Decentralized Optimal Leader Election for Shortest-Path Routing in Mobile Ad Hoc Networks

Daniel Rodrigues *Harvard University*
rodriguesmd@college.harvard.edu

Abstract

We present an algorithm[1] to elect a leader such as to optimize information distribution from leader to followers in mobile multi-agent robotic systems; the proposed algorithm is simple and only requires agents to communicate with their neighbors yet resilient to topology changes (thus applicable to mobile ad hoc networks). The algorithms ensure that eventually each agent in each connected component will locally determine the set of most optimal leaders.

Index Terms

Multi-agent systems; leader election; mobile ad hoc routing

I. Note for Final Project

Initially when pursuing dynamic leader election, I thought of exploring a PageRank-like algorithm. After trying some PageRank ideas without success, I considered persisting with PageRank or pivoting. When I was inspired with a potential packet based idea, I choose to pivot in favor of an approach that is naturally decentralized.

II. Introduction

Multi-agent robotic systems are on of the great driving forces of decentralized algorithms; however many time it is required or just more practical to have a leader in the system that can delegate work, unify information, and run algorithm that cannot be or have not yet been decentralized. Leader follower networks allows for efficient and scalable propagation of external operator commands and can enable swarms to act on information or state that may be limited to one or a few leaders.

Here we present a simple yet resilient completely decentralized algorithm that ensures that each agent eventually identifies all the optimal leader for its connected component where the leader is optimal for information distribution between the leader and their followers; the algorithm ensures that when the network topology changes, each leader updates its identification of the optimal leaders . The suggested algorithm allows for communication link transmission times (weighted edges) and provides a procedure for establishing shortest path routing. This is extremely useful for systems where communication within a swarm or between an operator and the swarm is a premium, and further useful for systems where communication times cannot be accurately modeled by euclidean based models.

We evaluate this problem in the context of a mobile ad hoc network where pair of agents (i.e. nodes) can communicate if there exists a path between in the graph that represents the communication topology. An agent can only send packets to its neighbors represented by an edge between the two agents. Communication links, represented by edges, can change over time due to persistence failures or mobility of agents and limitations of hardware, as well as any number of other issues.

In line with traditional and recent work in leader election, we adopt the leader election problem that each connected component whose topology is static for a sufficiently amount of time should eventually

[1]https://github.com/Danieltech99/CS-222-Final-Project

have at least one leader. We extend this problem by adding the notion of optimality, where the most optimal leaders are those that can communicate in the smallest time with all its respective followers. In this paper, we leave the usage of a identified leader up to the implementer, instead we ensure that each agent identifies the same set of all optimal leaders for their connected component; the implementer can then use a tie-breaking mechanism to determine which leader they choose if there is more than one optimal leader per connected component.

III. Related Work

A number of contributions in traditional networking and robotic systems have led to the question we are considering. Graph theory provides a strong foundation for formalization of communication optimality problems through ideas such as a graph's diameter and a graph's center [1]. Well established research in graph theory and networking has presented on graph centers/centriods, shortest path routing, optimizing network traffic and load balancing [2], some applying and responding to concepts formulated by graph theory, but mainly for fixed topologies. Foundational leader election algorithms in ad hoc networks have been long established and were explored in [3]. Recent work in swarm intelligence and flocking, like that of [4], has made great strides in asking questions regarding leader elections in topologies where the number of connected components can change over time and in the context of Non Compulsory protocals, but disregarding optimality of leader selection; this work serves as a major inspiration for our work. Further work has explored the Compulsory protocal setting where agents are repositioned with the goal of minimizing the distance between hosts and services [5]. Optimizing leader selection has more recently been explored in a similar optimization context where leaders were chosen to optimize convergence in terms of mixing time [6]. These approaches have made great progress but require close integration between the leader election algorithm and motion planning or formation control. These papers provide a starting point for our research but fail to address the question of optimal leader selection in the context of a dynamic topology where the leader election algorithm has no control over mobility and network communication is the premium. Reference [6] shares many of the motivations of our paper, however it focuses more on optimizing the convergence problem while we focus on longest of communication time in the context of message relays and commands instead of mixing.

IV. Definitions

A. Model and Assumptions

In this paper, a system consists of n independent mobile agents. A agent can send packets and communicate directly with its neighbors as defined by the agents hardware capabilities. The communication topology or network is represented by an undirected weighted graph G where the agents are vertices and the edges between two vertices represent direct communication ability between the agents; the weight of the edge is most naturally the average amount of time it takes for a packet to cross the communication link; however the implementer can customize the aspect of optimality by changing what the edge weight represents; if the number of agents the packet goes through should be minimized (as is the case where packet processing is the dominating factor over link travel time), then the implementer can assign a weight of 1 to each edge. The graph can change over time as the communication abilities of the agents change; this could be the result of physical rearrangement, persistent link failures, or any other number of issues. The graph is not necessarily connected and we refer to each connected component as a flock; flocks can split (into two separate connected components) and merge (into one connected component).

The proposed algorithm makes the following assumptions: each agent has a unique identifier; all communication links are bidirectional, constant travel time, and FIFO; no packets are dropped; a mechanism is in place for each agent to detect the identifiers of its neighbors at any given time and the travel time of the communication link for that neighbor (the weight of the edge); and processing time of packets is arbitrarily small in comparison to travel time over links (including computation triggered by packets).

Most of these constraints can be relaxed by modification of the presented algorithm: the undirected constraint; constant travel time; packet drops; processing time. The constant travel time constraint can be relaxed by correctly differentiating between small random difference in travel time and an edge weight update, for example by using a threshold. The packet drop assumption can be relaxed by implementing well studied acknowledgement packets and determining when to resend and when to remove a repeatedly failing communication link (remove edge). The processing time assumption can be relaxed by modifying the algorithm to add process time to the length of a path instead of only the sum of communication links in the path. We leave these relaxations and others to future works.

B. Problem Statement

Given a communication network between agents in a multi-agent system, how can we ensure that each connected component has a leader such that the leader is the agent which can communicate with all followers the quickest? Specifically, we want a leader that has the smallest communication time to all its followers which is the smallest communication time to its furthest follower.

We denote the graph that represents the communication network as G, the center of the graph as $center(G)$, and the graph diameter as $diam(G)$.

We break the election problem into two parts: electing an optimal leader and then updating which node is a leader as the topology changes. The entire algorithm consists of four major sequences (that sometimes run in parallel): establish shortest-path routing on each agent; use shortest-path routing to identify the graph center(s) - the set optimal leaders - by having each agent share their the longest shortest-path (lsp) to the rest of the flock (connected component); updating shortest-path routing when the topology changes; correcting and re-executing step 2. The first part of the problem can be presented as identifying the graph center in a decentralized fashion. The second part is approached by responding to when edges are added or removed (and by extension, when agents are connected, disconnected).

Because multi-robot systems can often consist of light-weight hardware with extremely limited hardware, specifically RAM/storage, we preferred sending more network packets over increasing storage complexity or packet size. In our approach, each agent cannot store packets that they have already processed, and processed packets in a FIFO pattern. Each agent uses an $O(n)$ space complexity table for routing, flock agents' lsp, and neighbor storage; the agent stores one layer of neighbor history. Every packet is one layer, meaning metadata size is constant and packet size does not change relative to route path size. The works in this paper are not optimized in terms of runtime or space complexity, but rather a deployable and simple proof of concept. We also recognize that there can be large reductions in number of network packets and system time complexity by implementing a more complex algorithm that leverages a greater order storage complexity; we believe this is a valid approach for future works, but not the original approach we wanted to take given the context of multi-robot systems.

V. Experiment Design

A. Testbed

In our experiments we tested a number of formations and random graphs in a simulated environment. The environment is inspired by traditional networking and the p4 programmable switch architecture and programming language In this environment, each agent could request the identifiers of their neighbors and could only send packets to these neighbors.

Packets include a number of basic proporties: data, created clock timestamp, created local source agent update counter (used later to avoid race conditions), source agent identifier, target agent identifier, time in transit; time in transit is updated when a packet is relayed across a communicating link. Packet transmission was handled by the environment; an agent would send a packet to the environment, the environment would verify that their was an communication link (edge) between the two agents, and then add it to a sorted queue owned by the environment where the sorted value was the time of arrival of the packet according to the send time and the travel time of the communication link (edge weight). Packets were cloned to

represent the real-world independence of data. Before a packet was inserted into the environment queue, it would be wrapped with metadata that included the environment set time of arrival, the sender identifier, and the receiver identifier. The target agent identifier and receiver identifier are not always the same, as is the case with the source identifier and the sender identifier. The target identifier is where the packet terminates and the source identifier is where the original packet was created. In a route from agent a to agent d where the path is $\{a, b, c, d\}$, the source identifier is a and the target identifier is d; in this scenario the sequence of "(sender identifier, receiver identifier)" is $\{(a, b), (b, c), (c, d)\}$. To implement this algorithm outside of this environment, the receiver identifier is not needed (this is used by the environment to direct the packet's next hop) and the sender identifier can be replaced by port queues if that is available. While packet transmission is handled by the environment, the environment provides no routing and it is up to each agent to establish routing - in our algorithm we use a routing table.

A main loop instructs the environment to pass each packet, in order of time of arrival, to the receiving agent; the receiving agent has to handle the packet before the environment passes any more packets; as mentioned before, any packets sent during the handling will enter the environment's sorted queue according to time of arrival; the main loop terminated when there are no packets in the environment queue. The environment queue is designed to accurately enforce FIFO and transmission times; this is crucial for modeling race conditions that can arise during updates of topology changes.

The environment is also used to simulate dynamics. The environment is the only entity with access to the graph that represents the full communication network. Consistent with our assumptions, each agent polls the environment for the identifiers of its neighbors. To model changes in the network topology, we alter the adjacency matrix stored in the environment; we provide a number of helper methods to do this. Altering the adjacency matrix does not cause the environment to trigger anything, instead it is each agents responsibility to poll the environment for its latest set of neighbors and detect any changes. This reveals a crucial concept, only the agents at the ends of (added, removed, or updated) edge can detect the change. In real implementations, the agents may poll for changes at some regular time interval or receive a (potentially hardware or about-to-disconnect) notification.

The proposed algorithm ran on each node. Other than the unique identifier for each agent and the communication capabilities as defined by the network topology, the agents were completely homogeneous and the algorithm ran completely decentralized. This makes the algorithm scalable, less complex, and applicable to many robotic systems and ad hoc network settings including swarm systems.

B. Formations

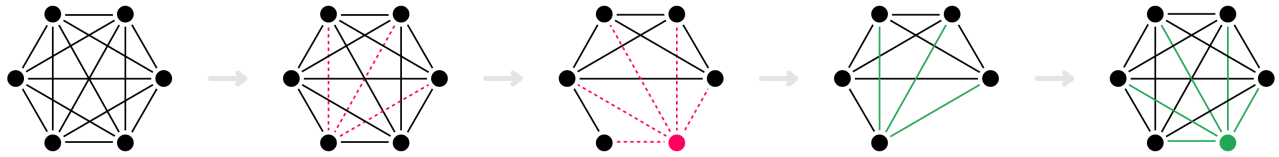
In our simulations, we tested our algorithm on a number of formations. We have 3 single-flock formations and 3 more multi-flock formations. All 6 formations have multi-flock timelines that simulate different states of the topology over time. We chose these formations because they represent common formations in practice, reflecting realistic scenarios, and they also test the different types of topology changes. To be thorough, we also test on a number of other unweighted graph and random weight graphs.

The first 3 single-flock formations (as shown in "Fig. 1") have two variations: full (original) and tree. The tree variation follows the same arrangement inspiration but is a spanning tree. These formations also all have a 5 step timeline (as shown in "Fig. 2") where the first step is the full graph, the second step removes some edges, the third step disconnects an agent from the original flock, the fourth step adds some edges back, and the fifth step merges the disconnected agent with the original flock.

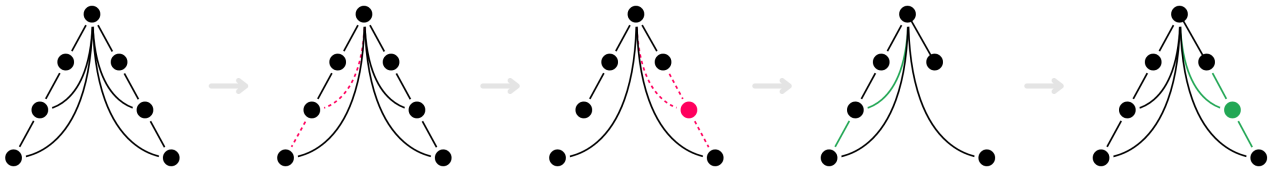
The 3 multi-flock formations are designed to validate key topological changes where a single edge can affect the whole flock. The "Rebalance-Update" timeline (as shown in "Fig. 3a") adds two new optimal leaders to the leader set, and evaluates the algorithms ability to propagate this update to the rest of the flock. The "Rebalance-Removal" timeline (as shown in "Fig. 3b") removes an edge that decides the previous set of leader and then has a new agent join the flock in an optimal position; this timeline evaluates the algorithms ability to propagate a route removal without disconnecting the flocks and then evaluates a lone agent joining a flock in the optimal position. The "Split-Merge" timeline (as shown in

(a) Circle Formation (b) Wedge Formation (c) Line Formation

Fig. 1: The full (original) and its corresponding tree variation for each of the single-flock formations.



(a) Circle Timeline



(b) Wedge Timeline

(c) Line Timeline

Fig. 2: The timeline for each of the single-flock formations.

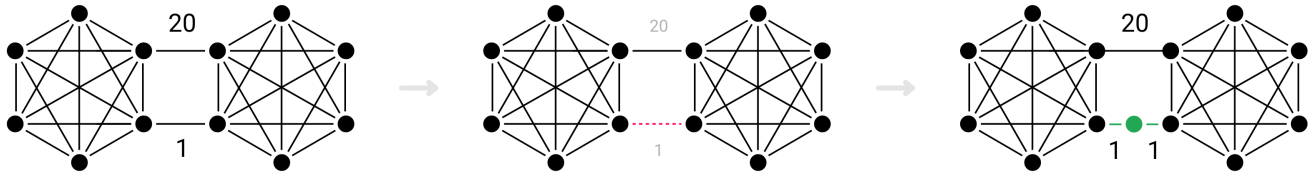
"Fig. 3c") disconnects the flock and then reconnects the two resulting flocks; this timeline tests one of the most important scenarios in multi-agent systems and the algorithms ability to help agents identify when other agents (not directly connected to the removed edge) are no longer accessible, and build routing when flocks merge.

C. Baseline and Validation

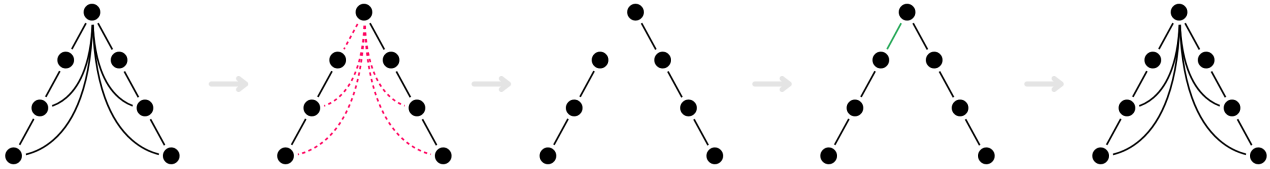
For our testing, we used a Floyd-Warshall algorithm to identify the true center of the graph. We also used this algorithm to log some useful characteristics like graph diameter. The single-flock formations are one connected component and could be tested with a simple Floyd-Warshall implementation. The multi-flock formations and all 6 timelines have, at one point in the configuration, multiple connected components; thus we used a depth-first-search to identify the connected components and then ran Floyd-Warshall on each connected component. We validated that our algorithm correctly identified all of leaders identified by Floyd-Warshall center algorithm and identified no extra agents.

We also graphed the network traffic according to different leader selections in the single-flock formations to for visual validation our approach that selecting the graph center as the leader would in fact result in the least traffic from leader to all followers.

(a) Rebalance-Update Timeline



(b) Rebalance-Removal Timeline



(c) Split-Merge Timeline

Fig. 3: The timeline for each of the multi-flock formations.

VI. Decentralized Optimized Leader Election Algorithm

A. Overview of Approach

Our goal is to present a decentralized algorithm for identifying all optimal leaders where a leader is optimal if they have the smallest communication time between the leader and all followers. It well established that that the graph center is the node who has the minimal greatest distance to all other vertices; the graph centers are the optimal leaders because by minimizing the greatest distance to all other vertices, we are minimizing the time it takes to propagate a command from the leader to all its followers. We begin by considering how we can identify the graph center is a decentralized manner. We choose to have each agent learn the distance of between themself and every other agent in the connected component; when they have done this, they broadcast the longest distance between themself and another agent to all other agents in the flock; this achieves the first part of identifying the graph center: calculating an agents greatest distance to all other vertices. By broadcasting this to all other connected agents, each agent can build a table of each other agents greatest distance route. Then the agents choose the optimal leaders (graph centers) to be the agent with the minimum greatest distance route.

To have each agent learn the between themself and every other agent in the connected component, we implement a classically-inspired broadcast based route learning method.

Over time the agents converge to locally discern the optimal leaders for their flock (connected component). The agents poll their environment for neighbor changes and when one is detected, they differentiate which edges have been added, removed, or had their weight changed. For simplicity, the algorithm treats weight changes as an edge removal and then addition. Then each agent responds according to the change for each edge in a way to update the routing tables for each agent and recalculate the leader.

B. Learning Shortest-Path Routing

1) Overview: Each agent must implement a routing mechanism to allow for future actions of the algorithm when the topology changes. It is also important that each agent have an accurate measure of

distance between themself and any other node to complete the first step of calculating the graph center. This can be done by having the agents learn shortest-path routing and then using the routing table's distances for the first step of calculating graph center.

2) Process: Each agent starts with an empty routing table and a local update counter. They all broadcast an empty packet to each of their neighbors. Each packet has a time in transit property that is updated each time a packet goes over a communication link. In a method called 'consider_routing_update', we consider any packet that an agent receives; if an agent receives a packet from a source identifier that is not in the routing table or has a time in transit less than what the routing table has stored for the source identifier, the agent updates their routing table to have a path to the source identifier by sending a packet to the sender identifier (the sender/relay had to be a neighbor) with the $key = source\ identifier$ and $value = (sender\ identifier, time\ in\ transit/length\ of route, source\ update\ counter)$; if the received packet causes the agent to update their routing table, then it forwards/rebroadcasts the original packet to all of its neighbors except the neighbor who sent the packet; otherwise, that packet ends there. This ensures that any slower routes are discarded and once everyone has their shortest route set, the packet stops being transmitted.

3) Analysis: The process is able to enable each agent to learn the shortest path to other agents because all paths are explored unless it would result in a further path than the agents route table and the route table is only updated if a shorter path presents itself. The algorithm stops transmitting packets after they do not provide a shorter path or a path to a node that did not otherwise have one. We know that if we consider the longest shortest-path in the graph as a line, which has length equal to $O(diam(G))$, going past the end of path and then returning will not provide a shorter path. Also a longer path that arrives at the same vertex won't trigger an update and thus won't get forwarded. As a result the time it takes for all agents learning paths to all agents they are connected to is proportional to the longest path which means the parallel runtime or traffic time of packets is $O(diam(G))$. Also, the proposed algorithm does not require acknowledgements or returned packets, instead since we assume an undirected graph, we use the symmetry of paths to ensure that both agents on either side of the path get updated. During this procedure, only a routing table is constructed and this table stores one entry per connected agents, thus the space complexity is $O(n)$. We leave analysis of number of sent packets to future works because we assume in this paper that packet processing takes arbitrarily small time.

C. Decentralized Leader Election (Identification)

1) Overview: Once each agent has a shortest-path routing table setup, they each have an accurate measure of distance between themself and thus know the length of the longest path between them and any neighbor. But each agent must know this information for others as well.

2) Process: Thus every time an agent updates their routing table, they send their greatest distance to all other vertices in their routing table, along with a timestamp, in a packet labeled as "lsp_update". While it would be more efficient to wait for the table to setting before sending this value, using a delay or waiting for a certain number of entries is unreliable and also makes it more difficult for updates during topological changes; thus we sending for every routing table update. Each agent has a local lsp (longest shortest-path) table where the key is an agents identifier and the value is the lsp of that agent and the timestamp as a tuple. Because this value can both increase or decrease when the topology changes, we include a timestamp; then whenever an agent receives a "lsp_update", it checks if the source identifier is not in the table or if the "lsp_update" has a newer timestamp then the table, if either are true, the agent adds the lsp and timestamp to their table for that source identifier. Every time an agent updates its lsp table, it recalculates the optimal leaders for its connected component; it does this by finding the identifiers in the table with the minimum lsp value (in the case of a tie, all are taken).

3) Analysis: In this procedure, we are able to accurately identify the graph centers because each agent communicates their maximum route distance with all other connected agents; thus each agent can select the same agent with the smallest maximum route distance because each agent in a connected component have the same and a complete set of information to consider.

In this procedure, each agent sends their update to all other vertices using their routing table; thus we know the distribution of each sent packet is a tree where the longest path is the graph diameter. We also don't have to worry about looping packets because the routing tables are being used, not broadcasting. Locally calculating the leader does not result in any packets being sent. As a result, again we get a runtime or traffic time of packets of $O(diam(G))$ per update. We know that when the routing table stabilizes there is only one update per agent in parallel and thus the two steps, which is the time it takes to converge and have every agent identify the same set of leaders, takes $O(diam(G) + diam(G)) = O(diam(G))$. Here the lsp table is like the routing table with one entry per connected agent, thus the space complexity is $O(n)$. While we don't provide a thorough analysis of the number of packets sent, this section sends fewer packets than the previous part as it sends directly to other agents rather than broadcasting and thus the number of packets in this section is $O(n)$ per update.

D. Responding to Removed Edges

1) Overview: Here the algorithm must work to inform agents of other agents that an edge has been removed from so that they can update their routing table. The process must ensure that agents find another path if one exists. Agents not directly connected to the removed edge must still update their routing tables, they must know which entry to remove and what to replace it with. Agents must also know if they are loosing routes to multiple agents, as is the case when one flock splits.

2) Initiated Process: Thus, the agent checks if it used that edge as a starting point (the first item in the routing tuple) for any routes; if it did, it compiles a list of the targets for those routes. Using that list, the agent sets the routing table entry to $(None, 0, entry_counter + 1)$ where the path start is None and the counter is the previous counter plus one. It also deletes the target agent from its lsp table. It then creates a packet labelled "edges_removed" with the list of targets with a route path that relied on that edge. It broadcasts this packet. The agent then increments their local update counter and broadcasts an empty packet like in **Learning Shortest-Path Routing**. It also shares an "lsp_update" and recalculates the optimal leaders like in **Decentralized Leader Election (Identification)**.

3) Receiving Process: When an agent receives a "edges_removed" packet it first double checks that it was not the sender. Then extracts the list of identifiers from the source and filters it to include only identifiers if this agent routes to that identifier through the sender agent. For each item in the filtered list, it removes the route from the table like above. For each, it also removes the entry from the lsp table if the "edges_removed" packet has a timestamp newer than the lsp entry timestamp. It then follows the above broadcast procedure: the agent then increments their local update counter and broadcasts an empty packet like in **Learning Shortest-Path Routing** and also shares an "lsp_update" and recalculates the optimal leaders like in **Decentralized Leader Election (Identification)**.

4) Analysis: Here we must ensure that if two edges are disconnected all the corresponding route entries are removed. This is achieved through the "edges_removed" packet which removes any routes that rely on a path to get to the other agent. It also ensures that if an agent had a different route to the agent in question, then that route is left alone. Leader election is updated because sharing and calculation is re-executed. We must also ensure that if removing an edge removes entries in the table, but another path exists, the new routes are entered into the table. This is achieved by rebroadcasting with a newer update counter.

E. Responding to Added Edges

1) Overview: When an edge is added the algorithm must be able to recalculate routes and the graph center if affected and must also be able to establish routes with agents further away from the edge; a more specific example is when two flocks merge, here agents not connected directly to the added edge in both flocks must still learn routes to all the agents in the other flock.

2) Initiated Process: An agent that detects a new neighbor and thus a new edge, checks if that new neighbor was not in their routing table (merging) or if that edge brings them closer to the neighbor; if either are true, it updates the routing table using this new edge. The agent then increments their local update counter and sends a packet called "edges_added" along the added edge; the packet contains the agent's routing table. When sending their routing table, they add entry for themself with route length 0. The agent then shares its lsp and recalculates the optimal leaders like in **Decentralized Leader Election (Identification)**.

3) Receiving Process: When an agent receives a "edges_added" packet it first double checks that it was not the sender. Then it considers each entry in the received table. If the entry is to the current agent, it skips it. If the entry is not set or the entry starts with a path that is not set, it skips it. It then calculates the adjusted route by taking the original route path length in the table and adding the packet's time in transit, since using that route would involve taking all the steps the packet did. If the target of the entry is not in this agents routing table or the adjusted route length is less than the current agents route table path length or has a newer timestamp, the current agent updates its route table entry. If any entries were updates, the original packet is rebroadcasted to all neighbors except the sender (not sent to entries) and the agent then shares its lsp and recalculates the optimal leaders like in **Decentralized Leader Election (Identification)**.

4) Analysis: Here we must ensure that when two nodes are connected, they can route to all other nodes on the other side. We accomplish this by forwarding the routing tables. We know by symmetry that all new paths will be considered and thus if there is a shorter path it will be excepted. Leader election is updated because sharing and calculation is re-executed.

F. Responding to Updated Edge Weights

1) Overview: Increasing or decreasing of edge weights can result in different graph center and thus a different set of optimal leaders. We want to find the updated shortest paths and choose the correct leaders.

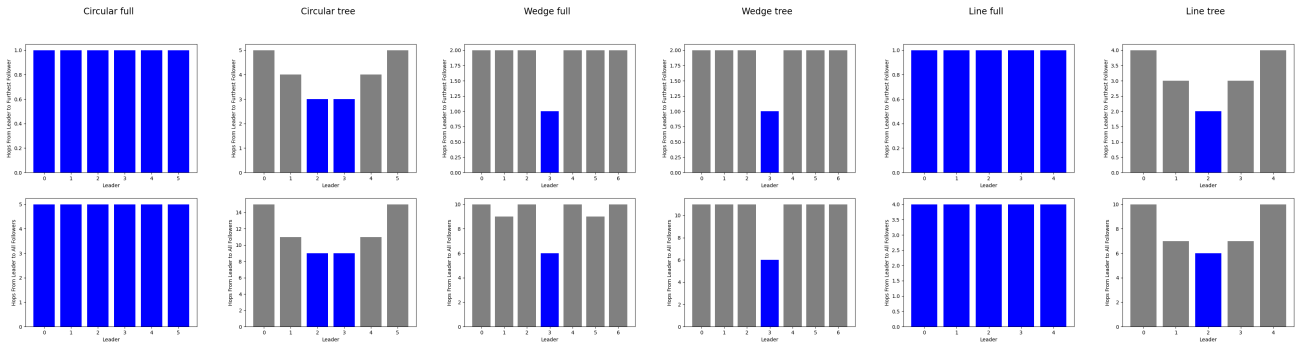
2) Initiated Process: When an agent polls for its neighbors, we assume that it also has the communication time of the link or the weight of the edge (can be obtained by sending a packet directly across and back). Every time the agent receives its neighbors and weights, it saves one layer back. This is how it detects new or missing neighbors and how it detects altered weights. When an edge is altered it is treated as a removed edge process like in **Responding to Removed Edges** and then an added edge process with the new weight like in **Responding to Added Edges**. This is actually done by returning the altered edges in the list of removed edges and in the list of added edges returned by the method that compares neighbors.

3) Analysis: We ensure that this results in accurate recalculation of routes and leaders because this can be through of a removing the edge that agents have in their table and potentially replacing it with a parallel edge or using a different shorter path. By marking it as a removed edge, all agents remove the old route and route length and then by marking it as an added edge, all agents can relearn the new route with the new weight.

VII. Experiments

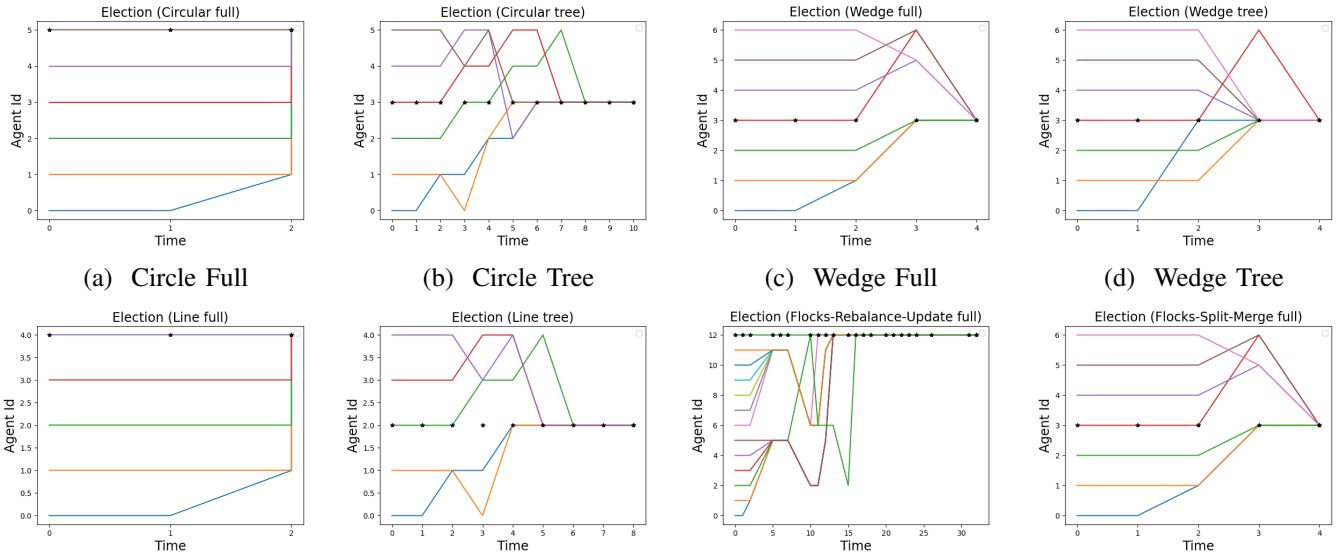
A. Basline

Before we began, we wanted to visually show that choosing the graph center does in fact result in electing the leader that can propagate commands to all followers in the least amount of time and traffic. In "Fig. 4", we tested the three single-flock formations, determining the graph center (using the Floyd Warshall Algorithm) and measuring hops to furthest follower (top) and total hops to all followers (bottom) for each selected leader. The graph centers are the blue bars. It is clear to see that choosing the graph center as the leader results in both less network traffic and faster propagation to all followers. Time to propagate to followers is measures as the longest path from leader to follower. The network traffic is measured as the sum of path lengths from leader to all followers.



(a) Circle Full (b) Circle Tree (c) Wedge Full (d) Wedge Tree (e) Line Full (f) Line Tree

Fig. 4: Hops to furthest follower (top) and total hops to all followers (bottom) for each selected leader; different bar is different leader. Blue bars are the graph centers. For formations: circular, wedge, line.)



(a) Circle Full (b) Circle Tree (c) Wedge Full (d) Wedge Tree

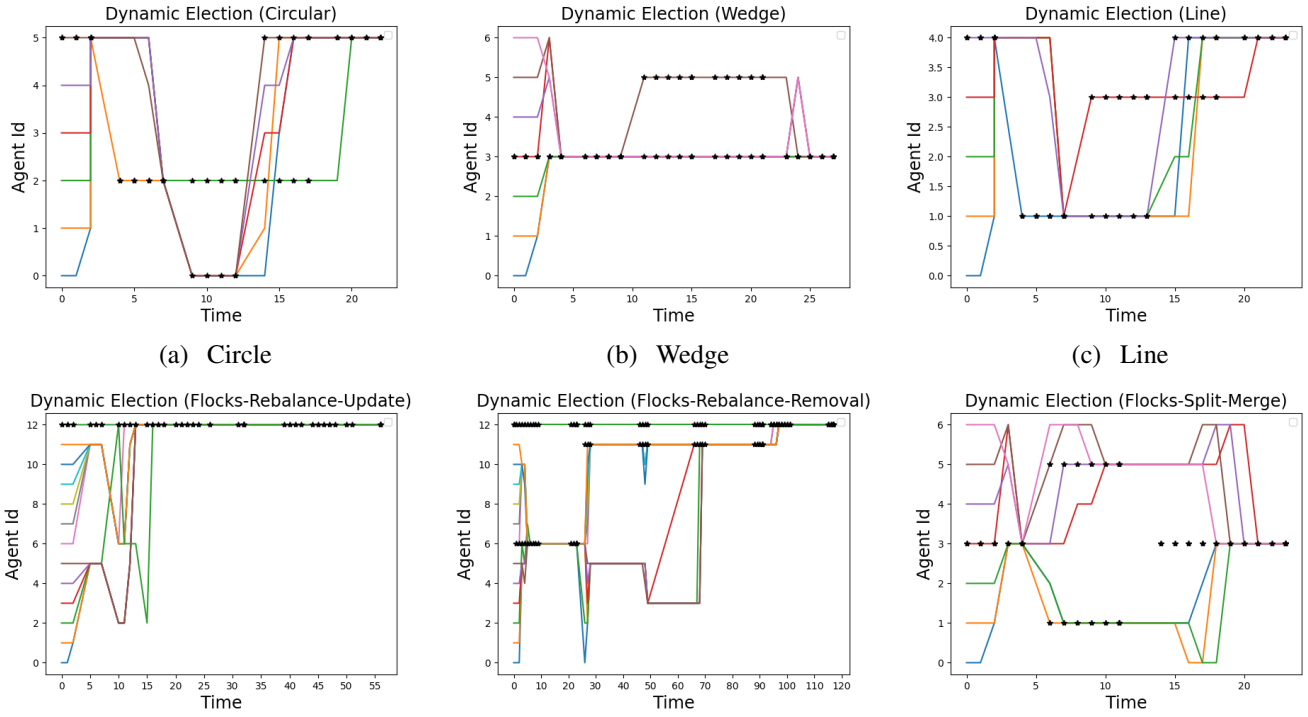
(e) Line Full (f) Line Tree (g) Rebalance Update (h) Split Merge

Fig. 5: Each agent as a line as they converge to select the (max) optimal leader; stars represent true optimal. We see that each agent converges to true optimal by the end.

B. Static

After establishing the Floyd Warshall algorithm as our baseline and validator, we were ready to go ahead and implement our proposed algorithm. During implementation, we tested our algorithm against the single-flock formations, a number of arbitrarily connected graphs, and a random weighted graphs. During this testing, we compared the set of centers returned by the Floyd Warshall algorithm to the converged set of leaders of each agent after running our proposed algorithm. The algorithm passed every test.

We then plotted this to show convergence over the network and provide a visual aid for our results. In this chart, each line is a agent; the x-axis is time; the y-axis is id of the selected leader; the stars represent the optimal leader. Because there can be multiple optimal leaders we chose the agent with the maximum id as the tie breaker. While this is not the most ideal visualization for convergence, it shown the agents as they locally calculate the optimal leader and then eventually converge to the true leader. As you can see in “Fig. 5”, in all simulations, every agent converges to the optimal leader which is represented by their line ending on the value with the stars.



(a) Circle (b) Wedge (c) Line

(d) Rebalance Update (e) Rebalance Removal (f) Split Merge

Fig. 6: Each agent as a line as they converge to select the optimal leader and topological changes according to timelines in "Fig. 2"; stars represent true optimal. We see that each agent converges to true optimal by the end of each topological phase.

A couple things are highlighted about the behavior of our algorithm through these charts, each lesson of which can be discovered also by investigating the procedures explained above. First, each node starts by establishing themself as the optimal leader; this is because at the beginning of the algorithm, they don't have routes to or know of any other nodes. Second, the time to converge and the time to process all packets is proportional to the diameter of the graph; this can be seen as Rebalance-Update takes signficantly longer than any other formation and Circle full and Line Full takes the least amount of time as both are fully connected and have a diameter of 1.

C. Dynamic

After verifying that the proposed algorithm could correctly converge to and determine the full set of optimal leaders, we proceeded to testing the dynamic case where the system experience topological changes. Here we tested each formation according to the timelines in "Fig. 2". In this experiment, the systems could partition into multiple flocks (connected components) or start as multiple connected components (as is the case with Rebalance Remove). To ensure absolute correctness, since the other ones were just checkpoints, we modified Floyd-Warshall to run on each connected component. Then we compared the results of the proposed algorithm to the results of Floyd-Warshall for each connected component. In every case, our algorithm performed correctly, recognizing the full set of optimal leaders for the agent's respective connected component.

We also plotted this; plotting with a similar fashion to the **Experiments: Static**. Again the stars represent the true optimal; but this time when there are multiple connected components, there are multiple horizontal lines of stars to represent the (max) optimal leader for each connected component. The system ran the algorithm and then after some time a topological change was introduced according to "Fig. 2".

Analyzing the charts in "Fig. 6", it is visible where the topology changes and the agents work to converge to the new optimal leader. For each segment that represents a topological state, we see that all agents converge to the optimal leader for their component. Split Merge is one of the most realistic and universal mobile multi agent flocking topology change patterns and looking at the chart, we can see that when the flock splits into two, the agents learn their new leader; then when the flocks merge, the agents converge again to the correct leader.

VIII. Discussion

We have proposed a simple, resilient, and completely decentralized algorithm that provides a mobile dynamic system with both shortest-path routing and optimal leader election. The algorithm is light-weight in space complexity, depends only on very basic networking concepts, and does not require integration with any other system algorithms. The algorithm is widely applicable due to its resilience to topological changes and ability to handle weighted communication links/edges. We have also provided guidance on reducing the few assumptions we made.

We showed the algorithm on a number of realistic formations and sequences. The Line and Wedge formations are representative of many survey-focused robotic formations. The Circle and Circle Tree are representative of many travel-oriented formations as well as most unmanned aerial vehicle formations. The first three timelines show the four types of basic topology changes: removing communication links, disconnecting agents, adding communication links, adding agents. These timelines can reflect a scenario where an agent in a flock separates to perform a task and then rejoins the flock. Rebalance Update shows the algorithms ability to recalibrate as communication link weights/values change over time, for example when the environments change or robots are able to communicate faster, potentially due to moving closer together. This timeline (Rebalance Update) specifically shows the proposed algorithms ability to persist and perform while other algorithms or the environment itself change the topology, like when an a motion planning or formation control algorithm is navigating an obstacle, rearranging the formation, or optimizing the formation. Rebalance Removal reflects the real world scenario where object interfere and remove a crucial communication link. This timeline's last step also shows how a system supervisor can add an agent to flock to optimize communication and the proposed algorithm will handle the integration and election. Finally, the Split Merge timeline shows the most common use case where a flock has to separate to fulfill its responsibilities or avoid obstacles. It also represents when two flocks come back together. In addition to providing optimal leader election, the proposed algorithm ensures that the flocks have a complete and shortest-path routing mechanism in all of these scenarios. This can also be easily extended to support shortest-multi-path routing.

IX. Conclusion

In this paper we investigated the problem of identifying optimal leaders in a decentralized manner in mobile multi-agent robotic system or ad hoc network. We presented a capable and light-weight algorithm inspired by core networking concepts that is able to identify optimal leaders in $O(diam(G))$ time and $O(n)$ space. The algorithm also provides shortest-path routing as an additional benefit.

Our future work will include providing more complete runtime analysis of computation on each agent and number of packets sent. We will also explore efficiencies that can be made when multiple topology changes occur concurrently or in quick sequence.

Future works may include investigating multi-path routing or more efficient updates by increasing the space complexity on each agent. More simply extensions could pursue the directed case of this problem, the case where only a subset of agents can be leaders, or integrating this system with a motion planning or formation control algorithm. We also encourage future work in a k-leader or k-centers extension of this work as well as the weighted or valued agent case where agents have a value or cost.

X. Acknowledgements

The authors would like Harvard Professor Michael Mitzenmacher and Harvard Professor Stephanie Gil for the guidance and support.

References

[1] S. L. Hakimi, "Optimum locations of switching centers and the absolute centers and medians of a graph," *Operations Research*, vol. 12, no. 3, pp. 450–459, 1964. [Online]. Available: https://doi.org/10.1287/opre.12.3.450

[2] Jie Gao and Li Zhang, "Load-balanced short-path routing in wireless networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 4, pp. 377–388, 2006.

[3] K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, and R. B. Tan, "Fundamental control algorithms in mobile networks," in *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 251–260. [Online]. Available: https://doi.org/10.1145/305619.305649

[4] N. Malpani, J. L. Welch, and N. Vaidya, "Leader election algorithms for mobile ad hoc networks," in *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ser. DIALM '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 96–103. [Online]. Available: https://doi.org/10.1145/345848.345871

[5] E. Sultanik and W. Regli, "Stable service placement on dynamic peer-to-peer networks: A heuristic for the distributed k-center problem," *Proceedings of the National Conference on Artificial Intelligence*, vol. 1, pp. 196–201, 01 2005.

[6] A. Clark, B. Alomair, L. Bushnell, and R. Poovendran, "Leader selection in multi-agent systems for smooth convergence via fast mixing," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, 2012, pp. 818–824.