots0, cons0, 200);
rings[1] = new Ring(robots1, cons1, 400);

// Returns the ring structure of the simple tree in Figure 10 of
// the thesis
// It is a root node with three children
public static Ring[] fourTree() {
  // the root node has one child directly left of it, one child
directly
  // below, and one child directly to the right of it
  // the ring starts at the top of the root node
  Connection[] theCons = new Connection[6];
  theCons[0] = new Connection(0, 25, 125);
  theCons[1] = new Connection(0, 125, 25);
  theCons[2] = new Connection(0, 150, 250);
  theCons[3] = new Connection(0, 250, 150);
  theCons[4] = new Connection(0, 275, 375);
  theCons[5] = new Connection(0, 375, 275);
```java
Robot[] theRobots = new Robot[4];
for (int i = 0; i < 4; i++) {
    theRobots[i] = new Robot(true, 100 * i + 25);
}

Ring[] theRings = new Ring[1];
theRings[0] = new Ring(theRobots, theCons, 400);
return theRings;
}

// returns the ring structure of a line of length n
public static Ring[] line(int n) {
    // Ring starts at leftmost point of leftmost line
    Connection[] theCons = new Connection[2 * (n - 1)];
    for (int i = 0; i < n - 1; i++) {
        theCons[i] = new Connection(0, 50 * (i + 1), 100 * n - 50 * (i + 1));
    }
    theCons[2 * (n - 1) - i - 1] = new Connection(0, 100 * n - 50 * (i + 1), 50 * (i + 1));

    Robot[] theRobots = new Robot[n];
    for (int i = 0; i < n; i++) {
        theRobots[i] = new Robot(true, 100 * i + 25);
    }

    Ring[] theRings = new Ring[1];
    theRings[0] = new Ring(theRobots, theCons, 100 * n);
    return theRings;
}

// returns the ring structure of a 2x2 grid
public static Ring[] twoByTwo() {
    // Ring 0 starts at the top of the top left node
    // Ring 1 starts at the top of the top right node
    Connection[] cons0 = new Connection[4];
    cons0[0] = new Connection(1, 25, 25);
    cons0[1] = new Connection(1, 50, 150);
    cons0[2] = new Connection(1, 125, 125);
    cons0[3] = new Connection(1, 150, 50);

    Connection[] cons1 = new Connection[4];
    cons1[0] = new Connection(0, 25, 25);
    cons1[1] = new Connection(0, 50, 150);
    cons1[2] = new Connection(0, 125, 125);
    cons1[3] = new Connection(0, 150, 50);
```
Robot[] robots0 = new Robot[2];
robots0[0] = new Robot(true, 0);
robots0[1] = new Robot(true, 100);

Robot[] robots1 = new Robot[2];
robots1[0] = new Robot(true, 0);
robots1[1] = new Robot(true, 100);

Ring[] rings = new Ring[2];
rings[0] = new Ring(robots0, cons0, 200);
rings[1] = new Ring(robots1, cons1, 200);

return rings;
}

public class Ring {

Robot[] robots;
Connection[] cons;
int length;

public Ring(Robot[] robots, Connection[] cons, int length) {
    this.robots = robots;
    this.cons = cons;
    this.length = length;
}

// sets the robot at this position to be not viable
public void removeRobotAt(int loc) {
    for (Robot r : robots) {
        if (r.distanceOnRing == loc) {
            if (!r.viable) {
                r.viable = false;
                // System.out.println("Removing robot at " + loc);
            } else {
                // System.out.println("Robot already removed at " +
                loc);
            }
        }
    }
    return;
}

}
System.out.println("ERROR: Robot not at " + loc + ". Your rings may not be well formed.");
}

public boolean isRobotAt(int loc) {
    for (Robot r : robots) {
        if (r.distanceOnRing == loc) {
            return true;
        }
    }
    return false;
}

// returns the number of viable robots
public int getNumViableRobots() {
    int count = 0;
    for (Robot r : robots) {
        if (r.viable) {
            count++;
        }
    }
    return count;
}

package com.abrunner;

public class Link {

    public Connection firstCon;
    public Connection secCon;
    public boolean firstSet = false;
    public int firstRing;
    public int firstRingDist;

    public Link() {
        firstCon = new Connection();
        secCon = new Connection();
        firstSet = false;
        firstRing = -1;
        firstRingDist = -1;
    }

}