Abstract

In collective transport, a search and rescue multi-robot formation must preserve its shape in order to safely carry an object from one location to another. In this thesis we focus on *persistent acyclic leader-follower formations* in which robots adhere to local pairwise distance constraints in order to maintain the formation as a whole. We rely on combinatorial *rigidity* and *persistence theory*, where robots are modeled as vertices and distance constraints as edges. In order to identify the formations that perform best, we define a property called *diameter* using the concept of waves. We hypothesize that diameter impacts the performance of a moving formation. By generating persistent acyclic leader-follower formations, we design a set of experiments both on simulation and on a multi-robot hardware platform. The results and analysis support our hypothesis that smaller diameter is related to better performance.

Mount Holyoke College
Senior Thesis

Coordination of Persistent Multi-Robot Formations

Formations

by
Haya Diwan

Supervised by
Prof. Audrey St. John

Presented to the faculty of Mount Holyoke College in partial fulfillment of the requirements for the degree of Bachelor of Arts with Honors

May 2021

Acknowledgements

I would first like to thank my thesis advisor Audrey St. John for constantly pushing me to realize my potential and for being an incredible role model for the professor I would like to become.

I would like to thank Prof. Jessica Sidman for being a great mentor and for inspiring me to pursue research in a difficult field.

I would like to thank the Mount Holyoke computer science and math departments for providing invaluable support over the last four years.

To my family and friends, thank you for supporting me through this experience. I could not have done it without you. I would particularly like to thank my sister Sophya for always being my greatest supporter and motivator.

Contents

List of Figures

List of Tables

1 Introduction

In *multi-robot formations*, robots work together to complete tasks that cannot be accomplished by a single human or robot [20]. Consider the task of moving a piece of rubble from a disaster site. If the rubble is too heavy or too dangerous for humans, using multi-robot formations is a possible solution.

During this *collective transport* task [13], it is crucial that robots stay in formation to ensure the piece of rubble is transported safely. If a robot moves out of formation, the rubble may fall and worsen the site. This prompts the question: how can we ensure that robots stay in their desired formation?

In this thesis we work with *persistent leader-follower formations*. These formations follow a *decentralized* control pattern in which each member of the formation follows its own predetermined *constraints* [7].

Contributions. The focus of this thesis is to determine how combinatorial properties of persistent acyclic leader-follower formations impact accuracy of moving formations. We only consider acyclic formations which allow us to define a concept called "diameter", which we define formally in Section 2.5.

We state our hypothesis below.

Hypothesis. *Formations with a smaller diameter are more likely to accurately maintain constraints during collective movement.*

To evaluate our hypothesis, we used Georgia Tech's Robotarium platform [19] which allows for Python and hardware experiments.

Related Work. In this thesis we focus on a control pattern in which robots only communicate with each other throughout movement. This idea is explored in [12, 17] where there is direct communication between two robots; one robot dictates movement patterns and the other follows movement instruction.

Methods which expand on inter-robot communication within multi-robot formations in collective transport are [1, 18, 6]. In [1], robots compute a convex hull that surrounds an obstacle-free region in which a formation can be made for movement. In this thesis, we identify one robot as a leader and another as a co-leader. Sources [18, 6] add additional leaders for robustness.

Examples of projects in which robots complete tasks without inter-robot communication are [13, 22, 23, 10]. In [10, 22] people or larger systems are involved.

In this project, we work with *persistent leader-follower formations*, which have been applied to formation control previously in [21, 4]. Examples in which leader-follower formations are used without persistence are [6, 18, 22].

Hendrickx et al. introduced persistence theory in [7, 8] which is founded in rigidity theory [5, 11]. In order to generate the formations used in simulation, we use Henneberg constructions as described in [15]. We define the diameter of a formation using the concept of waves [2].

Our work builds on work done in [2, 3] using persistent leader-follower formations which focus on the problem of collective transport. While [2, 3] focus on the concept of redundancy in formations, this thesis examines the correlation between formation diameter and accuracy in movement.

To run our experiments, we used the Georgia Tech Robotarium platform [19], which was developed for multi-robot simulations in a Python and hardware environment.

Structure. Section 2 details the theoretical background necessary to understand our hypothesis. We present an approach in Section 3 that generates persistent acyclic leader-follower formations. The details of our simulations and the data collected are described in Section 4. Section 5 gives the setup for analysis of the results of our simulations.

2 Preliminaries

We will start with the background in graph, rigidity and persistence theory that our approach is built on.

2.1 Graph Theory

In this thesis we will be working with two types of graphs: *directed* and *undirected*.

An *undirected graph* is defined on a set of vertices connected via edges. Formally, an undirected graph is given by $G = (V, E_G)$, where $V = \{v_1, ..., v_n\}$ is a set of vertices and E_G is a set of pairs of vertices, representing edges. We denote an edge in G as e_{ij} where $i < j$. An example of an undirected graph is shown in Figure 1a.

(a) Undirected (b) Directed

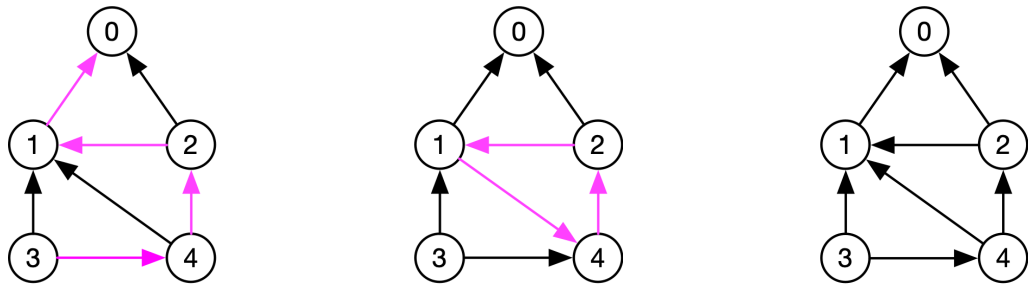
Figure 1: Example of directed and undirected graphs. The graph on the left is undirected, and the graph on the right is directed (arrowheads depict direction).

A *directed graph* is defined on a set of vertices connected via directed edges. We denote directed graphs as $H = (V, E_H)$, where $V = \{v_1, ...v_n\}$ is a set of vertices and E_H is a set of directed edges. We denote an edge from v_i to v_j in a directed graph as $\overrightarrow{e_{ij}}$. For both directed and undirected graphs we denote vertices as v_i where $v_i \in V$. We denote the number of vertices in a graph as n. The graph pictured in Figure 1b is directed; arrowheads depict the directions.

Every directed graph has a corresponding *underlying undirected graph*. Take some directed graph $H = (V, E_H)$. Then H has an underlying undirected graph $G = (V, E_G)$ such that $\forall e_{ij} \in E$, $\exists \overrightarrow{e_{ij}}$ or $\overrightarrow{e_{ji}} \in E_H$. An example of this can be seen in Figure 1 where Figure 1a is the underlying undirected graph of Figure 1b.

The *degree* of a vertex v, $d(v)$, is the number of edges incident to v. For undirected graphs, we count all edges incident to v and for directed graphs we count only the outgoing edges of v. For example, in Figure 1a, the degrees are $2, 4, 3, 2, 3$ for vertices v_0, v_1, v_2, v_3, v_4 respectively. In Figure 1b, the degrees are $0, 1, 2, 2, 2$ for vertices v_0, v_1, v_2, v_3, v_4 respectively.

A *simple path* between vertices v_1 and v_k is a sequence of distinct vertices $v_1, ..., v_k$ with edges $\overrightarrow{e_{12}}, \overrightarrow{e_{23}}, ..., \overrightarrow{e_{(k-1)(k)}} \in E$. An example of a path from v_3 to v_0 is shown in pink in Figure 2a. A *cycle* is a sequence of vertices $v_1, ..., v_k$ where $v_1 = v_k$. For example, Figure 2b contains a cycle. We call a directed graph without cycles *acyclic*. Refer to Figure 2c for a directed graph with no cycles, or a *directed acyclic graph*.



(a) Path from v_3 to v_0 (b) Cycle (c) Acyclic

Figure 2: Examples of paths and directed graphs with and without cycles. (a) shows an example of a path of distinct vertices from v_3 to v_0 $(v_3, v_4, v_2, v_1, v_0)$. (b) has a cycle because there exists a sequence of vertices such that the first and last vertex are the same (v_1, v_4, v_2, v_1). (c) is acyclic because the graph contains no cycles.

.

2.2 Rigidity Theory

In this section, we provide the foundation of rigidity theory necessary to understand persistence theory; for a more rigorous coverage see [5, 11].

We work with bar-and-joint structures and assume generic embeddings[1]. Intuitively, a graph is *rigid* if there is no way to alter any pairwise distances while maintaining given edge lengths. Conversely, a graph is flexible if the pairwise distances between vertices can be changed. For example, the graphs shown in Figures 3a and 4a are rigid, as there is no way to alter any pairwise distances. Conversely, in Figure 3b when e_{01} is moved, the pairwise distance between v_0 and v_3 changes and the pairwise distance between v_1 and v_2 changes. Note that the length of all edges in the graph remain the same, but edges are still able to move. Therefore, we can label this graph as flexible.

[1]The rigidity of such structures depends solely on their combinatorics. Therefore, we will work with graphs, whose edges represent bars and vertices represent joints.

(a) Rigid (b) Flexible

Figure 3: Rigid vs. flexible graphs. (a) is rigid because there is no way to alter any pairwise distances in the graph. (b) is flexible because if e_{01} is moved, the pairwise distance between v_0 and v_3 changes and the pairwise distance between v_1 and v_2 changes.

We now proceed to define rigidity more formally. Take some graph $G = (V, E_G)$ with n vertices and m edges. We formally define a 2-dimensional *bar-and-joint* framework denoted $G(p)$ to assign $p = (p_1, p_2, ...p_n) \in (\mathbb{R}^2)^n \ \ \forall v \in V$. The set of points p represents coordinates for all vertices in $G(p)$ such that all edge lengths are satisfied. We call p the realization of all $n \in G$, $G \notin \mathbb{R}^2$.

Given the framework described above, we define a distance equation $d : V \to \mathbb{R}^2$ as follows:

$$d(ij) = ||p(i) - p(j)|| \quad \forall e_{ij} \in E \tag{1}$$

We define two frameworks $G(p), G(q)$ as *equivalent* if their corresponding edge lengths are the same. We use this definition to say that if two frameworks $G(p), G(q)$ are equivalent and there exists a neighborhood N_p of p such that $q \in N_p$ then p is *congruent* to q. Equation 2 demonstrates this congruence.

$$||p(i) - p(j)|| = ||q(i) - q(j)|| \quad \forall e_{ij} \in E \tag{2}$$

Definition 2.1. *If $\forall q \in N_p$, (G, p) is congruent to (G, q), we can say that the framework F is (locally) rigid and otherwise flexible.*

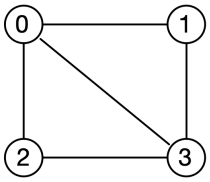
Given the definitions of rigid and flexible, we state the definition of *minimally rigid*.

Definition 2.2. *A graph $G = (V, E_G)$ is minimally rigid if*

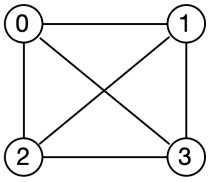
1. *G is rigid*
2. *$G - e$ is flexible $\forall\ e \in E$*

Note that due to the first condition of Definition 2.2, any flexible graph is not minimally rigid. Observe that the removal of any edge from the graph pictured in Figure 4a will result in a flexible graph. Therefore, Figure 4a is *minimally rigid*. Conversely, Figure 4b remains rigid even upon the removal of an edge. For example, Figure 5 shows the six different resulting graphs upon the removal of one edge from Figure 4b. We see that all six resulting frameworks are minimally rigid. Therefore, Figure 4b is *over-constrained*.

[2]Note: we abuse notation in equation 1 to view p as a function mapping vertex indices to points



(a) Minimally Rigid



(b) Over-Constrained

Figure 4: Minimally rigid (upon removal of any edge, graph is flexible) vs over-constrained (upon removal of an edge graph is still rigid)

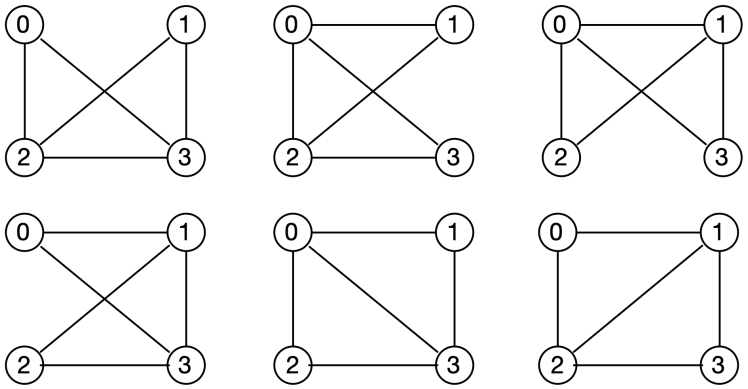


Figure 5: Upon removal of each of the 6 edges from Figure 4b, graph is still rigid

We use minimally rigid graphs to identify the fewest number of edges required for rigidity. This in turn helps to reduce the number of distance constraints required for formation control.

2.3 Henneberg Moves

We now describe inductive construction steps called *Henneberg moves* that characterize minimal rigidity in the plane. The definitions in this section are taken from [15]. There are two types of inductive moves: *Henneberg 1 (H1)* and *Henneberg 2 (H2)*.

Definition 2.3. *For some $G = (V, E_G)$ where $\{v_a, v_b, v_c\} \subseteq V$,*

1. *(vertex addition/Henneberg 1 (H1)) Add vertex v_c with $d(v) = 2$ and neighbors $= N(v) = \{v_a, v_b\}$ where $v_a \neq v_b$*
2. *(edge splitting/Henneberg 2 (H2)) remove edge e_{ab} and add vertex v with $d(v) = 3$ and $N(v) = \{v_a, v_b, v_c\}$ where $v_c \neq v_a$ and $v_c \neq v_b$*

An example of this type of construction is shown in Figure 6. Given the starting edge e_{01}, we see that v_2 is added via an H1 move. First, v_2 is added to the framework, and then is connected to two existing vertices $\{v_0, v_1\}$. This same process is seen with the addition of v_3 and v_4, where v_3 is connected to $\{v_0, v_1\}$ and v_4 is connected to $\{v_2, v_3\}$. The last step of the construction is an H2 move. First, e_{03} is removed, then v_5 is added to the framework. Then, v_5 is connected to the endpoints of the removed edge, $\{v_0, v_3\}$ and a new edge e_{45} is added.

We are particularly interested in Henneberg moves due to its relationship with the concept of *(2,3)-tightness*. We use the definition of *(2,3)-sparse* to define (2,3)-tightness.

Definition 2.4. *For some graph $G = (V, E_G)$, we say G is (2,3)-sparse if for every subgraph $G' = (V_{G'}, E_{G'})$ where $|E_{G'}| \geq 1$, $|E_{G'}| \leq 2|V_{G'}| - 3$.*

Definition 2.5. *For some graph $G = (V, E_G)$, we say G is (2,3)-tight if*

1. *G is (2,3)-sparse*
2. *$|E_G| = 2|V| - 3$*

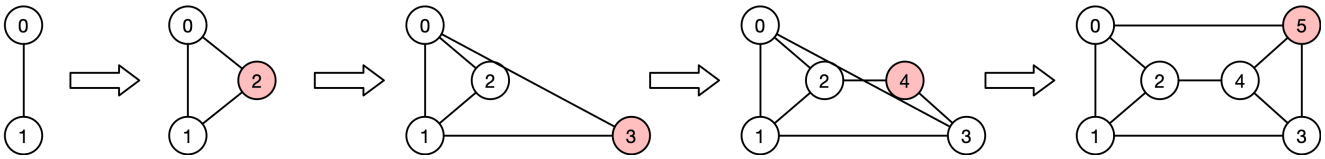


Figure 6: Henneberg construction example using H1 and H2 moves. $\{v_2, v_3, v_4\}$ added via H1, v_5 added via H2.

Figure 7 contains the subgraphs with $|E| \geq 1$ of Figure 3a. We see that all subgraphs adhere to the conditions of Definition 2.4. For example, for all subgraphs with $|E| = 1$, $|E| \leq 2|V| - 3 = 2(2) - 3 = 1$. Therefore, we can say that Figure 3a is (2,3)-sparse. Given that Figure 3a is (2,3)-sparse, the first condition of Definition 2.5 is satisfied. We also note that $|E| \leq 2|V| - 3 = 2(3) - 3 = 3$, which satisfies the second condition of Definition 2.5. Therefore, we can say that Figure 3a is (2,3)-tight.

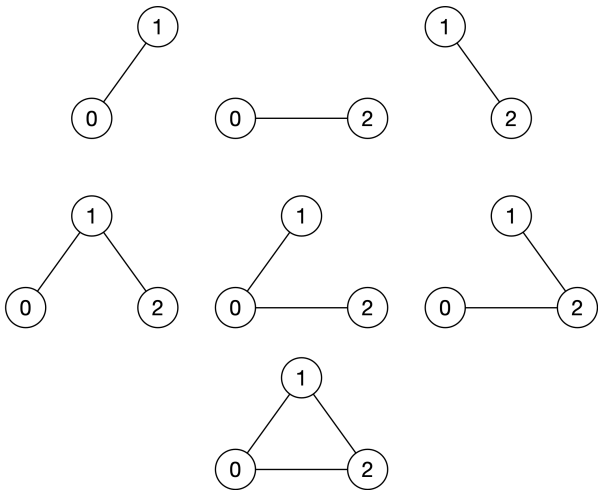


Figure 7: Subgraphs of figure 3a with $|E| \geq 1$. Given that each subgraph satisfies $|E| \leq 2|V| - 3$, we can say figure 3a is (2,3)-sparse.

Theorem 2.6. *(Henneberg [9], Laman [14]). A graph G is (2,3)-tight if and only if it can be derived recursively from a single edge by vertex additions (H1) and edge splitting (H2).*

We now connect (2,3)-tight to minimally rigid graphs.

Theorem 2.7. *(Pollaczek-Geiringer [16], Laman [14]). A graph G is generically minimally rigid in the plane if and only if G is (2, 3)-tight.*

Recall that the graph in Figure 6 is constructed inductively using Henneberg moves. Therefore, by Theorem 2.6 we can say that the graph is (2,3)-tight. Given that the graph in Figure 6 is (2,3)-tight, using Theorem 2.7, we can say that it is minimally rigid. In this thesis we will specifically be focusing on graphs constructed inductively using H1 moves.

It will be important for our project that there may be more than one sequence of only H1 moves that can construct a given minimally rigid graph. For example, in Figure 8, the resulting minimally rigid graph from both H1 sequences is the same. However, the sequence of H1 moves applied to construct the graph is different. For example, the sequence of H1 moves for the first graph is $\{v_1, v_3\}$, while the sequence for the second graph is $\{v_3, v_1\}$.

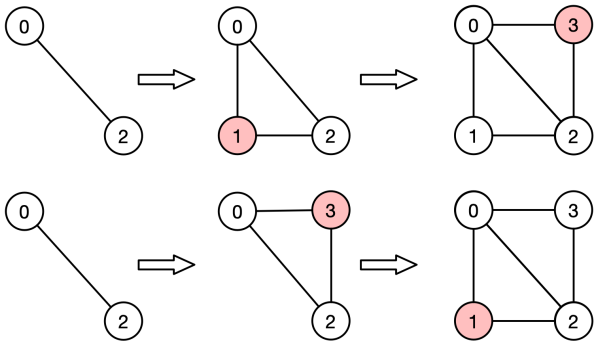


Figure 8: Example of different H1 construction sequences. H1 sequence for top graph is $\{v_1, v_3\}$. H1 sequence for bottom graph is $\{v_3, v_1\}$.

We note that if G has an H1 construction, not every edge may be used as the first edge in an H1 construction of G. For example, the graph in Figure 9 can be constructed using only H1 moves given starting edge e_{01} (as seen in Figure 10). However, if we take starting edge e_{24} or e_{34}, we are unable to recreate Figure 9 using only H1 moves. In order to perform an H1 move, there must be two previously existing vertices in the graph that have at least one shared neighbor. For example, v_0 and v_1 have neighbors $\{v_2, v_3\}$. However, vertices v_2 and v_4 have no neighbors, so an H1 move cannot be performed. Similarly with v_3 and v_4, the two vertices do not share any neighbors so no H1 moves can be performed. Note that Figure 8 is an example of a graph in which every edge may be used as the first edge in an H1 construction.

2.4 Persistence Theory

The relationship between graph theory and multi-robot formations can be thought of in terms of robots (vertices) and distance constraints (edges). When we apply rigidity theory to multi-robot formations, robots must maintain the distance constraints between them. Therefore, the undirected edges of rigidity theory result in redundancy of constraints. For example, in Figure 4a, v_0 is working to maintain the constraint given by e_{01} and v_1 is working to maintain the same constraint. We can reduce the number constraints by applying directed edges in persistence theory where robots only adhere to the constraints given by their outgoing edges. For example, in Figure 11a, v_0 is the only vertex that maintains the constraint given by outgoing edge $\overrightarrow{e_{01}}$.

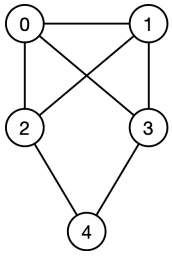


Figure 9: H1 framework that cannot be constructed from all starting edges. Figure 10 shows a construction from starting edge e_{01}. Starting edges $\{e_{34}, e_{24}\}$ have no possible H1 sequence ending in the desired graph.

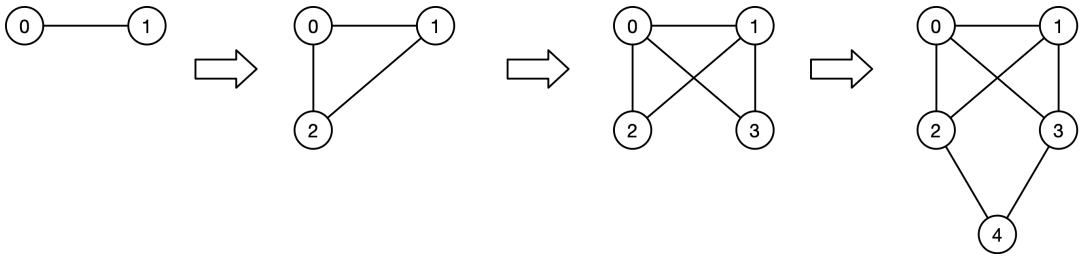


Figure 10: Henneberg 1 construction example of figure 9 from starting edge e_{01}.

In this section we review the foundations of persistence theory as set forth by Hendrickx et al. [7, 8]. Let p be an embedding of some directed graph $H = (V, E_H)$. We define a formation as the pair (H, p). Intuitively, a formation is *persistent* if it adheres to the following conditions:

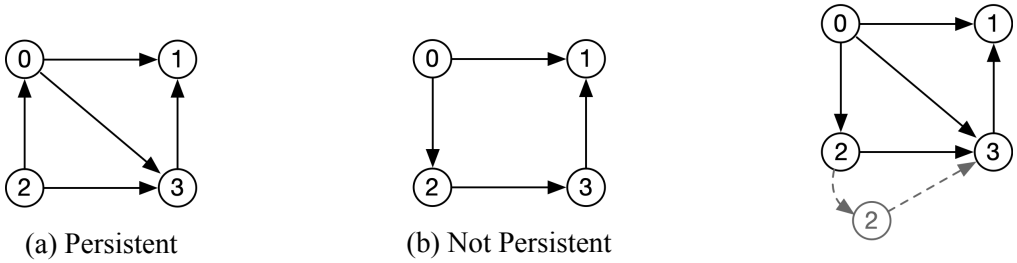
1. Every robot has a position that allows it to satisfy its constraints
2. Its underlying undirected graph is rigid

For example, Figure 11a is persistent because all vertices have a position in which they can maintain their constraints and its underlying undirected graph is rigid (as shown in Figure 4). Figure 11b is not persistent because its underlying undirected graph is flexible (as shown in Figure 3). Figure 11c is not persistent because when v_2 moves around v_3, v_0 is unable to find a position that satisfies all predetermined constraints. In Figure 11c we see that as v_2 moves down, v_0 is unable to maintain the constraint given by e_{02}.

We now formally capture the intuitive concepts described earlier. We call edge $\overrightarrow{e_{ij}}$ *active* if it adheres to $d(\overrightarrow{ij}) = ||p(i) - p(j)|| \quad \forall \overrightarrow{e_{ij}} \in E_H$. Note that this equation is an extension of equation 1. We say that a position of $v_i \in V$ is *fitting* if it is impossible to increase the set of active constraints of v_i by only moving the position of v_i (assuming no other vertex positions are changed). We say that an embedding is a *fitting embedding* if all $v_i \in V$ are fitting. For example, embedding (i) in Figure 12 is not fitting because only e_{01} is made active, while there exists a position in which e_{01} and e_{03} are active (Figure 12 (ii)). We formally define persistence below.

Definition 2.8. *A formation (H, p) is persistent if all fitting embeddings $q \in N_p$ are congruent to p.*

Alternatively, we can characterize (generic) persistence using rigidity.



(a) Persistent (b) Not Persistent (c) Not Persistent

Figure 11: Persistent vs not persistent. (a) is persistent because the underlying undirected graph is rigid and all vertices have a position in which they can maintain their constraints. (b) not persistent because the underlying graph is flexible. (c) not persistent because if v_2 moves then v_0 is unable to find a position that satisfies all its constraints.

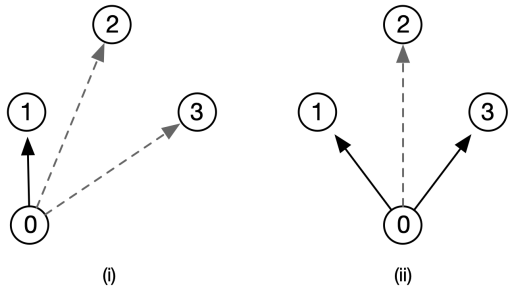


Figure 12: Fitting embedding examples. Dotted lines represent non-active constraints, and solid lines represent active constraints. (i) is not fitting because there is a a different position of v_0 such that the number of active constraints increases. If v_0 is moved to the position shown in (ii), we see that the maximum number of active constraints is reached. Therefore, (ii) is a fitting embedding.

Theorem 2.9. [7]
A directed graph H is persistent if and only if the underlying undirected graph of every subgraph obtained by removing outgoing edges from $v \in V$ with out-degree > 2 until all $v \in v_H$ have out-degree ≤ 2 is generically rigid.

In robotics, constraints are detected using sensors. To reduce the number of sensors required in formation control, we must reduce the number of constraints.

Similar to the concept of a graph being minimally rigid, we take a subset of persistent graphs that allows for the reduction in the number of sensors used in formation control.

Definition 2.10. *A formation $F = (H, p)$ is minimally persistent if*

1. *F is persistent*
2. *$F - e'$ is no longer persistent $\quad \forall e' \in E_H$*

To determine whether or not a formation is *minimally persistent* we can check the out-degrees of each vertex in the formation.

Theorem 2.11. [7]

A formation is minimally persistent if and only if one of the following conditions hold:

1. *Three vertices in the formation have out-degree = 1 and all other vertices in the formation have out-degree = 2.*
2. *One vertex in the formation has out-degree = 0, one vertex has out-degree = 1, and the rest of the vertices have out-degree = 2.*

Figure 11a is minimally persistent because $d(v_1) = 0, d(v_3) = 1$, and $d(v_0), d(v_2) = 2$. An example of a minimally persistent formation that adheres to the first condition of Theorem 2.11 is Figure 13 (i) with $d(v) = 1, 1, 1, 2$ for v_0, v_1, v_2, v_3 respectively. Observe that Figure 13 (ii) is persistent (by Theorem 2.9) but not minimally persistent (as shown in Figure 5).

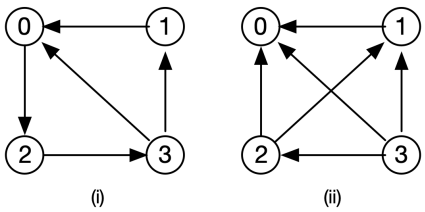


Figure 13: Minimally persistent vs. persistent. (i) is minimally persistent because v_0, v_1, v_2 have out-degree = 1 and all remaining vertices have out-degree = 2. (ii) is persistent but not minimally persistent as shown by Figure 5.

2.5 Persistent Leader-Follower Formations

In this thesis, we focus on the subset of minimally persistent graphs given by condition (2) of Theorem 2.11. This subset, which we refer to as persistent leader-follower formations, contains three types of vertices.

Definition 2.12. *We define the three types of vertices in a persistent leader-follower formation.*

1. *One vertex with out-degree = 0 is called the leader*
2. *One vertex with out-degree = 1 is called the co-leader*
3. *All remaining vertices with out-degree = 2 are called the followers*

An example of a persistent leader-follower formation can be seen in Figure 14 (ii) with starting edge e_{01}. We call v_0 the leader ($d(v_0) = 0$), v_1 the co-leader ($d(v_1) = 1$), and $\{v_2, v_3\}$ the followers ($d(v_2) = 0$, $d(v_3) = 0$). Figure 14 (iii) represents a different persistent leader-follower formation, with starting edge e_{02}, that shares the same underlying undirected graph as Figure 14 (ii).

As a formation moves, the leader dictates which location the formation must move to next. The co-leader is used to determine the amount a formation should rotate during movement. If the starting edge rotates, the rest of the formation must rotate in order to continue maintaining their constraints.

We characterize persistent acyclic leader-follower formations using H1 constructions below.

Theorem 2.13. *[Corollary 2 of 8]*
A minimally persistent acyclic graph with $|V| > 1$ always has an associated leader-follower formation and can always be obtained by inductive construction using H1 moves. [8]

For persistent acyclic H1 graphs, we use the concept of *waves* [2]. We divide the steps of a Henneberg sequence into a sequence of waves w_i such that the vertices in wave w_i only have outgoing edges to vertices in previous waves, $\{w_0, ..., w_{i-1}\}$. For example, in Figure 14, (ii) has 4 waves $(\{v_0\}, \{v_1\}, \{v_2\}, \{v_3\})$ where e_{01} is our starting edge, v_2 has out-going edges to $\{v_0, v_1\}$ in previous waves, and v_3 has out-going edges to $\{v_0, v_2\}$ in previous waves. Figure 14 (iii) has 3 waves $(\{v_0\}, \{v_2\}, \{v_1, v_3\})$. Note that $\{v_1, v_3\}$ share the same wave because both have outgoing edges to $\{v_0, v_2\}$ in previous waves.

Given that waves w_0 and w_1 will always be occupied by the leader and co-leader respectively, we define the *diameter* of a formation to be the number of waves following w_1. For example, in Figure 14, (ii) has diameter $d = 2$ while (iii) has $d = 1$.

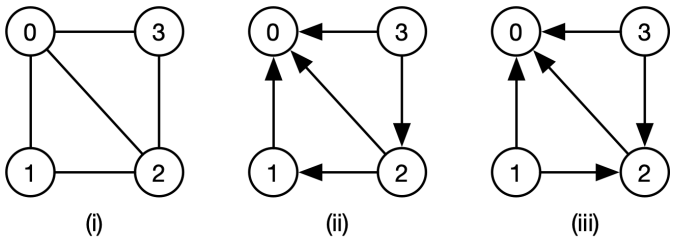


Figure 14: Persistent leader-follower examples. (i) is the underlying undirected graph of (ii) and (iii). (ii) is a persistent acyclic leader-follower formation with leader v_0, co-leader v_1, and followers $\{v_2, v_3\}$. (ii) has four waves $(\{v_0\}, \{v_1\}, \{v_2\}, \{v_3\})$, with diameter $d = 2$. (iii) is a persistent acyclic leader-follower formation with leader v_0, co-leader v_2, and followers $\{v_1, v_3\}$. (iii) has three waves $(\{v_0\}, \{v_2\}, \{v_1, v_3\})$ and $d = 1$.

3 Approach

The approach to this thesis is divided into four steps: generating combinatorics, associating geometry, simulation, and analysis. We now describe these four stages in more detail.

3.1 Generate Combinatorics

To generate the persistent acyclic leader-follower graphs (described in Section 2.4) for use in experiment, we use Procedure 1. Procedure 1 starts by generating all non-isomorphic minimally rigid undirected frameworks that can be constructed using only H1 moves, on n vertices. Next, all persistent leader-follower graphs are generated from each non-isomorphic minimally rigid undirected framework. Finally, all cyclic and isomorphic persistent leader-follower graphs are removed and we are left with non-isomorphic persistent acyclic leader-follower graphs. Note that Procedure 1 returns a list of persistent acyclic leader-follower graphs. This is done for clarity of presentation here, whereas in the actual script we instead write each persistent acyclic leader-follower graph to

file instead of holding in memory.

Procedure 1: Persistent acyclic leader-follower graph generation

1 **Input:** Some number of vertices n
2 **Output:** All persistent acyclic leader-follower graphs on n vertices
3 **if** $n = 4$ **then**
4 get all combinations of 5 edges to find all minimally rigid frameworks on 4 vertices (u)
5 **else if** $n > 4$ **then**
6 generate all combinations (c) of 2 vertices in range (0, $n - 1$)
7 **for** *each minimally rigid framework, G, on $n - 1$ vertices* **do**
8 **for** *each combination $c_i \in c$* **do**
1. B = copy(G)
2. add v_{n-1} to B via H1 move to existing vertices given by c_i
3. insert B into list of minimally rigid frameworks on n vertices (u)
9 **for** *each G in u* **do**
10 x = []
11 **if** *G is not isomorphic to any $x_i \in x$* **then**
12 append G to list of non-isomorphic minimally rigid undirected frameworks (x)
13 z = []
14 **for** *each G in x* **do**
15 **for** *$e \in G$ and the reverse of e* **do**
16 B = copy(G)
17 y = []
1. designate e the starting edge; source has out-degree = 1 and sink has out-degree = 0
2. inductively assign out-degree = 2 to each remaining vertex $v \in B$

insert B into list of persistent leader-follower graphs derived from G (y)
18 **for** *each H in y* **do**
19 **if** *H is not isomorphic to any $z_i \in z$ and H has no cycles* **then**
20 append H to list of non-isomorphic persistent acyclic leader-follower graphs (z)
21 **return** z

To the best of our knowledge, the following is unknown.

Conjecture 3.1. *For each starting edge of an H1 graph there is a unique persistent acyclic leader-follower graph.*

Theorem 3.2. *If conjecture 3.1 holds, then Procedure 1 generates all persistent acyclic leader-follower graphs on n vertices.*

Proof. We show that if Conjecture 3.1 is true, a formation is a persistent acyclic leader-follower graph if and only if Procedure 1 produces it.

$\Rightarrow$ Let H be some persistent acyclic leader-follower graph on n vertices. Note that lines 4 and 6

generate all possible minimally rigid graphs on n vertices that can be constructed using H1 moves. Given that all possible H1 sequences are generated on n vertices, using Theorem 2.13, Procedure 1 must generate H.

$\Leftarrow$ Let $H = (V, E_H)$ be a directed graph generated from Procedure 1. From theorems 2.6 and 2.7, we can say that the H1 construction outlined in lines 4 and 6 of the procedure imply that the underlying undirected graph of H is minimally rigid. Observe lines 11-12 of the procedure. We see by Theorem 2.11 that this inductive construction gives a minimally persistent leader-follower graph. All cyclic persistent leader-follower graphs are removed in lines 13-14 of the procedure. Therefore, we have a resulting set of persistent acyclic leader-follower graphs. Using Corollary 2.13, we can say that all persistent acyclic leader-follower graphs on n vertices are generated in Procedure 1. □

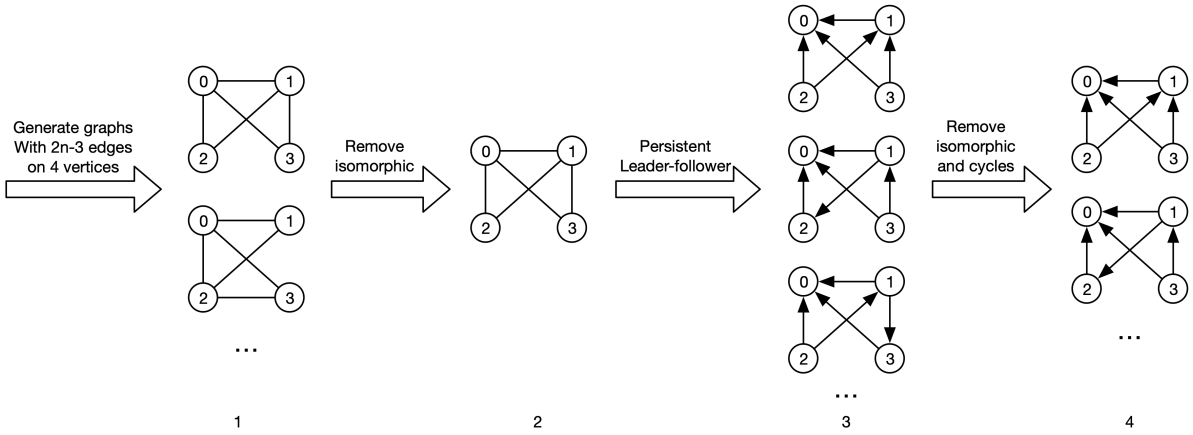


Figure 15: n = 4 Graph Generation Example

The majority of the running time for this procedure occurs in lines 11-12 where persistent acyclic leader-follower graphs are generated given some underlying undirected graph. We observe the running time of this procedure to be $O(2n|E|^2|V|)$.

An example of the graph generation described in Procedure 1 can be seen in Figure 15. Consider the underlying undirected graph present in step 2. Note that all persistent leader-follower graphs present in step 3 have different starting edges but share the same underlying undirected graph. The starting edges of the persistent leader-follower graphs in step 3 are $\overrightarrow{e_{01}}, \overrightarrow{e_{02}}, \overrightarrow{e_{03}}$ respectively. All possible remaining starting edges ($\overrightarrow{e_{12}}, \overrightarrow{e_{13}}$) are considered in generation, but not shown in Figure 15.

The total number of minimally rigid undirected graphs and persistent acyclic leader-follower graphs generated by Procedure 1 are shown in Table 1. This table also shows the average number of iterations taken per experiment and the average amount of time taken per experiment for each value of n. We were only able to run simulations through $n = 8$ due to the large amount of time required to run all simulations for $n > 8$.

error statistic/n	4	5	6	7	8
# of minimally rigid undirected frameworks	1	3	11	61	499
# of persistent acyclic leader-follower graphs	3	13	79	633	6430
Total # of experiments	9	39	237	1899	19290
Average # of iterations taken	1982.33	2186.72	2838.28	3527.70	3595.47
Average amount of time taken (seconds)	65.42	72.16	93.66	117.88	118.65

Table 1: Total number of minimally rigid undirected frameworks, persistent acyclic leader-follower graphs, number of experiments performed, average number of iterations taken, and average amount of time taken for each value of n.

3.2 Associate Geometry

In order to define starting positions in simulation for each robot in a persistent acyclic leader-follower graph, we must assign embeddings to the graphs outputted by Procedure 1.

For each value of n we generate three random embeddings in the plane. Then, for each persistent acyclic leader-follower graph H in the output of Procedure 1, we assign each of three random embeddings. The frame size associated with the Robotarium simulator is a 1×1.5 frame. Within this frame, each $v \in H$ is randomly assigned an (x, y) coordinate within an x range of [-0.5, 0.5] and a y range of [0.5, 0.9]. The leader and co-leader of each graph are assigned to coordinates (0, 0.3) and (0, 0.5) respectively. The (x, y) coordinate of the leader is slightly ahead of all other robots in the formation in order to reduce collisions. We fix the (x, y) coordinate of the co-leader across embeddings in order to remove any possible impact that the co-leaders position may have on formation accuracy in movement. Observe the persistent acyclic leader-follower graph given in Figure 16. We see that this graph is generated via the process shown in Figure 15. In Figure 17, note that there are three different embeddings in the plane. All three embeddings stem from the same persistent acyclic leader follower graph (Figure 16), but each follower has three unique (x, y) coordinates across 3 unique random embeddings.

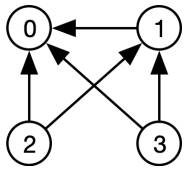


Figure 16: Persistent acyclic leader-follower graph on $n = 4$ vertices

Each of the 3 random embeddings generated for each persistent acyclic leader-follower graph outputted by Procedure 1 are used in simulation.

4 Experiments

For this thesis, we are using the Robotarium platform from Georgia Tech [19]. It includes a simulator allows us to simulate robot swarms virtually which can then be remotely applied to real robots at Georgia Tech. In this section, we detail the setup and details of each experiment.

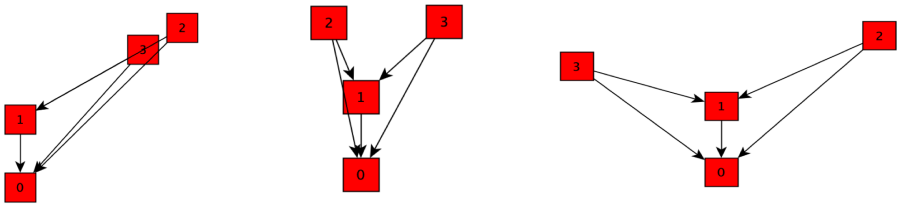


Figure 17: Three random embeddings for $n = 4$ applied to figure 16. See table 5 for associated (x, y) coordinates.

To start the experiment, we set destination points along a circular path that dictate where the formation should move next. The circle shape was chosen based on other related works [4, 18]. After the circular path is given, we rely on the Robotarium leader-follower script for our experiment [19]. The setup given by Robotarium is as follows. The Robotarium leader-follower code calculates the distance of each non-leader robot to their neighbors. These distances are then used to calculate the desired velocity of each robot for the next iteration. To check to see if the leader has reached the next destination point in the circle, the distance from the leaders position to the destination point is measured. If the distance is less that 0.05, then the leader is deemed close enough and the next destination point is given to the leader.

During each experiment, we run a persistent acyclic leader-follower formation in leader-follower movement. For example, we run each of the three formations present in Figure 17 individually in experiment. Table 1 displays the total number of experiments run for each value of n. Each persistent acyclic leader-follower graph has three corresponding simulations (for each of three embeddings). Figure 18 pictures the start of experiment for each of the three embeddings present in Figure 17. Each of these formations then complete a circular pathway as seen in Figure 19. The rectangular region in Figure 19 represents the region in which all embeddings are randomly generated. The leader then moves to a starting place on the circle (blue point) and then moves counterclockwise along the red circle. The circle also allows us to test how accurately the shape is maintained while the formation is turning.

During the experiment, we record the positioning of the robots and all $\binom{n}{2}$ pairwise distances at each iteration. Note that we record edge and non-edge pairwise distances. We also take note of the number of iterations taken for each formation to complete the full circle.

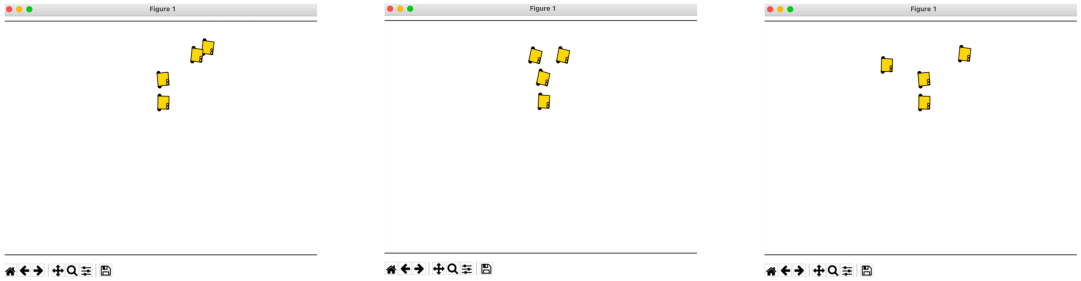


Figure 18: Start of simulation for each embedding shown in figure 17.

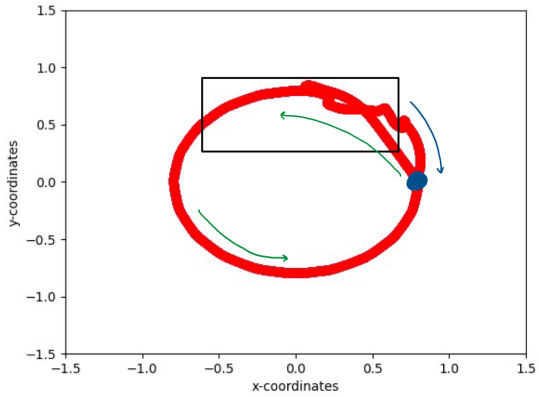


Figure 19: Circular path that robots follow during experiment.

In order to validate our python experiments on hardware, we took two valid python experiments from $n = 8$ and submitted them to Robotarium where they were run on real robots. We chose to use $n = 8$ because there is more variety in the diameter of a formation than for smaller values of n. Additionally, using a larger value of n allows us to observe more variability through collision.

To reduce the number of variables that might affect performance we chose two persistent acyclic leader-follower graphs that shared the same underlying undirected graph (Figure 20) and embedding. Figure 21 shows the formations of the two experiments submitted to Robotarium. The formation on the left has starting edge e_{02} with diameter = 3, and the formation on the right has starting edge e_{30} with diameter = 5. We chose these starting edges because their difference in diameter allowed us to note any preliminary pattern between diameter and hardware experiments. We collected the same data and calculated the same error statistics at each iteration.

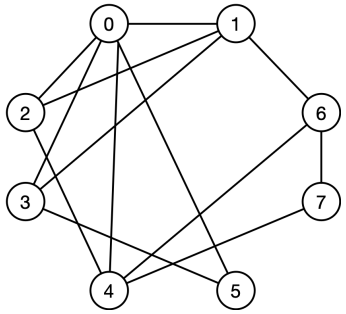


Figure 20: Underlying undirected graph of the formations submitted to Robotarium for hardware experiments. Both formations can be seen in Figure 21.

Due to an upgrade in Robotarium software, it was not possible to start the robots at the exact starting positions. To adjust, we instantiate a controller prior to the circular movement that moved robots within ± 0.18cm of their starting positions. Note that the robot size in Robotarium is about 0.06cm meaning that their inability to reach their exact positions is due to collision with other robots. When the robots collide they may be unable to adjust and and reach the desired starting place. Instead they would remain stuck behind the robots they collided with.

Figure 22 shows the start of the python simulation next to the start of the hardware simulation for the

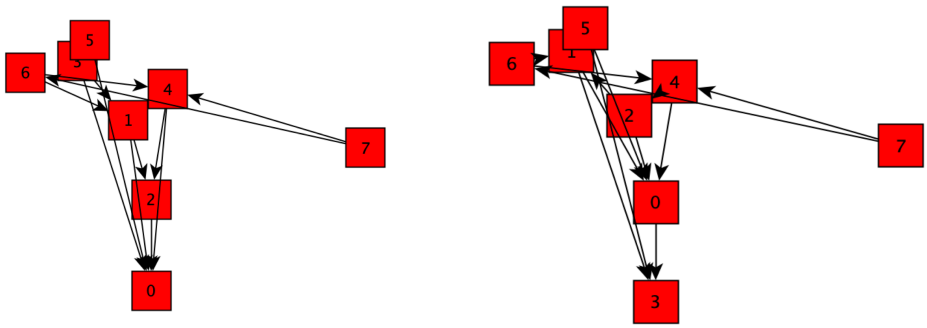


Figure 21: Both formations submitted to Robotarium for hardware experiments. Formation on the left has starting edge e_{02} and diameter = 3. Formation on the right has starting edge e_{30} and diameter = 5.

same formation. Note that the starting positions are similar for both simulations. Both the python and hardware simulations are useful to help determine how well shape is maintained throughout movement.

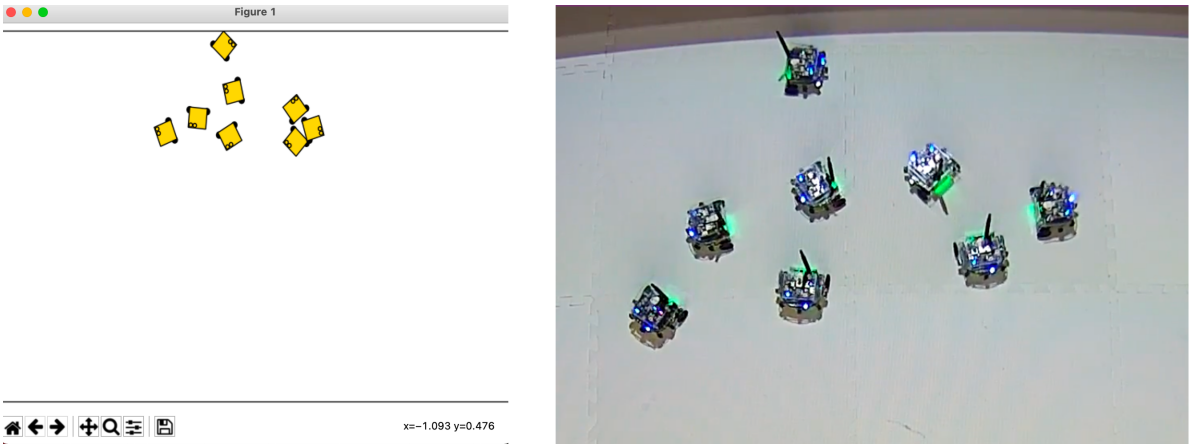


Figure 22: Python vs hardware starting positions. Photo on the left shows starting positions of robots in python experiment. Photo on the right shows starting positions of robots in hardware experiment. Both photos correspond to the same $n = 8$ experiment.

5 Analysis

After all our experiments were run, we divided the experiments into two groups. One group (*incomplete experiments*) represented experiments in which the formation was not able to complete the full circle path. The percent of incomplete experiments increased as n increased due to an increase in collision. For larger values of n, there are more collisions and if robots are unable to adjust and move past the collision then they are unable to complete the circle path. The second group (*valid experiments*) represents experiments in which the formation is able to complete the full circle path. For each valid experiment, we calculated error statistics and rank them by how they performed.

Note that each error statistic is averaged over the number of iterations taken for each formation to complete the full circle path.

After all of error statistics are calculated, each experiment has four associated error statistics.

1. RMSD of average pairwise distances over all iterations ($RMSD_{dist}$)
2. Percent error of average pairwise distances over all iterations ($Error_{dist}$)
3. RMSD of the percent error of each pairwise distance over all iterations ($RMSD_{error}$)
4. RMSD of the average distance robots were from their desired positions over all iterations ($RMSD_{pos}$)

Each of these error statistics will be described in more detail below.

The first error statistic ($RMSD_{dist}$) we calculate is the root-mean-square deviation (RMSD) of each average pairwise distance. We first calculate the difference between each measured and desired pairwise distance. We then square these differences and add them together for each iteration. We then calculate the RMSD value of each of the average pairwise distance and averaged all resulting RMSD values. This statistic helps us analyze how close each average pairwise distance is to the desired distance. Each valid experiment has one average RMSD value representing the accuracy of distant constraint maintenance.

The second error statistic ($Error_{dist}$) is the percent error of the same pairwise distances described earlier. We first calculate the absolute value of the difference between each measured and desired pairwise distance. We then divide that value by the desired pairwise distance. We sum this value at each iteration and then average at the end of the experiment. Finally, we multiply the average by 100 to get a percent error value for each of $\binom{n}{2}$ pairwise distances. We then average all such percent error values. Each valid experiment has one average percent error value that represents the percent error of pairwise distances in each formation.

The third error statistic ($RMSD_{error}$) is the RMSD value of each of the average percent error values calculated for $Error_{dist}$. Each valid experiment has one average RMSD value representing the RMSD of $Error_{dist}$.

Note that for $Error_{dist}$ and $RMSD_{error}$, the expected value is 0, implying that the robots exactly maintained their pairwise distances throughout movement.

The final error statistic ($RMSD_{pos}$) calculated is the RMSD value of the distance each robot was from its desired position over all iterations. The desired position of each robot is calculated at each iteration using translation of the leader and rotation of the starting edge. Note that this implies that the leader is always in the desired position. For example, Figure 23 displays the initial formation at iteration 0 (left) and the formation after one iteration (right). The red overlay represents the desired positions of each robot assuming all constraints are maintained. The black formation on the right represents the actual positions of the robots after one iteration. The green represents the distance between where the robot is measured to be versus its desired position is. At iteration 0, all robots are in their desired position. Each formation has one RMSD average value measuring how close robots were on average to their desired position.

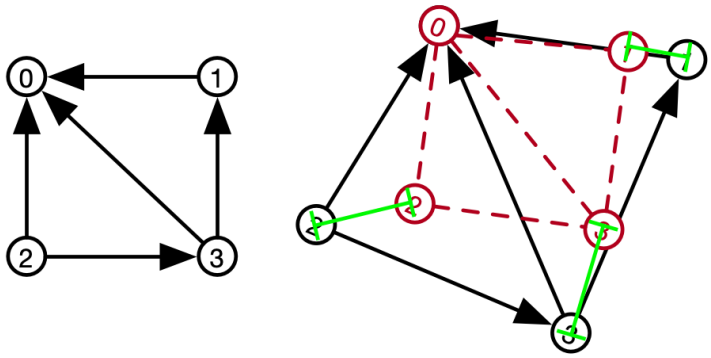


Figure 23: Black = actual position, red = desired position, green = distance from actual position to desired position

Each of these values are then used to rank simulations from lowest (performed best) to highest (performed worst). To make a distinction between the rigidity of the formation versus the formation control movement, $RMSD_{dist}$, $RMSD_{error}$, and $Error_{dist}$ are split into two rankings. One ranking considered only values for each edge in the graph while the other considered edge and non-edge values.

The results of the analysis support our hypothesis: as diameter increases, formation accuracy decreases. Tables 2 and 3 show the increase in average error statistic values as diameter increases[3]. This can also be seen in the Figures 25, and 26 respectively; as diameter is increasing, the average error value as well as the standard deviation is increasing. It is clear from the figures that this performance follows for all embeddings of $n = 8$. We choose to display $n = 8$ here because there is more variety in the diameter of a formation than for $n < 8$.

d\n	4	5	6	7	8
1	0.18871704	0.15127979	0	0	0
2	0.6145895	0.53296548	0.60303784	0.63159749	0.634582913
3	0	0.60295534	0.65207362	0.66487494	0.644175197
4	0	0	0.65972821	0.6768057	0.668898979
5	0	0	0	0.68168558	0.682686174
6	0	0.0	0.0	0.0	0.683833158

Table 2: $RMSD_{pos}$ (third embedding). As diameter increases, $RMSD_{pos}$ increases.

In observing the data, we noted two additional observations. First, given that all vertices for all values of n are placed in the same starting region, higher values of n lead to relatively smaller distances between vertices. We found that the smaller distances seem to lead to improved performance in simulation. This is likely due to collisions. For example, in Figure 24 we see that there are four robots that start the experiment in collision. These robots will maintain their distances well throughout the experiment because even in collision their constraints are being maintained. When there are larger distances between robots, robots must avoid collisions or constraints will not be

[3]All remaining data tables can be found in the appendix.

d\n	4	5	6	7	8
1	0.42088129	0.39888466	0	0	0
2	0.91240378	1.13357645	0.99943633	1.03955448	0.91324315
3	0	1.24546836	1.28843124	1.15603295	0.981036571
4	0	0	1.40382787	1.27426547	1.074839752
5	0	0	0	1.30209779	1.131013459
6	0	0.0	0.0	0.0	1.149592799

Table 3: $Error_{dist}$ (third embedding). As diameter increases, $Error_{dist}$ increases.

maintained. Smaller distances make it easier for robots to maintain their given constraints. Our second observation is that if there is a formation with a heavy concentration of robots surrounding the starting edge, the formation could perform worse due to collisions with the leader or co-leader. If the leader gets stuck in collisions, then the formation is unable to move to the next destination point until the leader adjusts. While adjusting, the leader is likely altering its path from the desired circle path which can lead to a decrease in formation accuracy.

In this thesis, we also had the chance to collect data from some hardware simulations. Table 4 shows the average error statistics in the python experiments and their corresponding hardware experiment. We see from the table that the formation with starting edge e_{02} (diameter = 3) consistently performs better than the same formation with starting edge e_{30} (diameter = 5). These preliminary results suggest that our hypothesis holds even in hardware.

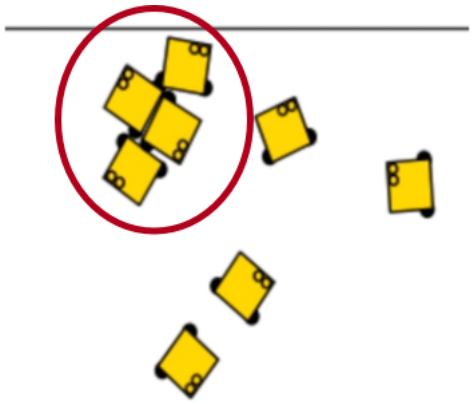


Figure 24: Example of some robots starting experiment in collision due to smaller desired pairwise distances.

error statistic	python e_{02}	e_{02}, d = 3	python e_{30}	e_{30}, d = 5
$RMSD_{error}$	0.664831022	0.378922889	1.47917955	1.049006855
$RMSD_{dist}$	0.288289518	0.295855243	0.554959667	0.474810987
$RMSD_{pos}$	0.496728699	0.623636076	0.917997495	0.82864371
$Error_{dist}$	0.604547405	0.807702048	1.294803504	1.02166483

Table 4: Hardware vs python data for $n = 8$ experiments. Both formations share same underlying undirected graph and embedding. The formation in the second column has diameter = 3 and the formation in the third column has diameter = 5.

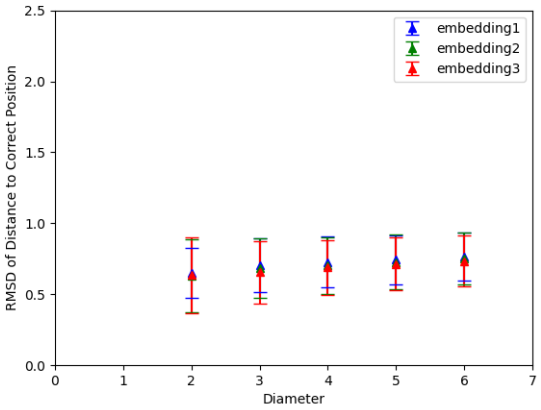


Figure 25: $n = 8$ (third embedding) $RMSD_{pos}$. As diameter increases, $RMSD_{pos}$ increases.

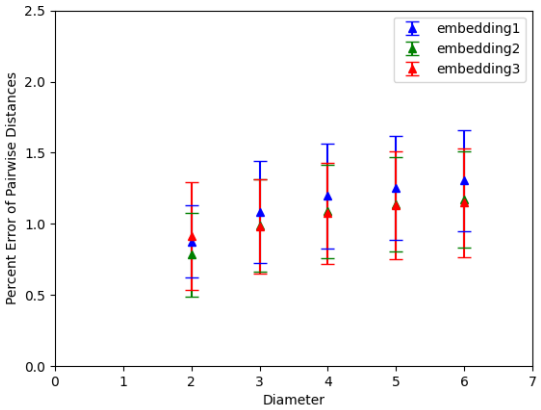


Figure 26: $n = 8$ (third embedding) $Error_{dist}$. As diameter increases, $Error_{dist}$ increases.

6 Conclusions and Future Work

The main research question for this thesis was to identify which persistent leader-follower formations most accurately maintain their distance constraints in movement. We stated a hypothesis in terms of a combinatorial concept which we called diameter, defined using waves capturing inductive H1 steps. We designed experiments to evaluate the performance of formations with varying diameters. Our results support the hypothesis that, as diameter increased, performance worsened.

For future work, the effects of leader and co-leader positioning in the formation could be explored. Through early experimentation in this thesis, we noted that there seemed to be some correlation between the performance of the formation in simulation and the positioning of the leader and co-leader. To reduce this extra variable and focus on the newly defined concept of formation diameter, we fixed the positions of the starting edge for all embeddings. However, alternate positioning for these two robots could lead to different discoveries. Along this idea, future research could explore the initial pairwise distance between the leader and co-leader and its effect on formation performance.

Preliminary results from Robotarium hardware experiments suggest that diameter also impacts the performance of real robots. An extension to this thesis could be perform more hardware experiments to see if this pattern holds.

In this thesis we proposed an approach to generate persistent acyclic leader-follower graphs. An expansion on this would be to prove conjecture 3.1 and in turn identify whether this is an procedure that truly generates all such graphs, or if there are some left to be generated. It is still left to be proved whether each leader to co-leader edge has a unique graph, or if one starting edge can have multiple associated persistent acyclic leader-follower graphs.

Appendix

For completeness, we include experimental data and associated scripts for the project. We first include the three random embeddings for each value of n in the plane, and their point sets (for starting edge e_{01}). Next we include additional data collected for python and hardware experiments that were not included earlier. Note that we just include data that considers edge and non-edge constraints. We also include charts that represent the data collected over all three embeddings (similar to Figures 25 and 26). Finally, we include python scripts necessary to recreate this project.

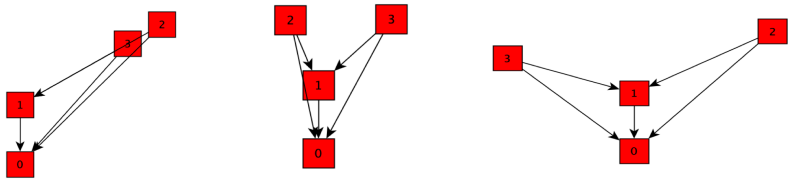


Figure 27: $n = 4$ embeddings

	v_0	v_1	v_2	v_3
x	0	0	0.449491227	0.34051327
y	0.3	0.5	0.769925783	0.705165985

Table 5: Random Point Set For Embedding 1 $n = 4$

	v_0	v_1	v_2	v_3
x	0	0	-0.07690891	0.198751517
y	0.3	0.5	0.691011633	0.693996412

Table 6: Random Point Set For Embedding 2 $n = 4$

	v_0	v_1	v_2	v_3
x	0	0	0.408990259	-0.374189291
y	0.3	0.5	0.713483423	0.620412193

Table 7: Random Point Set For Embedding 3 $n = 4$

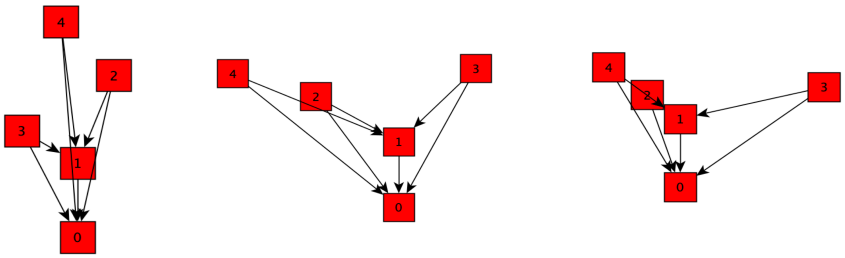


Figure 28: $n = 5$ embeddings

	v_0	v_1	v_2	v_3	v_4
x	0	0	0.089470845	-0.137095439	-0.040711689
y	0.3	0.5	0.736415298	0.586954436	0.880689868

Table 8: Random Point Set For Embedding 1 $n = 5$

	v_0	v_1	v_2	v_3	v_4
x	0	0	-0.228970784	0.213622782	-0.459666109
y	0.3	0.5	0.637262891	0.723372176	0.707613902

Table 9: Random Point Set For Embedding 2 $n = 5$

	v_0	v_1	v_2	v_3	v_4
x	0	0	-0.087500136	0.383716074	-0.19109638
y	0.3	0.5	0.57014785	0.593942504	0.651523727

Table 10: Random Point Set For Embedding 3 $n = 5$

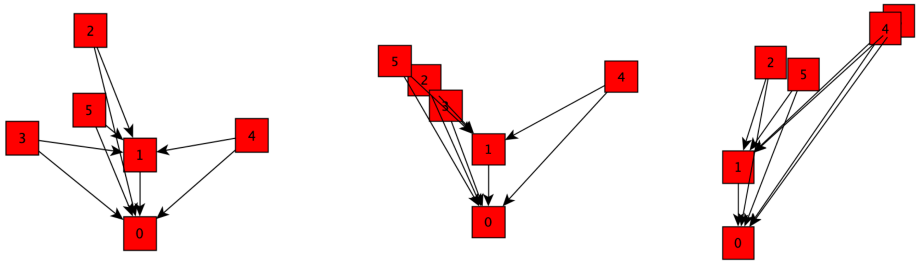


Figure 29: $n = 6$ embeddings

	v_0	v_1	v_2	v_3	v_4	v_5
x	0	0	-0.129605767	-0.307303841	0.294019141	-0.132858187
y	0.3	0.5	0.815841578	0.542839034	0.548624836	0.613783538

Table 11: Random Point Set For Embedding 1 $n = 6$

	v_0	v_1	v_2	v_3	v_4	v_5
x	0	0	-0.166271697	-0.111345662	0.346272755	-0.243076637
y	0.3	0.5	0.689956524	0.61938367	0.698943904	0.74158974

Table 12: Random Point Set For Embedding 2 $n = 6$

	v_0	v_1	v_2	v_3	v_4	v_5
x	0	0	0.089679377	0.440936639	0.40394344	0.179290529
y	0.3	0.5	0.775537591	0.884475857	0.864503108	0.744278144

Table 13: Random Point Set For Embedding 3 $n = 6$

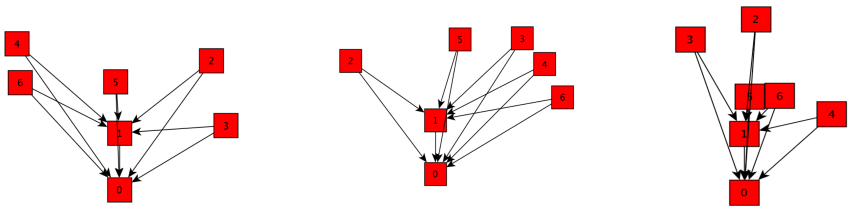


Figure 30: $n = 7$ embeddings

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
x	0	0	0.326545465	0.37749446	-0.363958082	-0.013052432	-0.351933564
y	0.3	0.5	0.755659032	0.526104436	0.815490711	0.680580093	0.678459738

Table 14: Random Point Set For Embedding 1 $n = 7$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
x	0	0	-0.330818876	0.339443126	0.42706497	0.095959535	0.496005107
y	0.3	0.5	0.721281549	0.804793602	0.708926566	0.801087365	0.585182056

Table 15: Random Point Set For Embedding 2 $n = 7$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
x	0	0	0.034303273	-0.157134358	0.255704854	0.016110166	0.103154582
y	0.3	0.5	0.889577725	0.821070467	0.566786911	0.62663855	0.62910292

Table 16: Random Point Set For Embedding 3 $n = 7$

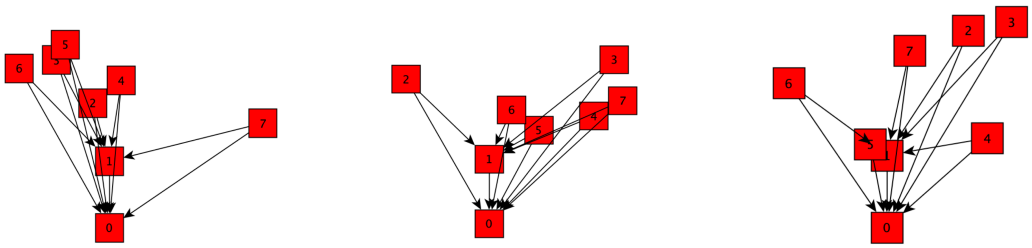


Figure 31: $n = 8$ embeddings

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
x	0	0	-0.050654273	-0.161807364	0.036567452	-0.134539362	-0.276072492	0.471092953
y	0.3	0.5	0.673802252	0.80305222	0.74208323	0.849552704	0.777985542	0.61316444

Table 17: Random Point Set For Embedding 1 $n = 8$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
x	0	0	-0.250710739	0.378464016	0.316329651	0.15006715	0.068208853	0.403860992
y	0.3	0.5	0.747945129	0.808457856	0.632912019	0.589873504	0.652370746	0.680894457

Table 18: Random Point Set For Embedding 2 $n = 8$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
x	0	0	0.220287485	0.337720167	0.271620949	-0.045486181	-0.265601337	0.061541356
y	0.3	0.5	0.84690308	0.870677901	0.547750136	0.530920763	0.700725702	0.790166466

Table 19: Random Point Set For Embedding 3 $n = 8$

d\n	4	5	6	7	8
1	0.223638298	0.33276429	0.189019354	0	0
2	0.518498476	0.569378889	0.501526564	0.696503706	0.653183571
3	0	0.625103495	0.647410511	0.70077347	0.69345009
4	0	0	0.719995436	0.708568537	0.706881886
5	0	0	0	0.741861726	0.714964127
6	0	0.0	0.0	0.0	0.715479644

Table 20: $RMSD_{pos}$ (first embedding). As diameter increases, $RMSD_{pos}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.490540979	0.289121148	0.350693313	0	0
2	1.061496746	0.967935918	0.769223052	0.858648329	0.990001243
3	0	1.222844557	1.074705968	0.883532784	1.23263042
4	0	0	1.189037386	0.92509225	1.364275338
5	0	0	0	0.985365971	1.426384113
6	0	0.0	0.0	0.0	1.479467177

Table 21: $RMSD_{error}$ (first embedding). As diameter increases, $RMSD_{error}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.198136048	0.142801862	0.212838779	0	0
2	0.337588306	0.335535732	0.316285048	0.397572703	0.36268223
3	0	0.404490347	0.407460968	0.427132346	0.427723456
4	0	0	0.452482324	0.45302632	0.462556213
5	0	0	0	0.483644997	0.479835814
6	0	0.0	0.0	0.0	0.495166736

Table 22: $RMSD_{dist}$ (first embedding). As diameter increases, $RMSD_{dist}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.481629931	0.264997371	0.313641804	0	0
2	0.929403162	0.808766352	0.672852712	0.744118576	0.874700581
3	0	1.052917546	0.924267049	0.765549952	1.083594414
4	0	0	1.034995398	0.80212304	1.19554545
5	0	0	0	0.854001047	1.252488236
6	0	0.0	0.0	0.0	1.304404017

Table 23: $Error_{dist}$ (first embedding). As diameter increases, $Error_{dist}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.230542671	0.224976177	0	0	0
2	0.552238834	0.559155632	0.515253007	0.692562549	0.632081421
3	0	0.719700338	0.61812473	0.733122252	0.672514101
4	0	0	0.721048775	0.739233477	0.681521298
5	0	0	0	0.74181863	0.696358257
6	0	0.0	0.0	0.0	0.703710513

Table 24: $RMSD_{pos}$ (second embedding). As diameter increases, $RMSD_{pos}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.263295397	0.2806341	0	0	0
2	1.147319892	0.75493726	1.030914192	0.735941065	0.962661946
3	0	1.03672352	1.342078762	0.874917092	1.143071693
4	0	0	1.617923311	0.948965089	1.239829396
5	0	0	0	0.967394083	1.28631406
6	0	0.0	0.0	0.0	1.327273479

Table 25: $RMSD_{error}$ (second embedding). As diameter increases, $RMSD_{error}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.104451154	0.220508718	0	0	0
2	0.350871755	0.318663347	0.332896675	0.370271094	0.353562755
3	0	0.454333516	0.412144341	0.431614929	0.414097284
4	0	0	0.483359532	0.46936348	1.445112498
5	0	0	0	0.481574868	0.465068978
6	0	0.0	0.0	0.0	0.480503769

Table 26: $RMSD_{dist}$ (second embedding). As diameter increases, $RMSD_{dist}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.254382387	0.263923605	0	0	0
2	0.98015616	0.645685301	0.923602396	0.650633096	0.784426034
3	0	0.887628782	1.180221219	0.765796255	0.988536023
4	0	0	1.423800465	0.828799141	1.087952427
5	0	0	0	0.842911203	1.135130945
6	0	0.0	0.0	0.0	1.170174487

Table 27: $Error_{dist}$ (second embedding). As diameter increases, $Error_{dist}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.226010432	0.193913245	0	0	0
2	0.397064954	0.345366518	0.331966241	0.367838215	0.379135001
3	0	0.434686289	0.419543731	0.411513901	0.40598038
4	0	0	0.452416801	0.453829805	0.436004701
5	0	0	0	0.474534011	0.454682791
6	0	0.0	0.0	0.0	0.462160908

Table 28: $RMSD_{dist}$ (third embedding). As diameter increases, $RMSD_{dist}$ increases for all values of n.

d\n	4	5	6	7	8
1	0.42589826	0.412725225	0	0	0
2	0.99939116	1.268610093	1.121921947	1.180404804	1.03756521
3	0	1.415706426	1.44072528	0.320583304	1.126931006
4	0	0	1.56480408	1.461923864	1.237790568
5	0	0	0	1.498173578	1.305722844
6	0	0.0	0.0	0.0	1.327642508

Table 29: $RMSD_{error}$ (third embedding). As diameter increases, $RMSD_{error}$ increases for all values of n.

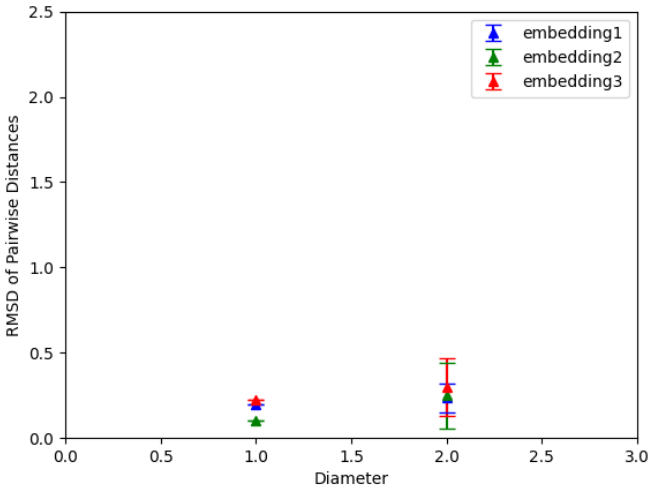


Figure 32: $n = 4$ $RMSD_{dist}$

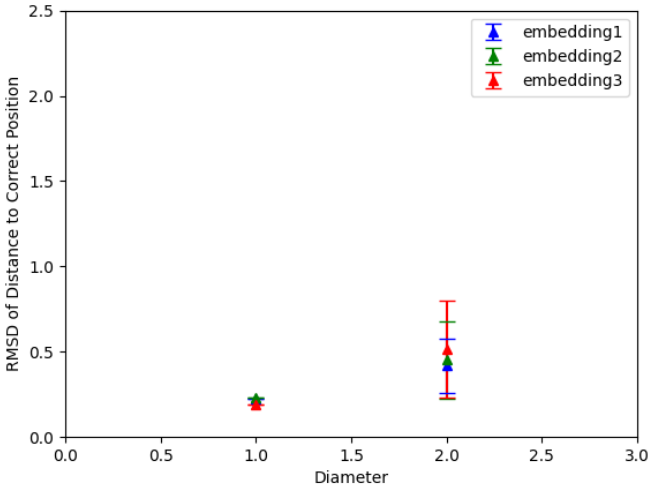


Figure 33: $n = 4$ $RMSD_{pos}$

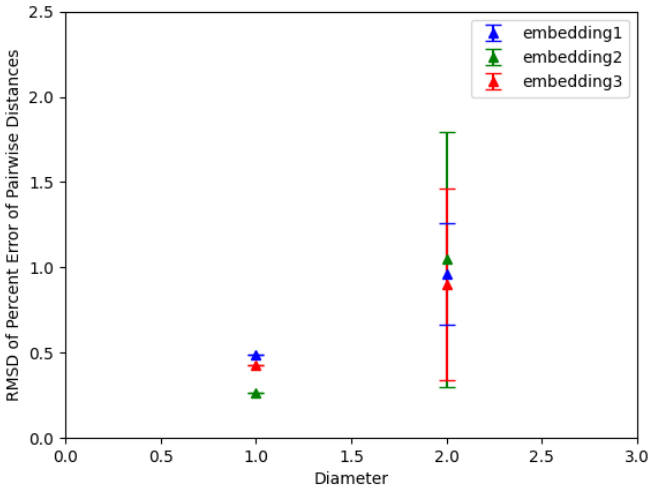


Figure 34: $n = 4$ $RMSD_{error}$

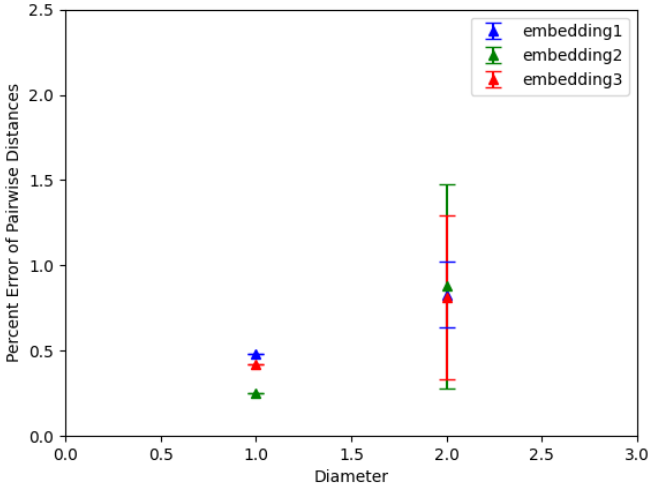


Figure 35: $n = 4$ $Error_{dist}$

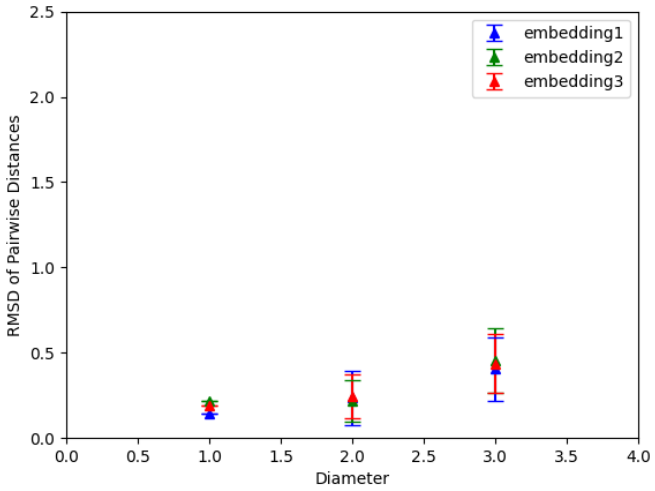


Figure 36: $n = 5$ $RMSD_{dist}$

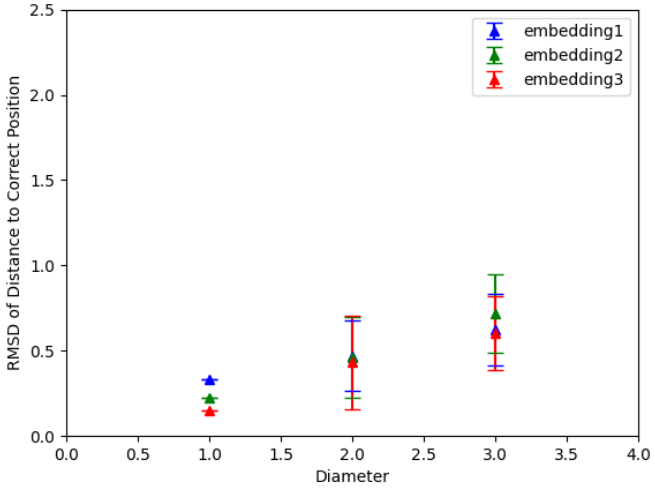


Figure 37: $n = 5$ $RMSD_{pos}$

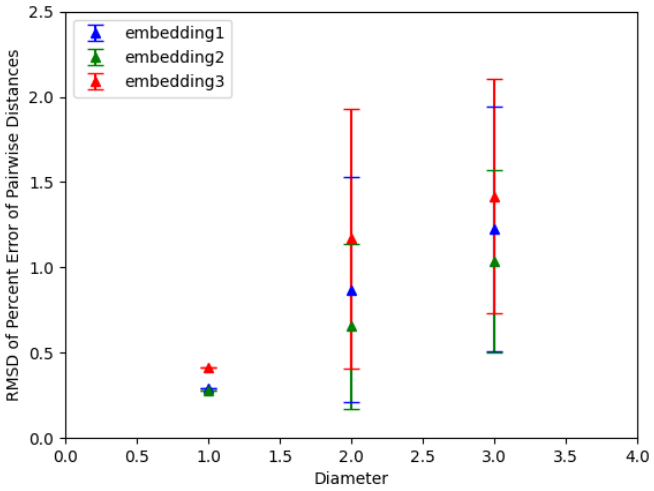


Figure 38: $n = 5$ $RMSD_{error}$

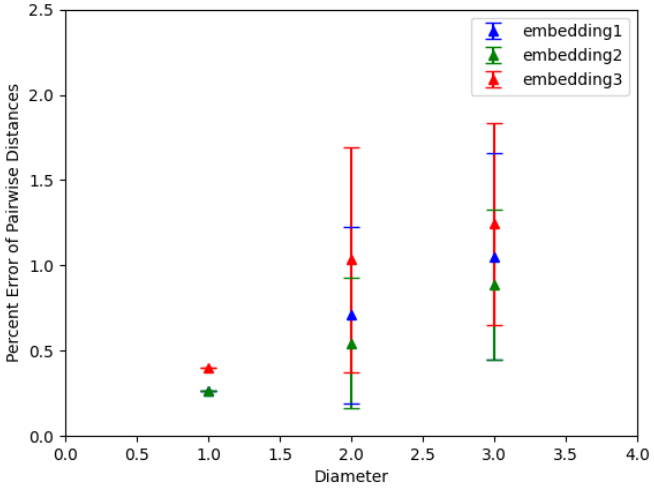


Figure 39: $n = 5$ $Error_{dist}$

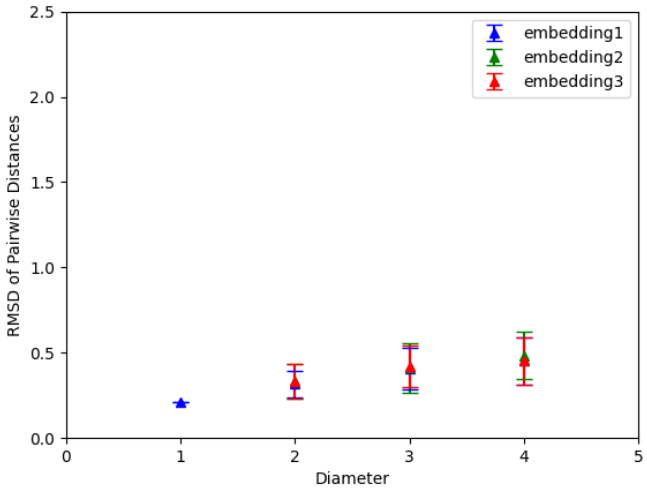


Figure 40: $n = 6$ $RMSD_{dist}$

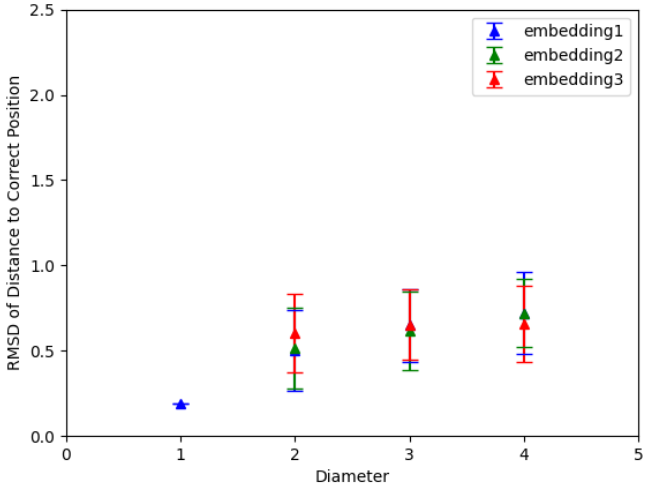


Figure 41: $n = 6$ $RMSD_{pos}$

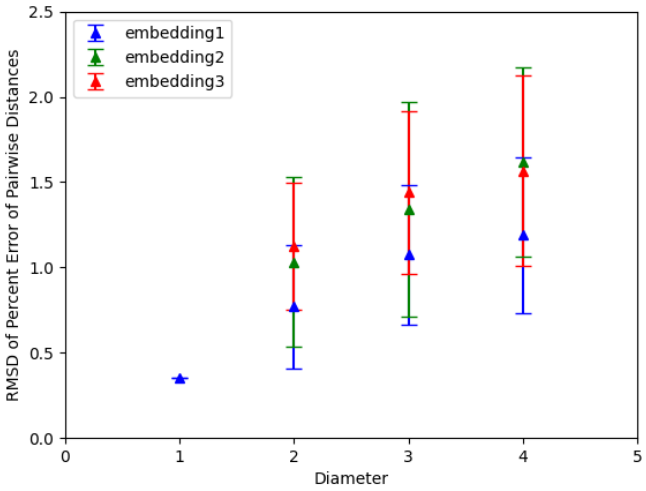


Figure 42: $n = 6$ $RMSD_{error}$

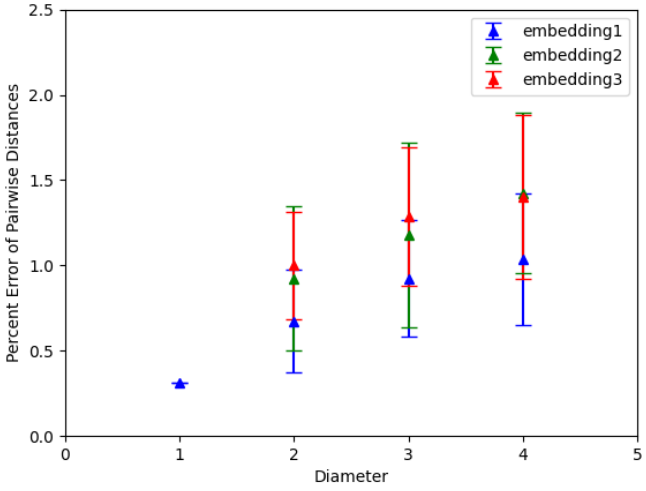


Figure 43: $n = 6$ $Error_{dist}$

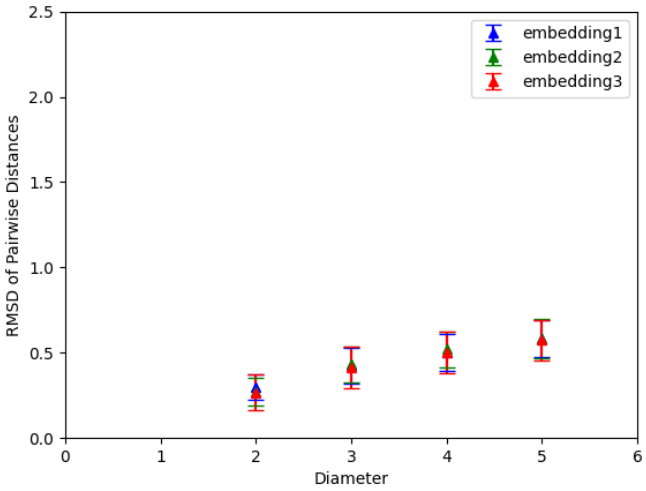


Figure 44: $n = 7$ $RMSD_{dist}$

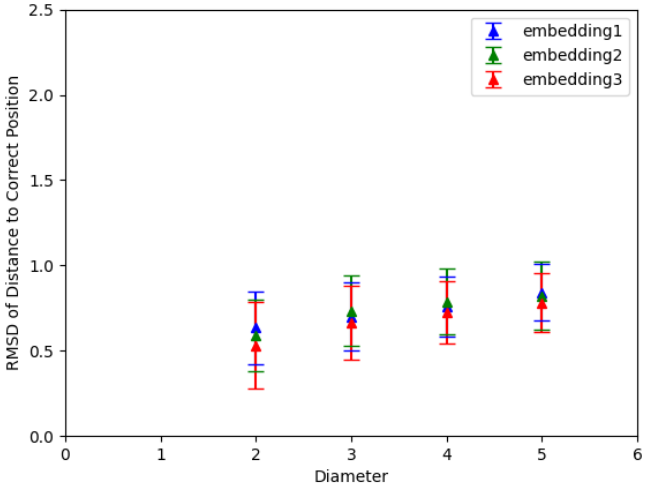


Figure 45: $n = 7$ $RMSD_{pos}$

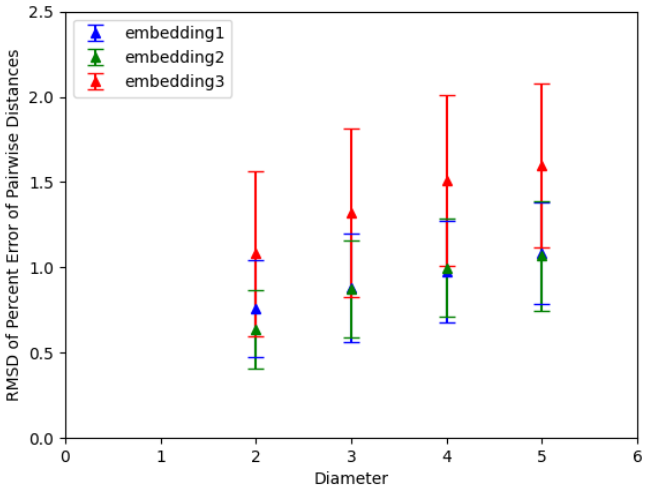


Figure 46: $n = 7$ $RMSD_{error}$

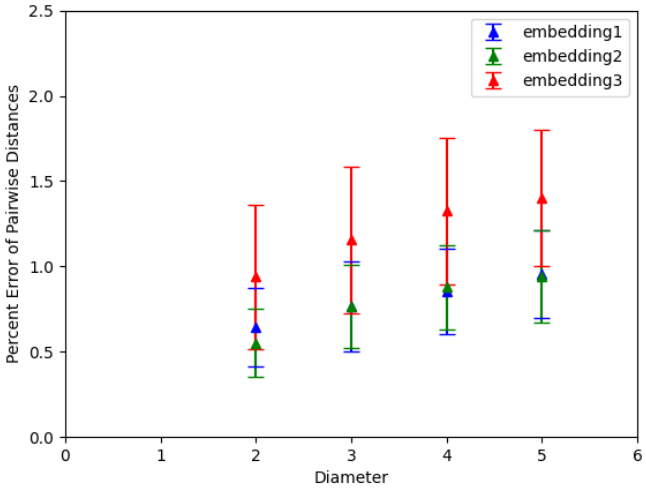


Figure 47: $n = 7$ $Error_{dist}$

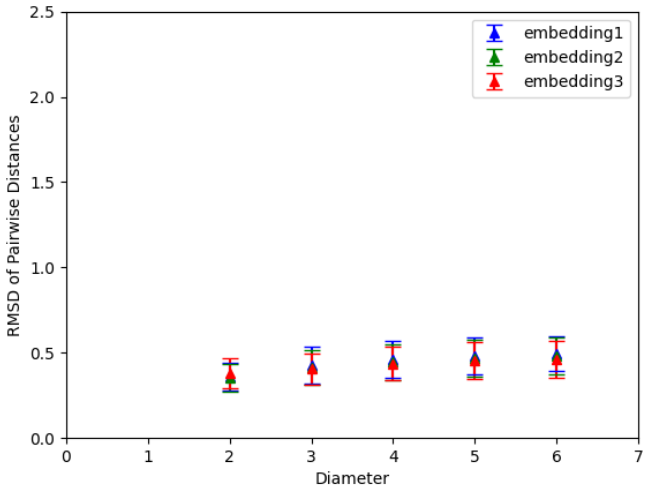


Figure 48: $n = 8$ $RMSD_{dist}$

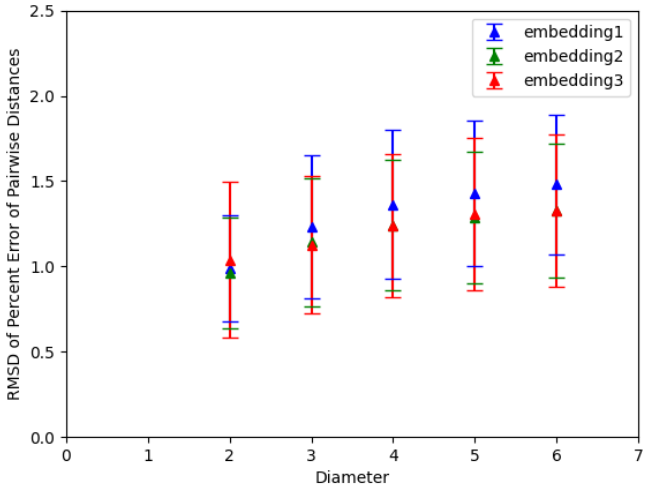


Figure 49: $n = 8$ $RMSD_{error}$

Graph Generator Code:

```
from itertools import combinations
import sys
import networkx as nx
from networkx.algorithms import isomorphism
import numpy as np
import copy

class graphGenerator:
  N = 0
  numAdd = 0
  storedList = []

  def __init__(self, N):
    self.N = int(N)
    self.getAllGraph(self.N)

  #return False if currentGraph is not isomorphic to any previous graph; True
   otherwise
  def checkUndirectedIsomorphic(self, currentEdgeList, nonIsoGraphList):
    #create empty networkx graph that we are currently looking at
    currentGraph = nx.Graph()
    currentGraph.add_edges_from(currentEdgeList)

    #compare current graph to all previous non-isomorphic undirected graphs
    for prevGraph in nonIsoGraphList:
      GM = isomorphism.GraphMatcher(prevGraph, currentGraph)

      #if isomorphic return return true
      if GM.is_isomorphic():
        return (nonIsoGraphList, True)

    #at this point, current graph is not isomorphic to any previous graphs
    #add graph to list of non-isomorphic undirected graphs
    nonIsoGraphList.append(currentGraph)
    return (nonIsoGraphList, False)

  def checkDirectedIsomorphic(self, currentEdgeList, nonIsoDirectedGraphList):
    #create empty networkx graph that we are currently looking at
    currentGraph = nx.DiGraph()
    currentGraph.add_edges_from(currentEdgeList)

    #compare current graph to all previous non-isomorphic undirected graphs
    for prevGraph in nonIsoDirectedGraphList:
      DiGM = isomorphism.DiGraphMatcher(prevGraph, currentGraph)

      #if isomorphic return return true
      if DiGM.is_isomorphic():
        return (nonIsoDirectedGraphList, True)

    #at this point, current graph is not isomorphic to any previous graphs
    #add graph to list of non-isomorphic undirected graphs
```

```
    nonIsoDirectedGraphList.append(currentGraph)
    return (nonIsoDirectedGraphList, False)

  def checkCycle(self, directed, N):
    #constructs directed graph
    G = nx.DiGraph(directed)

    #gets all cycles from directed graph
    cycles = list(nx.simple_cycles(G))

    #if len(cycles) is 0 there were no
    if len(cycles) == 0:
      return False
    else:
      print("cycle")
      return True

  def updateDegree(self, N, temp, finished):
    outDegree = {}

    for i in range(N):
      outDegree[i] = 0

    #if the vA is connected to vB and vB has been visited then increase the
   out degree of vA
    for i in temp:
      if i[1] in finished:
        outDegree[i[0]]+=1

    return outDegree

  def updateFinished(self, outDegree, finished, N):
    #goes through vertices and counts which ones have 2 outgoing edges
    for i in range(N):
      if outDegree[i] == 2 and i not in finished:
        finished.append(i)

    return finished

  #return False if H1 construction does not exist; True otherwise
  def checkH1Construction(self, prevFin, finished):
    #prevFin = list before next inductive step, finished = list after next
   inductive step

    #by H1 construction, at least one vertex is added at each inductive step
    #so if a vertex is added len(finished) should not equal len(prevFin)
    if len(prevFin) == len(finished):
      return False

    return True

  def countWrong(self, outDegree, N, leader, co_leader):
    #goes through vertices and checks to see whether there is an unfinished
   follower
```

```
    #unfinished follower is one that doesn't have out-degree = 2
    for i in range(N):
      if outDegree[i] != 2 and i != leader and i != co_leader:
        return False
    return True

  def directedMatrix(self, undirectedEdgeList, N):
    outDegree = dict.fromkeys(range(N),0)
    allDirectedFormations = []

    #go through all starting edges
    for i in undirectedEdgeList:
      #consider starting edge and it's reverse
      for k in range (2):
        directed = []
        finished = []

        #try both directions of starting edge to ensure getting all possible
    combinations
        if k == 0:
          leader = i[0]
          co_leader = i[1]
        else:
          leader = i[1]
          co_leader = i[0]

        #leader and co-leader start finished
        finished.append(leader)
        finished.append(co_leader)

        #goes through all neighbors of leader and co-leader and reverses edges
        for j in undirectedEdgeList:
          if j[0] == leader or (j[0] == co_leader and j[1] != leader):
            directed.append((j[1], j[0]))
            outDegree[j[1]]+=1
          else:
            directed.append(j)

        #update out-degree of each vertex and finished list
        outDegree = self.updateDegree(N, directed, finished)
        finished = self.updateFinished(outDegree, finished, N)

        prevFin = []
        #while there is still an unfinished vertex and while a new vertex is
    added @ every inductive step, continue
        while self.countWrong(outDegree, N, leader, co_leader) == False and
    self.checkH1Construction(prevFin, finished) == True:
          prevFin = copy.copy(finished)

          #point edges towards finished vertices
          for j in directed:
            if j[0] in finished and j[1] not in finished:
              directed[directed.index(j)] = (j[1], j[0])
              outDegree[j[1]]+=1
```

```
153
154             if outDegree[j[1]] == 2:
155               finished.append(j[1])
156
157         #update out-degrees and finished vertices after current inductive
    step is complete
158         outDegree = self.updateDegree(N, directed, finished)
159         finished = self.updateFinished(outDegree, finished, N)
160
161       #either the while loop ends successfully or there was no H1
    construction
162       #if H1 construction exists and while loop ended successfully, add
    formation to list
163       if self.checkH1Construction(prevFin, finished) == True:
164         allDirectedFormations.append(directed)
165
166     #return allDirectedFormation -> list of all directed graphs under one
    undirected graph
167     return allDirectedFormations
168
169   def inductiveGenerationH1Graphs(self, N):
170     N = int(N)
171     baseVertexList = []
172     edgeLists = []
173     filename = "n" + str(N-1) + ".csv"
174     newVertex = N-1
175
176     baseVertexList = list(range(0, N-1))
177
178     #gets all combinations of 2 vertices (0, n-1)
179     comb = list(combinations(baseVertexList, 2))
180
181     undirected = []
182
183     #read in non-iso undirected graphs from n-1 vertices
184     with open(filename) as file:
185       edges = file.readlines()
186     edges = [x.strip() for x in edges]
187
188     #convert edges read in to points (i.e. read in [0,1,2,3] want [(0,1),(2,3)
    ])
189     for i in edges:
190       pointList = []
191       splitList = i.split(",")
192       splitList.pop(len(splitList)-1)
193       splitList = [int(x) for x in splitList]
194
195       for j in range(0, len(splitList), 2):
196         pointList.append((splitList[j], splitList[j+1]))
197       undirected.append(pointList)
198
199     #goes through all undirected non-iso graphs for n-1 and adds two edges to
    each one
200     #two edges are vertex N-1 being added to two other vertices in the graph
```

```
201     #adding vertex N-1 to the undirected non-iso graph in every way under H1
    construction laws
202     for i in undirected:
203       for j in comb:
204         baseGraph = copy.copy(i)
205         baseGraph.append((newVertex, j[0]))
206         baseGraph.append((newVertex, j[1]))
207 
208         edgeLists.append(baseGraph)
209 
210     return edgeLists
211 
212   def getAllGraph(self, N):
213     #if n > 4 use n-1 graphs to generate non-isomorphic graphs for n
214     if N > 4:
215       edgeLists = self.inductiveGenerationH1Graphs(N)
216     else:
217       #create list of vertices (0, N)
218       vertexList = list(range(0, N))
219 
220       #gets all the possible edgelist combinations on 2n-3 edges (all possible
    minimally rigid frameworks)
221       comb = combinations(vertexList, 2)
222       edgeNumber = 2*len(vertexList) - 3
223       edgeLists = list(combinations(comb, edgeNumber))
224 
225     #create file to write edgeLists too
226     filename = "n" + str(N) + ".csv"
227     textFile = "n" + str(N) + ".txt"
228     fileEdgeList = open(filename, "w")
229     textEdgeList = open(textFile, "w")
230     directedFormationFile = open("edges.csv", "w")
231 
232     #create first graph and add it to list of non-isomorphic undirected graphs
233     firstGraph = nx.Graph()
234     firstGraph.add_edges_from(edgeLists[0])
235     nonIsoGraphList = [firstGraph]
236 
237     #add first edge list to list of non-isomorphic undirected graph edgelists
238     nonIsoUndirected = [edgeLists[0]]
239 
240     #go through all undirected graphs
241     for currentEdgeList in edgeLists:
242       #if graph we are currently looking at is isomorphic to previous graph do
    nothing; otherwise add to nonIsoUndirected
243       if self.checkUndirectedIsomorphic(currentEdgeList, nonIsoGraphList)[1]
   == False:
244         nonIsoUndirected.append(currentEdgeList)
245 
246     #write edges in file
247     headers = "source, sink\n"
248     directedFormationFile.write(headers)
249     total = 0
250 
```

```
251     #for loop goes through each non-isomorphic undirected graph
252     for g in nonIsoUndirected:
253       #write full non-isomorphic undirected graph edge lists to textEdgeList
254       textEdgeList.write(str(g) + "\n")
255 
256       #writes individual edges of each non-isomorphic undirected graph to
    fileEdgeList
257       for e in g:
258         fileEdgeList.write(str(e[0]) + "," + str(e[1]) + ",")
259       fileEdgeList.write("\n")
260 
261       #gets directed graphs for observed undirected graph
262       diM = self.directedMatrix(g, int(N))
263 
264       directedFormationFile.write("new undirected graph \n")
265 
266       #create first graph and add it to list of non-isomorphic undirected
    graphs
267       firstGraph = nx.DiGraph()
268       firstGraph.add_edges_from(diM[0])
269 
270       #add first edge list to list of non-isomorphic undirected graph
    edgelists
271       nonIsoDirectedGraphList = [firstGraph]
272       directed = [diM[0]]
273 
274       #writes first formation to edges.csv file
275       for k in diM[0]:
276         directedFormationFile.write(str(k[0]) + "," + str(k[1]) + ",\n")
277       directedFormationFile.write("\n")
278 
279       persAcyLF = []
280 
281       for j in diM:
282         #if current formation is not isomorphic to any previous formations,
    and formation contains no cycles add to list
283         if self.checkDirectedIsomorphic(j, nonIsoDirectedGraphList)[1] ==
    False and not self.checkCycle(j, int(N)):
284           directed.append(j)
285           persAcyLF.append(j)
286 
287           #write edges of formation to file
288           for k in j:
289             directedFormationFile.write(str(k[0]) + "," + str(k[1]) + ",\n")
290           directedFormationFile.write("\n")
291       total+=len(directed)
292     #writes total number of all formations for some value of n
293     directedFormationFile.write("total\n")
294     directedFormationFile.write(str(total))
295 
296     directedFormationFile.close()
297 
298     return persAcyLF
299 
```

```
if __name__ == '__main__':
  graphGenerator(sys.argv[1])
```

Leader-Follower Simulation Code:

```
#Import Robotarium Utilities
import rps.robotarium as robotarium
from rps.utilities.transformations import *
from rps.utilities.graph import *
from rps.utilities.barrier_certificates import *
from rps.utilities.misc import *
from rps.utilities.controllers import *

#Other Imports
import numpy as np
import sys
import math

#calculates distance between two points
def distance(source_xy, target_xy):
  x_dist = math.pow(source_xy[0] - target_xy[0], 2)
  y_dist = math.pow(source_xy[1] - target_xy[1], 2)
  distance = math.sqrt(x_dist+y_dist)
  return distance

def distanceMatrix(endWeights, poses):
  #calculates distance between all vertices (edge and non-edge)
  for i in range(0, len(L)):
    for j in range(0, len(L)):
      if i != j:
        dist = distance([poses[0][i], poses[1][i]], [poses[0][j], poses[1][j
    ]])
        endWeights[i][j] = dist
        average[i][j] = dist + average[i][j]
        error[i][j] = (abs((dist - distMatrix[i][j]) / distMatrix[i][j])) +
    error[i][j]
        if L[i][j] != 0:
          distDiff[i][j]+= math.pow((L[i][j] - dist), 2)
        else:
          distDiff[i][j]+= math.pow((L[j][i] - dist), 2)

  return endWeights

def undirectedMatrix(L):
  #makes directed matrix undirected
  for i in range (N):
    for j in range (N):
      if L[i][j] == 0 and i != j:
        L[i][j] = L[j][i]

  return L

def calcWaypoints():
```

```
47    n = 20 #number of points you want generated
48    r = 0.8 #radius of circle
49    pi = math.pi
50    x = []
51    y = []
52
53    #calculates 20 points (n) around a circle of radius 0.8 (r)
54    points = [(math.cos(2*pi/n*x)*r, math.sin(2*pi/n*x)*r) for x in range(0,n+1)
       ]
55
56    #goes through points and splits them into x and y lists
57    for i in range (0, len(points)):
58      x.append(points[i][0])
59      y.append(points[i][1])
60
61    return x, y
62
63  def calculateDataAndWriteToFile(numIter):
64    #calculates all data values
65    for i in range (len(average)):
66      for j in range (len(average)):
67        if i != j:
68          average[i][j] = average[i][j] / numIter
69          distDiff[i][j] = math.sqrt(distDiff[i][j] / numIter)
70          error[i][j] = error[i][j] / numIter
71
72    #writes data values to file
73    writeDataToFile(fileDist, "average", average)
74    writeDataToFile(fileDist, "error", error)
75    writeDataToFile(fileDist, "RMSE", distDiff)
76
77    fileDist.close()
78    filePos.close()
79
80  def writeDataToFile(file, keyword, dataList):
81    file.write(keyword + ": ")
82    file.write("\n")
83
84    #goes through matrix and writes data to file
85    for i in dataList:
86      for j in i:
87        file.write(str(j) + ",")
88      file.write("\n")
89    file.write("\n")
90
91  # Experiment Constants
92  iterations = 7000 #Run the simulation/experiment for 5000 steps (5000*0.033 ~=
       2min 45sec)
93  N=int(sys.argv[1]) #Number of robots to use, this must stay 4 unless the
      Laplacian is changed.
94  initCondFile = sys.argv[2]
95  close_enough = 0.05 #How close the leader must get to the waypoint to move to
      the next one.
96  undirected = int(sys.argv[3])
```

```
97
98  # For computational/memory reasons, initialize the velocity vector
99  dxi = np.zeros((2,N))
100
101 #Initialize leader state
102 state = 0
103
104 #Limit maximum linear speed of any robot
105 magnitude_limit = 0.15
106
107 # Create gains for our formation control algorithm
108 formation_control_gain = 10
109 desired_distance = 0.3
110
111 #calculate and create waypoints matrix (these are the destination points)
112 x, y = calcWaypoints()
113 waypoints = np.array([x, y])
114 numberOfPoints = len(waypoints[0])
115
116 rows, cols = (N, N)
117 L = [[0 for i in range(cols)] for j in range(rows)]
118 startingXDist = [[0 for i in range(cols)] for j in range(rows)]
119 startingYDist = [[0 for i in range(cols)] for j in range(rows)]
120
121 with open(initCondFile) as f:
122     content = f.readlines()
123 content = [x.strip() for x in content]
124
125 tempX = content[1].split(",")
126 tempY = content[3].split(",")
127 xInit, yInit = [], []
128
129 #initialize starting x/y-coord
130 for i in range (N):
131   xInit.append(float(tempX[i]))
132   yInit.append(float(tempY[i]))
133
134 # Initial Conditions to Avoid Barrier Use in the Beginning.
135 third_vector = np.zeros(N)
136 initial_conditions = np.array([xInit, yInit,third_vector])
137
138 for i in range(len(content)):
139   if content[i] == "initial distances:":
140     tempIndex = 0
141     #read in initial distances between robots (graph edges only) and puts
    distances in matrix L
142     for j in range (i+1, i+1+N):
143       splitLine = content[j].split(",")
144       for k in range (N):
145         L[tempIndex][k] = float(splitLine[k])
146       tempIndex+=1
147   elif content[i] == "Leader:":
148     #stores leader robot id
149     leader = int(content[i+1])
```

```
150
151 L = np.array(L)
152
153 #UNDIRECTED CASE
154 if undirected == 1:
155   L = undirectedMatrix(L)
156
157 rows, cols = (len(L), len(L))
158 distMatrix = [[0 for i in range(cols)] for j in range(rows)]
159
160 #calculates initial distances between all vertices (edge and non-edge)
161 for i in range(len(L)):
162   for j in range(len(L)):
163     distMatrix[i][j] = distance((xInit[i], yInit[i]), (xInit[j], yInit[j]))
164 distMatrix = np.array(distMatrix)
165
166 # Instantiate the Robotarium object with these parameters
167 r = robotarium.Robotarium(number_of_robots=N, show_figure=False,
    initial_conditions=initial_conditions, sim_in_real_time=True)
168
169 # Grab Robotarium tools to do simgle-integrator to unicycle conversions and
    collision avoidance
170 # Single-integrator -> unicycle dynamics mapping
171 _,uni_to_si_states = create_si_to_uni_mapping()
172 si_to_uni_dyn = create_si_to_uni_dynamics()
173 # Single-integrator barrier certificates
174 si_barrier_cert = create_single_integrator_barrier_certificate_with_boundary()
175 # Single-integrator position controller
176 leader_controller = create_si_position_controller(velocity_magnitude_limit
    =0.1)
177
178 rows, cols = (len(L), len(L))
179 endWeights = [[0 for i in range(cols)] for j in range(rows)]
180 average = np.zeros([rows, cols], dtype = float)
181 distDiff = np.zeros([rows, cols], dtype = float)
182 error = np.zeros([rows, cols], dtype = float)
183
184 #set up files to write data to
185 fileDist = open("distances.csv", "w")
186 filePos = open("lfPos.csv", "w")
187 headers = "iteration,id,distances\n"
188
189 for t in range(iterations):
190   fileDist.write(headers)
191
192   #writes distance matrix at this iteration to distances.csv
193   endWeights = distanceMatrix(endWeights, r.poses)
194   index = 0
195   for ind in endWeights:
196     tempStr = ""
197     for c in ind:
198       tempStr+=(str(c) + ", ")
199
200     fileDist.write(str(t) + "," + ("robot" + str(index)) + "," + tempStr)
```

```
201     fileDist.write("\n")
202     index+=1
203   fileDist.write("\n")
204 
205   # Get the most recent pose information from the Robotarium. The time delay
       is
206   # approximately 0.033s
207   x = r.get_poses()
208   xi = uni_to_si_states(x)
209 
210   headers = "iteration,id,x,y\n"
211   filePos.write(headers)
212 
213   #writes position of each robot to lfPos.csv
214   for i in range (N):
215     filePos.write(str(t) + "," + ("robot" + str(i)) + "," + str(r.poses[0][i])
       + "," + str(r.poses[1][i]))
216     filePos.write("\n")
217   filePos.write("\n")
218 
219   #Follower movement
220   for i in range(1,N):
221     # Zero velocities and get the topological neighbors of agent i
222     dxi[:,[i]]=np.zeros((2,1))
223     neighbors = topological_neighbors(L,i)
224 
225     for j in neighbors:
226       dxi[:,[i]] += formation_control_gain*(np.power(np.linalg.norm(x[:2,[j]]-
      x[:2,[i]]), 2)-np.power(desired_distance, 2))*(x[:2,[j]]-x[:2,[i]])
227 
228   #Leader movement
229   waypoint = waypoints[:,state].reshape((2,1))
230   dxi[:,[leader]] = leader_controller(x[:2,[leader]], waypoint)
231 
232   #if leader is close enough to the next waypoint (destination point) move to
       the next waypoint
233   if np.linalg.norm(x[:2,[leader]] - waypoint) < close_enough:
234     state = (state + 1)%numberOfPoints
235     #if state == 0 then full path has been completed so write all data and
      exit
236     if state == 0:
237       fileFinished = open("finishedRoute.txt", "a")
238       fileFinished.write(initCondFile + "\n")
239       calculateDataAndWriteToFile(t)
240       exit(0)
241 
242   # Keep single integrator control vectors under specified magnitude
243   # Threshold control inputs
244   norms = np.linalg.norm(dxi, 2, 0)
245   idxs_to_normalize = (norms > magnitude_limit)
246   dxi[:, idxs_to_normalize] *= magnitude_limit/norms[idxs_to_normalize]
247 
248   #Use barriers and convert single-integrator to unicycle commands
249   dxi = si_barrier_cert(dxi, x[:2,:])
```

```
  dxu = si_to_uni_dyn(dxi,x)

  # Set the velocities of agents 1,...,N to dxu
  r.set_velocities(np.arange(N), dxu)

  # Iterate the simulation
  r.step()

calculateDataAndWriteToFile(t)
```

Result Chart Generation Code:

```

import matplotlib.pyplot as plt
import numpy as np
import sys, copy, math

class standardDeviationCharts:
  def __init__(self, N):
    self.N = int(N)

    #(directory of error file, plot label for y axis, name of png file that plot is printed to)
    directory1 = ("/Ranking/avgRmseCorrectRanking.csv", 'RMSD of Distance to Correct Position', 'rmseCorrect.png')
    directory2 = ("/Ranking/OnlyEdges/avgErrorRmseOnlyEdgesRanking.csv", 'RMSD of Percent Error of Pairwise Distances (Only Edges)', 'rmseErrorOnlyEdges.png')
    directory3 = ("/Ranking/OnlyEdges/avgRmseOnlyEdgesRanking.csv", 'RMSD of Pairwise Distances (Only Edges)', 'rmseOnlyEdges.png')
    directory4 = ("/Ranking/EdgeAndNonEdge/avgErrorRmseRanking.csv", 'RMSD of Percent Error of Pairwise Distances', 'rmseError.png')
    directory5 = ("/Ranking/EdgeAndNonEdge/avgRmseRanking.csv", 'RMSD of Pairwise Distances', 'rmse.png')
    directory6 = ("/Ranking/EdgeAndNonEdge/percentErrorRanking.csv", 'Percent Error of Pairwise Distances', 'error.png')
    directory7 = ("/Ranking/OnlyEdges/percentErrorOnlyEdgesRanking.csv", 'Percent Error of Pairwise Distances (Only Edges)', 'errorOnlyEdges.png')

    directoryList = [directory1, directory2, directory3, directory4, directory5, directory6, directory7]

    markerColors = ['b', 'g', 'r']

    #walks through error file and creates charts for each of the 7 error types
    for j in directoryList:
      plt.xlabel('Diameter')

      #set limits of plot
      plt.ylim(0, 2.5)
      plt.xlim(0, self.N-1)

      #go through all three embeddings and plot their average error values
```

```
    individually
      for i in range (1, 4):
        x, y, sdList = self.createCharts("finishedSim/n" + str(self.N) + j[0], i)
        plt.errorbar(x, y, sdList, linestyle='none', marker='^', c=markerColors[i-1], mfc=markerColors[i-1], mec=markerColors[i-1], capsize=5, label=("embedding" + str(i)))

      plt.ylabel(j[1])

      plt.legend()
      plt.savefig("finishedSim/n" + str(self.N) + "/" + j[2])
      plt.close('all')

  def createCharts(self, directory, embedding):
    #read content of error file
    with open(directory) as f:
        content = f.readlines()
    content = [x.strip() for x in content]

    diamValues = [], []
    avgValues, numFormations = np.zeros(9), np.zeros(9)
    errorLists = [[] for x in range(10)]
    added = False

    for i in range (len(content)):
      #if at even spot in content we are looking at edge name (not RMSD value)
      if i%2 == 0:
        splitLine = content[i].split(":")[0].split(" ")
        initialCondFile = "initialCond_" + splitLine[4] + splitLine[2] + "_" + splitLine[3][-1]
        initialCondDirectory = "finishedSim/n" + str(self.N) + "/" + splitLine[3] + "/" + initialCondFile + "/" + initialCondFile + ".csv"

        #make sure we are observing desired embedding
        if int(splitLine[3][-1]) == embedding:
          #reads in and cleans initial condition file
          with open(initialCondDirectory) as f:
            init = f.readlines()
          init = [x.strip() for x in init]
          init = [x.replace(",", "") for x in init]

          #gets diameter of observed graph from initialCond file
          diam = int(init[init.index("Graph Diameter:") + 1])

          #adds diameter value to end of list
          diamValues.append(diam)

          #add one (counting all formations for each diameter value for each embedding)
          numFormations[diam-1]+=1
          added = True
      else:
        #if at odd spot in content just add float RMSD value to end of list
```

```
        if added == True:
          toAdd = float(content[i])

          #add error value to correct diameter list
          errorLists[diamValues[-1]-1].append(toAdd)

          #avgValues contains sum of all average
          avgValues[diamValues[-1]-1]+=toAdd
          added = False

    totalFormations = 0

    #calculates mean error value for each diameter
    for i in range (len(avgValues)):
      size = len(diamValues)

      if numFormations[i] != 0:
        avgValues[i] = avgValues[i]/numFormations[i]

      totalFormations+=numFormations[i]

    #x contains diameters, y contains avg error values, sdList contains
    standard deviation values
    x, y, sdList = [], [], []

    #calculates standard deviation for each diameter
    for i in range (len(avgValues)):
      sd = 0
      if avgValues[i] != 0:
        for j in errorLists[i]:
          sd+=math.pow((j - avgValues[i]), 2)
        sd = math.sqrt(sd/len(errorLists[i]))
        sdList.append(sd)

        x.append(i+1)
        y.append(avgValues[i])

    return (x, y, sdList)

if __name__ == '__main__':
    standardDeviationCharts(sys.argv[1])
```

Embedding Generation Code:

```
import random, sys
import numpy as np

"""
generateRandStartingPoints.py generates random (x,y) coords for each vertex
"""
class generateRandStartingPoints:
  def __init__(self, N):
    #N = number of vertices in graph
```

```
    self.N = int(N)

    #initialX -> x coordinates, initialY -> y coordinates
    initialX = np.zeros(self.N)
    initialY = np.zeros(self.N)

    #generates random (x, y) values for every vertex
    for j in range(self.N):
      if j != 0:
        initialX[j] = random.uniform(-0.5, 0.5)
        initialY[j] = random.uniform(0.5, 0.9)

    initialX[0] = 0
    initialY[0] = 0.3

    initialX[1] = 0
    initialY[1] = 0.5

    #writes generated x and y coordinates to file
    file = open("randomPoints.csv", "w")
    for i in range(self.N):
      file.write(str(initialX[i]) + ",")
    file.write("\n")
    for i in range(self.N):
      file.write(str(initialY[i]) + ",")

if __name__ == '__main__':
    generateRandStartingPoints(sys.argv[1])
```

References

[1] Alonso-Mora, Javier, Eduardo Montijano, Mac Schwager, and Daniela Rus. “Distributed Multi-Robot Formation Control among Obstacles: A Geometric and Optimization Approach with Consensus.” In 2016 IEEE International Conference on Robotics and Automation (ICRA), 5356–63. Stockholm, Sweden: IEEE, 2016. https://doi.org/10.1109/ICRA.2016.7487747.

[2] Burns A, St. John A, Klemperer P, Solyst J. Redundant Persistent Acyclic Formations for Vision-Based Control of Distributed Multi-Agent Formations. Proceedings of the 31st Canadian Conference on Computational Geometry. 2019:29-35.

[3] Burns A, Schulze B, St. John A. Persistent Multi-robot Formations with Redundancy. In: Groß R, Kolling A, Berman S, et al., eds. Distributed Autonomous Robotic Systems. Vol 6. Springer Proceedings in Advanced Robotics. Springer International Publishing; 2018:133-146. doi:10.1007/978-3-319-73008-0_10

[4] Eren, Tolga. “Shape Control of Cyclic Networks in Multirobot Formations.” Turkish Journal of Electrical Engineering & Computer Sciences no 21 (2013): 1182 – 1198. doi:10.3906/elk-1111-46.

[5] Graver, Jack E. Counting on Frameworks: Mathematics to Aid the Design of Rigid Structures. Cambridge University Press, 2001.

[6] Haghighi, R., and C. C. Cheah. “Multi-Group Coordination Control for Robot Swarms.” Automatica 48, no. 10 (October 1, 2012): 2526–34. https://doi.org/10.1016/j.automatica.2012.03.028.

[7] Hendrickx, J.M., B.D.O. Anderson, and V.D. Blondel. “Rigidity and Persistence of Directed Graphs.” In Proceedings of the 44th IEEE Conference on Decision and Control, 2176–81. Seville, Spain: IEEE, 2005. https://doi.org/10.1109/CDC.2005.1582484.

[8] Hendrickx, Julien M., Brian D. O. Anderson, Jean-Charles Delvenne, and Vincent D. Blondel. “Directed Graphs for the Analysis of Rigidity and Persistence in Autonomous Agent Systems.” International Journal of Robust and Nonlinear Control 17, no. 10–11 (July 10, 2007): 960–81. https://doi.org/10.1002/rnc.1145.

[9] Henneberg, Lebrecht. Die graphische Statik der starren Systeme. Vol. 31. BG Teubner, 1911.

[10] Inglett, J. E., and E. J. Rodríguez-Seda. “Object Transportation by Cooperative Robots.” In SoutheastCon 2017, 1–6, 2017. https://doi.org/10.1109/SECON.2017.7925348.

[11] Jackson, Bill, and J.C. Owen. “Equivalent Realisations of a Rigid Graph.” Discrete Applied Mathematics 256 (March 2019): 42–58. https://doi.org/10.1016/j.dam.2017.12.009.

[12] Jäger, Markus, and Bernhard Nebel. “Decentralized Collision Avoidance, Deadlock Detection, and Deadlock Resolution for Multiple Mobile Robots.” In In IROS, 1213–19, 2001.

[13] Kube, C. Ronald, and Eric Bonabeau. “Cooperative Transport by Ants and Robots.” Robotics and Autonomous Systems 30, no. 1 (January 31, 2000): 85–101. https://doi.org/10.1016/S0921-8890(99)00066-4.

[14] Laman, G. “On Graphs and Rigidity of Plane Skeletal Structures.” Journal of Engineering Mathematics 4, no. 4 (October 1970): 331–40. https://doi.org/10.1007/BF01534980.

[15] Nixon, Anthony, and Elissa Ross. “One Brick at a Time: A Survey of Inductive Constructions in Rigidity Theory.” ArXiv:1203.6623 [Math], June 17, 2013. http://arxiv.org/abs/1203.6623.

[16] Pollaczek-Geiringer, Hilda (1927), ”Über die Gliederung ebener Fachwerke”, Zeitschrift für Angewandte Mathematik und Mechanik, 7 (1): 58–72.

[17] Raghunathan, D., and J. Baillieul. “Motion Based Communication Channels between Mobile Robots - A Novel Paradigm for Low Bandwidth Information Exchange.” In 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, 702–8, 2009. https://doi.org/10.1109/IROS.2009.5354808.

[18] Ren, Wei, and Nathan Sorensen. “Distributed Coordination Architecture for Multi-Robot Formation Control.” Robotics and Autonomous Systems 56, no. 4 (April 30, 2008): 324–33. https://doi.org/10.1016/j.robot.2007.08.005.

[19] S. Wilson et al., ”The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems,” in IEEE Control Systems Magazine, vol. 40, no. 1, pp. 26-44, Feb. 2020.

[20] Tuci, Elio, Muhanad H. M. Alkilabi, and Otar Akanyeti. “Cooperative Object Transport in Multi-Robot Systems: A Review of the State-of-the-Art.” Frontiers in Robotics and AI 5 (2018). https://doi.org/10.3389/frobt.2018.00059.

[21] Wang, Hua, and Yi Guo. Minimal Persistence Control on Dynamic Directed Graphs for Multi-Robot Formation, n.d.

[22] Wang, Zijian, and Mac Schwager. “Kinematic Multi-Robot Manipulation with No Communication Using Force Feedback.” In 2016 IEEE International Conference on Robotics and Automation (ICRA), 427–32. Stockholm, Sweden: IEEE, 2016. https://doi.org/10.1109/ICRA.2016.7487163.

[23] Yamashita, Atsushi, Tamio Arai, Jun Ota, and Hajime Asama. “Motion Planning of Multiple Mobile Robots for Cooperative Manipulation and Transportation.” IEEE Transactions on Robotics and Automation 19 (2003): 223–37.