Supervised Classification of Motion Graphs for Swarm Robotics

by

Ryan Lear

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. Anshumali Shrivastava, Chair
Assistant Professor of Computer Science

Dr. Chris M. Jointaine
Associate Professor of Computer Science

Dr. Luay K. Nakhleh
Associate Professor of Computer Science

Houston, Texas
May, 2016
ABSTRACT

Supervised Classification of Motion Graphs for Swarm Robotics

by

Ryan Lear

Tasks for robot swarms, such as measuring performance, diagnosing robots, and allocating actions, can all be improved by empirically understanding how the swarm is behaving. For many of these tasks, direct observation is typically used to understand whether a group of robots is correctly completing some task. However, this is especially challenging for swarms with a large number of robots spread around a wide geographical area. To bridge this gap, we focus on classifying behaviors. Although methods exist for classifying groups of robots assuming global localization, this is the first study to automatically classify one group of robots with another using fully distributed sensing.

This work develops a method for classifying motion of groups of robots using a novel graph-based data structure, which we call a motion graph, that embeds continuous motion into a discrete graph. We explore the use of k-Graphlets to derive features from path lengths in graphs. Using k-Graphlets classifies several simulated behaviors with 85% accuracy, but runs relatively slowly. To improve this, we develop an algorithm using path lengths, which proves to be just as accurate as using k-Graphlets but orders of magnitude faster, making it well within practical use for problems like diagnostics, automatic deployment of robots, and task allocation.
Acknowledgments

This would not have happened without the help of my advisors. I’m grateful to my first advisor, James McLurkin, for helping me understand what it takes to discover and solve a problem. While the process might have been non-standard, it certainly kept things exciting. I’m just as grateful to my second advisor, Anshumali Shrivastava, for helping me find a path to my goal, no matter how challenging it seemed.

I would also like to thank my committee members, Luay Nakhleh and Chris Jermaine, for supporting me and looking out for my success.

I thank Eugene Ng for helping me navigate the surprising variety of issues I ran into. Without his support, I would not have come nearly as far.

I am grateful of all the help Sarah offered, both on the thesis itself and in making sure this work always felt possible. I thank Shannon for constantly helping me work out what I actually want to say. I also thank Didier for showing me the things I thought I was saying, but never actually was.

Of course, I thank all of my friends and colleagues who have helped me think about this project in different ways, regardless of whether then know it or not.

And lastly, I thank my family for always being there for me.
Contents

Abstract ii
List of Illustrations vi
List of Tables xi

1 Introduction 1
  1.1 Motivation .................................................. 2
  1.2 Contributions ............................................... 3

2 Background 6
  2.1 Swarm Robotics ............................................. 6
  2.2 Graphs ....................................................... 11
  2.3 Classification .............................................. 12
  2.4 Graph Kernels ............................................. 17

3 Motion Graphs 19
  3.1 Generating Graphs .......................................... 19

4 Classification Methods 24
  4.1 Preliminary Tests ........................................... 24
  4.2 k-Graphlets ............................................... 26
  4.3 Path Lengths .............................................. 29

5 Simulations 34
  5.1 Assumptions ................................................. 34
5.2 Behaviors .................................................. 35
5.3 Validation Techniques .................................... 37

6 Results ....................................................... 40
   6.1 Overall Results ...................................... 40
   6.2 4-Graphlets .......................................... 43
   6.3 Path Lengths ......................................... 51
   6.4 Kernel Tests .......................................... 57

7 Conclusions and Future Work ............................ 59

Bibliography .................................................. 62
Illustrations

2.1 An example of a decision tree classifying three possible weather states. In this example, if air pressure is high, the classifier will always say it is sunny. If not, then in low temperatures it will snow, while in high temperatures it will rain.

2.2 Left: Example of separating clusters of examples with the dotted line. The solid lines show the margin between the two clusters. Right: Clusters that are not linearly separable. The dotted line still separates all examples except for the blue outlier.

3.1 Generating edges for motion graphs. The robot travels along the path and enters the range of A at time $t_i$. At time $t_j$, the robot has left the range of A but is still sighted by B, so an edge is formed between A and B with associated time $t_j$.

3.2 Examples of motion graphs generated from individual robots. Each of these graphs was generated using five seconds of simulated data. Even with this small amount of information, some behaviors are visually easy to tell apart from each other.

3.3 Motion graphs for the entire set of robots. Behaviors like resource collection and static motion are clearly visually different from other behaviors.
3.4 Generating multiple edges at once. When the robot reaches the point on the path shown by the red dot, it leaves the range of $C$, but remains in range of $A$ and $B$. Because of this, edges $(C, A)$ and $(C, B)$ are created.

4.1 The decision tree generated for our proof of concept. Ellipses represent decisions while rectangles show classified behaviors.

4.2 An example of a feature vector generated using 3-graphlets. The graph above contains four vertices and three edges. Each subset of three vertices is isomorphic to one of graphs on the bottom. Three subsets of vertices have only one edge, while one subset of vertices contains three edges. Looking at these as a histogram of counts returns the feature vector $(0, 3, 0, 1)$.

4.3 Sets of degrees of vertices for each possible graph containing three vertices. Each possible set is unique, so finding isomorphic graphs from a subgraph is straightforward.

4.4 Sets of degrees of vertices for each possible graph containing four vertices. Each subgraph also maps to a unique set. (Adapted from Shervashidze (2009) [1])

4.5 An example of an undirected multigraph. There are six possible ways to choose edges from this graph while still maintaining isomorphism to the underlying simple graph.

4.6 Two graphs with different walks. The upper graph contains a cycle generated by the first three edges. In the algorithm, a path of length 3 would be recorded in the feature vector, and the path would be reset. The lower graph would not record anything, since the walk contains no cycles. This separates short local motions from longer global motions.
4.7 Algorithm for generating path length features .................................. 31

4.8 On the left, a sample path for a robot. On the right, The *motion graph* for this path. Note that there are two cycles representing a single motion. Following the algorithm, both cycles will be recorded as paths of length two, resulting in a possible overcount. ................ 32

5.1 A snapshot of the simulation environment. Blue dots represent observers while red robots represent agents. The behavior in this example is resource collection, with home base in the top left corner, and resources in the other three corners. .................. 35

5.2 A possible connectivity graph for the observers. Note that a message can be passed from a robot to any other robot using a multi-hop routing protocol. ..................... 36

6.1 Timing and Accuracy results for 4-Graphlets (top) and path lengths (bottom). From left to right are tables for accuracy, amount of time spent generating features, and training time. Results for 4-Graphlets are shown with respect to the number of samples, while results for path lengths are shown with respect to the maximum path length. . . 42

6.2 Confusion matrix for 4-Graphlet feature vectors. Training and testing data were taken from all subsets of data. Overall, 4-Graphlets classified behaviors with 85% accuracy, but showed some difficulty with random motion. Labels in this figure are the same as Figures 6.3 and 6.4. .......................... 44
6.3 Confusion matrices for 4-Graphlets using a classifier generated from the entire data set. Labels for these matrices are the same as in Figure 6.2. The subsets of data tested for each matrix were, (a) the entire data set, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the amount of time the simulations ran, (f) varying the range of vision for agents or observers, (g) varying vision for only observers, (h) varying vision for only agents.

6.4 Confusion matrices for 4-Graphlets using a classifier generated from each subset of varied parameters. The subsets of data tested for each matrix were, (a) varying the amount of time the simulations ran, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the range of vision for agents or observers, (f) varying vision for only observers, (g) varying vision for only agents.

6.5 An example stack graph to show how predictions vary with respect to the number of agents. Data points are separated evenly to show when classifications for each behavior succeed or fail.

6.6 Comparison of predictions using 4-Graphlets for each variable (Number of observers or agents, range of vision of observers or agents, and amount of time). Each chart shows classifications of each of the five behaviors. For example, increasing the range of vision of the agents causes dispersal to appear like random motion. This may be because agents jitter a lot while dispersing with a high range of vision, which also occurs in random motion.
6.7 Confusion matrix for path length feature vectors. Training and testing data were taken from all subsets of data. Path lengths were classified with 85% accuracy overall, but showed some confusion between dispersal and random motion. Labels in this figure are the same as Figures 6.8 and 6.9.

6.8 Confusion matrices for path lengths using a classifier generated from the entire data set. The subsets of data tested for each matrix were, (a) the entire data set, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the amount of time the simulations ran, (f) varying the range of vision for agents or observers, (g) varying vision for only observers, (h) varying vision for only agents.

6.9 Confusion matrices for path lengths using a classifier generated from each subset of varied parameters. The subsets of data tested for each matrix were, (a) varying the amount of time the simulations ran, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the range of vision for agents or observers, (f) varying vision for only observers, (g) varying vision for only agents.

6.10 Comparison of predictions using path lengths for each variable (Number of observers or agents, range of vision of observers or agents, and amount of time). Each chart shows classifications of each of the five behaviors. For example, decreasing the range of vision of the agents causes dispersal to appear static.
### Tables

4.1 Confusion matrix for early tests. The only mistakes were there examples which were classified as coverage instead of random motion. 25

6.1 Precision and recall for all subsets of data for 4-Graphlets using 10000 samples and path lengths using max lengths of 10. No values are included for *all data* sub-groups since this is already equivalent to training on all data. 43

6.2 Time and Accuracy for preliminary kernel tests sorted by mean accuracy. Counting all 3-Graphlets was the fastest and most accurate. 58
Chapter 1

Introduction

While an individual complex robot may have a wide range of mobility and sensing capabilities, that robot will still be limited in how much it can sense and do at any given time. In contrast, individual simple robots may not be able to sense their environment as richly, but a large number of simple robots can sense many parts of an environment at once through distributed perception.

Multi-robot systems with large populations are ideal for completing tasks that require coverage over a large geographic area or can be broken into many parallel operations. These robots can accomplish a wide variety of tasks including, but not limited to, exploration [2], patrolling [3], and foraging [4]. Additionally, using a large number of robots increases robustness to failure, reduces problem complexity, increases parallelism, and simplifies hardware design [5].

There are many understudied issues in swarm robotics such as measuring correctness, finding malfunctioning robots, deploying robots into an already running group, and competing with robots accomplishing some unknown task. A common thread in all of these applications is the need to understand what some group of robots is doing, while not being limited by the size of the swarm. Especially if robots are deployed in a large geographic area, having methods for automatically understanding what they are doing is key to making swarm robotics practically useful.

In particular, observing and responding to behaviors with a group of robots is a challenging problem. If a robot in a swarm is malfunctioning in some way that
obstructs other robots, it is reasonable to identify and remove that robot from the environment. Similarly, if two groups of robots are competing to accomplish some task, one group of robots must understand what the other is doing before reacting with some plan. If we assume we can recognize what task the group is trying to accomplish, then this reduces to a task allocation problem [6]. Just as tasks need to be allocated to people when solving some problem, robots need to find a way to break up tasks in a swarm. This assumes that the distribution of tasks can be easily determined, which may not always be the case, so it is important to find a method for automatically understanding what a swarm is doing.

Determining swarm behaviors can also be used for diagnostics. Currently, evaluating whether or not a swarm of robots is properly completing some task is accomplished by simply observing the robots. Proofs and quantitative data can be used to argue that the robots are completing their task, but there is no way to automatically determine whether or not this is the case. Even worse, there are no agreed upon metrics to compare different methods solving the same problem [7]. If there are bugs in the software or damaged hardware, fault detection can be used to check when hardware fails [8]. However, this still requires learning how to check for faults for each hardware configuration, which may not generalize well to different robots or show algorithmic bugs.

1.1 Motivation

Distributed algorithms frequently utilize local information to achieve some global result. As an example, drivers use only what they can see around them when navigating to some location. No individual sees their entire environment, but drivers can still get from one point to another without too much trouble. Even so, drivers still benefit
from using traffic information to make decisions about which route to take. This traffic information is not gathered by looking at individuals, but by analyzing traffic as a whole. In the same way, a robot can act in its local environment to create global behaviors in a swarm.

Because of this, we look at classification to automatically label behaviors in a distributed fashion. Walcott (2006) has done work in classifying swarms, but only while assuming the position of each robot is known [9]. Because Walcott uses global information, they require either a central processor or a high network overhead to share information. As such, it is difficult to argue that this approach can scale to large environments. This still leaves the issue of finding a way to understand motion of robots using distributed sensing. To do this, we need to utilize a data structure that properly abstracts motion in a way that is both easy to compute and fully represents the motion of swarms.

While the classification problem is not new, to the best of our knowledge, there has been no previous work on classifying robots with another group of robots in a distributed fashion. With this said, there has been some focus classifying multiple targets [10] and detecting outliers [11] in sensor networks. Eyal (2010) develops a method most similar to the goal of our work in terms of fully distributed classification by focusing on the mechanics of fully distributed classification using sensor networks [12]. Even with this method, we are still left with the problem of generating and classifying features particular to motion of swarms of robots.

1.2 Contributions

The objective of our work is to classify robot swarm behaviors in a distributed fashion. In our thesis, we show that this can be done in practice, using supervised learning
as a proof of concept. Because the robots are assumed to be simple, we attempt to
solve the problem with minimal sensing capabilities and very limited localization. We
also anticipate the need to mitigate many common noise and communication issues
in swarm robotics in the design of our algorithms.

To accomplish this, we implemented a data structure for embedding and discretiz-
ing motion, which we call a motion graph. This data structure is designed to abstract
the motion of the group of robots as a whole while addressing many engineering
concerns of multi-robot systems including limited memory, small message sizes, and
noisy sensors and communication.

We also propose a general methodology to derive classification features from mo-
tion graphs, using various graph descriptors. The first descriptors we utilize are a
modification of k-graphlets [1]. Then, we introduce novel descriptors that we call
path lengths. Both kinds of descriptors are used to generate features in the form of
histograms of counts, or more precisely vectors of counts.

To test this, we used supervised learning on five swarm behaviors and evaluated
the results with 10-fold cross validation. We also measured robustness to changing
variables such as the number of robots, the range of vision of each robot, and the
amount of time gathering data.

Our experiments show that both k-Graphlets and path lengths can lead to classi-
fication with around 85% accuracy. However, using path lengths proves to be at least
1200 times faster and lends itself well to a fully distributed system. These methods
also outperform commonly used kernels by using features only available using mo-
tion graphs. Fundamentally, this shows that classifying robot swarms is possible and
potentially usable in real systems.

A possible application for classifying behaviors is diagnosing and debugging swarms.
If a set of canonical behaviors is known beforehand, classification can be used to verify the correctness of a new implementation. Another use would be automatically deploying some group of robots into an already existing swarm without needing to specifically program each robot. The deployed robots would only need to observe the behaviors of the swarm. This is closely related to task allocation, where we may want a group of robots to respond to another swarm in some way. Classification paves the way for solving these kinds of problems.
Chapter 2

Background

2.1 Swarm Robotics

In many robotics applications, large sophisticated robots are used to accomplish complex, repetitive, or physically difficult tasks. However, large robots are either prohibitively expensive or too specialized to solve other problems. Taking inspiration from swarms of animals like insects and birds, recent research has looked into using many simpler but significantly cheaper robots to achieve a variety of tasks.

A motivating example in nature is a swarm of ants foraging for food, in that one ant will find food eventually, but many ants can cover a wide area in parallel [14]. Using this as inspiration for robots, searching disaster areas for survivors can be accomplished with a few complex remote controlled robots, but many simple, durable robots can explore a wide area much more efficiently [13]. Once a survivor is located, the robots might still need to notify a human operator, so relaying information without relying on an Internet connection is key to making the use of swarms practical in this situation and a variety of other applications. Often, communication and sensing is an issue, so developing algorithms to manage these limitations is a frequent subject of interest. Using GPS or some other method of global positioning is not feasible or reliable in many cases, but clever algorithms can compensate for this.

If someone wanted to solve a maze without knowing the layout beforehand, they might have a robot follow the walls until it reached the end [2]. However, in cases
where a single robot would need to make a choice, a sufficiently large group of robots could explore each possible path and leave one robot behind to mark the junction. Once a robot found the exit, it could message the other robots to follow it out so that the remaining robots are not stuck in the maze. Using a similar search method can help find the shortest path through the maze [15]. Again, if ants need to create a path from food to their nest, finding an efficient path is important even in complicated environments. Accomplishing this class of problems without global knowledge is challenging, so there is often a trade off between theoretical guarantees and practical solutions. Regardless, some research has been done on creating physical data structures, such as k-redundant trees [16], to meet practical requirements. Since most experiments are run in controlled settings, leaving a few robots behind has not been a significant issue. This points to the possible existence of many other issues that simply have not been examined because researchers have not needed to solve these practical concerns.

Researchers typically focus on problems that emulate behaviors seen in nature. In particular, research in swarm robot behaviors often involves understanding how to generate collective motion or organize many robots into a formation. This is important both on a small scale to understand how collective behaviors form with only local information, as well as on a large scale to use these small behaviors in real environments. We highlight some examples here, but refer to Brambilla (2013) for a more thorough taxonomy of behaviors [7].

Flocking behaviors have been frequently studied since groups of animals moving together as a cohesive unit are often observed in nature. One of the main obstacles in achieving this with robots is that animals can quickly understand relative positions of nearby neighbors, but sensors on robots are often too simple to get sufficiently rich
information [17]. Even when looking at separating groups of different types of agents, global knowledge of the system is often required [18]. Despite this, some problems like coverage control still have fully distributed adaptive solutions that can be verified for correctness [19]. This allows full use of a mobile sensing network without relying on a central processor. On a smaller scale, a high level of coordination is needed to transport some object using a handful of robots. However, taking advantage of geometry and abstracting motion to simply move around the centroid of the object removes a lot of this complexity [20].

While there is a lot of focus on moving groups of robots in general, there is also interest in using robots to achieve particular goals. Pursuit-evasion is a well studied area, where a group of robots is trying to catch another group of robots in an environment. Depending on the constraints, the problem could be based on visibility [21]. Assuming global knowledge of the environment, this would make sense for robots with cameras, and could be used for patrolling some building. On the other hand, many robots have limited range sensors, so a frontier based algorithm should be used in this situation [22]. In both of these cases, the problem is more about solving the environment than the tactics of actually catching a robot. A group of robots moving directly towards a target may not be as successful as multiple robots strategically cooperating. Coordinating motions with AI intuitively makes pursuit more effective [23]. From the opposite perspective, flocks can be designed to avoid predators through cooperative learning [24]. Without global knowledge, an environment typically needs to be mapped out. Instead, with enough robots, a local environment can be embedded using triangulation that can be used to solve problems such as patrolling [3, 25].

Taking this to an extreme, some algorithms focus exclusively on abstractions of
the environment. For example, if an environment is already known to be constrained, it can be modeled as a graph in the space. Using only limited communication, a group of robots can plan a path from start to end states using only a small set of operations [26]. In industry, a company might have a distribution center with a known floor plan and need some way to plan the motions of a fleet of robots to transport stock. In this case, the problem is still limited enough to solve the high level path finding problem with well known algorithms such as A* while focusing on defining low level interactions for avoiding robot collisions [27].

Many of these problems rely on understanding where each robot is relative to some reference frame. Systems may achieve this by using global positioning or relative localization. A fundamental problem, not only in multi-robot systems, but in robotics in general, is simultaneous localization and mapping, or SLAM [28]. Moving around without constantly colliding with objects is something that many people take for granted. Simply perceiving and maneuvering objects in test settings is fairly difficult for robots, but adding in real world constraints of navigating around and interacting with people or manipulating complex or compliant objects makes this much more challenging [29]. Even making use of limited sensor information relies on sophisticated techniques to understand the environment. These methods range from working directly with computer vision [30] to using Kalman filters on less feature rich sampled data [31].

Even assuming robots can sense their environment well enough, ensuring communications between robots is still an issue. Similar to networks in general, this problem requires engineering protocols which ensure that messages that are sent are handled correctly, and more importantly, that the system is robust to failure. Many of the issues present in networks are amplified, especially in multihop routing, where
a message needs to be sent from one robot to another outside of its communication range [32]. Besides typical networking protocols, there are now problems of dynamic network topologies and potentially unreliable connections. While many of the issues can be mitigated, it is still necessary to build algorithms with the understanding that communication may not be perfect, especially with a swarm in a large geographic area.

Sensor networks, although not directly related to swarm robotics, run into many of the same problems with limited sensors and power consumption. Some research focuses specifically on reducing the energy and computation requirements of algorithms, [33] while other studies will examine specific sensors such as audio sensors [34]. Because of the growing number of sensors in the world, more research is focusing on dealing with both the limitations and benefits of the sheer number of sensors.

Ideally in swarm robotics, solutions to many problems should scale well into hundreds, thousands, or millions of robots. Some experiments show how successfully human operators can control a system while varying properties like the number of robots, control methods, and how the swarm is visualized [35]. These methods hint at novel control schemes such as using single global inputs for entire swarms. While these are still simulations, it highlights the necessity to think about what using thousands of robots means both in effectiveness and in the user experience. Other experiments have examined ways to express and analyze the dynamics [36] or distributions [37] of multi-robot systems. Currently, there are few to no common benchmarks, so comparing algorithms is nearly impossible [7]. Solving this issue moves beyond simply developing algorithms for behaviors towards providing a common framework for validation and verification. Regardless, many solutions still attempt to address problems in real systems.
Because costs are still prohibitive, the largest real experiments have only been run on 1024 robots using Kilobots [38, 39]. Along with engineering hardware, they also consider practical issues, including finding ways to power on every robot, since simply turning on individual robots by hand would take too much time, and does not scale well for even larger swarms. Despite the long term goal of using large swarms, many problems still use hundreds of robots on actual hardware as a testbed for understanding both the issues in algorithms that only come up in real systems as well as issues fundamental to hardware itself.

Especially at this scale, managing the structure of the system requires a useful model. At the low level, solutions in robotics must manage sensing, communication, and other engineering issues, but at a more abstract level, it is useful to only work in algorithms. To facilitate this, we often model robot swarms with graphs.

### 2.2 Graphs

In general, graphs model discrete relationships between objects. Graphs are often described as an ordered pair of vertices and edges, or \( G = (V, E) \). Each object in the graph can be described as a vertex, and the relationships between vertices are represented with edges. For example, a social network can be modeled as a graph, where people are represented as vertices and an edge between two vertices shows that those two people know each other. Somebody interested in different properties of the social network could create algorithms to run on the graph. For example, someone may want to find a list of mutual friends between two people, or find cliques where everybody in some group knows everybody else in the group. While these questions seem fairly different, each can modeled in similar ways using graphs.

Modeling various relationships is often accomplished with different kinds of graphs.
If a graph is restricted so that only one edge may exist between two vertices, that is referred to as a simple graph. Alternatively, a multigraph can have many edges between vertices. A directed graph has edges which represent relationships that only go one way. For example, projects may consist of many tasks, some of which are dependent on others. Modeling this as a directed graph would help analyze what tasks are critical to completing the entire project on time.

Fundamentally, the goal is to abstract some relationship between objects to make meaningful decisions. Many algorithms have been created to solve problems for graphs in general, but in some cases, developing algorithms explicitly for a particular model is ideal. This presents a trade off between designing an algorithm general enough for use in multiple domains and designing it to exploit properties of that particular problem.

### 2.3 Classification

Because we aim to solve a general problem, but also want to potentially tune for domain-specific properties, we turn to classification. Machine learning aims to learn solutions to problems without explicitly programming the solution. In many cases, this involves using a general set of knowledge and extracting a few important features which represent that knowledge. Often, this boils down to an optimization problem in which a classifier starts from a guess and improves that guess until it can not improve much further. This transforms the problem from finding an exact algorithmic solution into searching for some optimal point.

For many problems, there is some set of expert knowledge that is developed through learning. A doctor spends years learning what is or is not important when diagnosing patients. Classification problems make use of similar features when learn-
In this example, if air pressure is high, the classifier will always say it is sunny. If not, then in low temperatures it will snow, while in high temperatures it will rain.

A very simple classifier is the decision tree. This model takes in \textit{training examples} as inputs and builds a tree for later use on previously unseen \textit{testing examples}. Using medical treatment as an example, a patient might have some list of traits such as age, weight, sex, history of illness, and so on that might all contribute to that patient’s health. A decision tree can be built by finding which traits are most important and effectively splitting all the examples based on that decision. This process repeats until a decision is made for every training example. An illustrative example of this using simple weather classification is shown in Figure 2.1.

In general, supervised learning attempts to achieve this sort of classification au-
tomatically. The classifier learns using *training examples* consisting of labels representing the class each example belongs to and a feature vector containing important properties of the example. This differs from unsupervised learning where the classifications are not known to begin with. In the context of classifying robot behaviors, we assume we have some proven or canonical set of behaviors, so labels for each behavior can be generated automatically. Because of this, we focus on supervised learning.

Support vector machines are models that separate clusters of labeled data. Figure 2.2a shows an example of separating two clusters of examples with the dotted line. In the case of two dimensions, the separator is a line, but the same concept applies using hyperplanes in higher dimensions. While there are many lines that could separate the data, arguably the best line is the one that leaves the widest gap between the two clusters. Support vector machines determine the optimal hyperplane which separates the data so that new examples have a better chance of being correctly classified. This avoids problems with overfitting, where training data is successfully classified, but the model might be too limited to classify new data well. A more comprehensive tutorial on support vector machines is given in Burges (1998) [40].

Practically, there will often be outliers in data which make it so that examples are not linearly separable. For example, in Figure 2.2b, one of the blue examples is located in the middle of the red examples, so separating the data with a line will not work. Despite this, the same hyperplane from Figure 2.2a will still reasonably classify the data. This method of allowing for some slackness is the soft margin hypothesis [41], and is useful in classifying real world data where these sorts of outliers frequently occur.

Since support vector machines are binary classifiers, classifying multiple clusters with support vector machines alone is impossible. To solve this problem, we can
Figure 2.2: Left: Example of separating clusters of examples with the dotted line. The solid lines show the margin between the two clusters. Right: Clusters that are not linearly separable. The dotted line still separates all examples except for the blue outlier.

use multi-class support vector machines [42]. Two common techniques are one-vs-all and one-vs-one classifiers. As shown in Figure 2.3a, one-vs-all creates classifiers by examining one cluster at a time, and treating the rest of the examples as another aggregated cluster. Assuming there are $h$ clusters, a new data point can be classified by going through each of the $h$ classifiers and choosing the one that maximizes the likelihood of the data point being in that cluster.

On the other hand, one-vs-one, as shown in Figure 2.3b, generates a classifier for each pair of clusters. A new example is classified by iterating through each classifier and counting in which cluster the example most frequently belongs. Because the one-vs-one method creates a classifier for each pair, there are $O(h^2)$ classifiers as opposed to $O(h)$ in one-vs-all. Despite this, one-vs-one tends to be more accurate in practice while still running in the same amount of time as one-vs all.
(a) One-versus-all classifiers for multi-class support vector machines. Each classifier separates a single cluster from every other cluster (a-c). New examples are classified by iterating through each classifier and choosing the one which best fits the example.

(b) One-versus-one classifiers for multi-class support vector machines. Each classifier separates pairs of clusters (a-c) and the group of classifiers (d) is used on new examples.

Figure 2.3
2.4 Graph Kernels

Designing features fundamentally depends on the problem being solved. In the case of the doctor from our previous example, it is obvious that features revolve around the patient’s health. With graphs, it is less clear which features should be used. At first glance, using properties of the graph such as the number of edges, the degree of each vertex, or even isomorphism seem as though they should be useful. However, using isomorphism is not necessarily robust to graphs that may actually be similar, but appear very different because of varying numbers of vertices or edges. Because of this, we use methods of measuring similarity that rely only on the structure of the graph.

Graph kernels focus on examining small parts of graphs to describe the entire graph. State of the art methods are based on walks, subgraphs, and subtrees. Walk-based kernels count how many common walks exist in two graphs, where a walk is a series of vertices connected by edges. An example of this is the random walk kernel [43, 44]. In principle, two graphs that have many common sequences are more likely to be similar, so counting random walks should measure similarity. If vertices in a graph are labeled, this gives a measure of how frequently some sequence of the same labels in two graphs. If vertices are unlabeled, random walks can simply describe the lengths of walks, which may vary between graphs. The shortest path kernel [45] is a method based on the Floyd-Warshal algorithm used to count the shortest paths between pairs of vertices. This measure expresses similarity in a principled way, while also running in polynomial time.

Subgraph kernels examine subgraphs of the graph with finite vertices to measure the structure of the graph as a whole [1]. For example, triadic closures occur often in social networks, so determining their frequency will give a measure of how similar
different graphs are. Because of this, previous work has shown ways to analyze many
graphs in terms of graphlets [46] as well as finding methods to more quickly calculate
certain graphlets [47]. Because we use this method in our research, we give a more
in-depth explanation in Section 4.2.

Subtree kernels are similar to subgraph kernels, but look specifically at the local
neighborhood of each vertex [48]. Refinements of this kernel look only at subtrees with
bounded numbers of children from each vertex to make computation more feasible [49].
Weisfeiler-Lehman graphs extend this concept further to generate kernels for graphs
with labeled or unlabeled nodes [50]. This method opens up possibilities to using
kernels on much larger scale problems.

A more recent approach uses covariance matrices instead of directly finding his-
tograms of features. Power kernels [51] summarize many properties of graphs in
general without having to deal with the combinatorial complexity that shows up in
finding substructures of graphs. By working with the adjacency matrix directly using
well known methods in linear algebra, this allows for fast computation of similarity
even on very large graphs.

These kernels attempt to describe similarity between graphs without necessar-
ily relying on domain knowledge. Although these methods seem abstract, they all
fundamentally extract important properties of the graph regardless of what individ-
ual graphs represent. Because of this, these methods are very successful in areas
where graphs fully model the problem and are clearly separable, including social
networks [52] and chemical informatics [53].
Chapter 3

Motion Graphs

Graphs are frequently used to abstract relationships between objects. From social networks [54] to routing flights [55] and biology [56], graphs simplify problems that would otherwise be extremely difficult. Finding an appropriate model is key for extracting useful information. Motion in continuous spaces is often difficult to capture and requires sampling or decomposing the space in order to make problems simpler to solve [57, 58]. For example, decomposing a continuous environment into cells transforms the problem into only having to work with discrete objects. In many cases, such as path finding, this allows us to use well known methods to find a solution.

This chapter describes motion graphs, a proposed data structure for abstracting motion of individual robots, while still containing enough information to make useful classifications. Although in many cases it makes sense to use measurements of exact positions when solving problems, distributed systems provide several benefits that encourage using graphs. At a high level, using motion graphs attempts to reduce the complexity of many algorithmic and engineering challenges inherent to swarm robotics by taking noisy sensing or movement and translating it into something manageable.

3.1 Generating Graphs

A motion graph consists of vertices representing observers and directed edges representing motion. Each edge also contains information about an observed agent and the
Figure 3.1: Generating edges for *motion graphs*. The robot travels along the path and enters the range of A at time $t_i$. At time $t_j$, the robot has left the range of A but is still sighted by B, so an edge is formed between A and B with associated time $t_j$.

time motion was detected. When an agent is within sensing range of an observer, we say it is *sighted* by the observer. When an agent stops being sighted by an observer, an edge is created between the first observer and the next observer that senses it as shown in figure 3.1. An alternative perspective is that a single *motion graph* is formed for each robot using the same method of generating edges, where edges exist between observers and are labeled by time. Viewed this way, a *motion graph* can be constructed by taking the union of multiple *motion graphs*. Choosing between these two options is mostly a matter of implementation, although both methods represent the same information. An example of *motion graphs* for individual robots is shown in Figure 3.2. The union of the graphs of all robots for the same example is shown in Figure 3.3.

As an engineering issue, an additional restriction can be used to ensure edges can only be made between observers that see each other, or alternatively, by communicat-
Figure 3.2: Examples of motion graphs generated from individual robots. Each of these graphs was generated using five seconds of simulated data. Even with this small amount of information, some behaviors are visually easy to tell apart from each other.
Figure 3.3: Motion graphs for the entire set of robots. Behaviors like resource collection and static motion are clearly visually different from other behaviors.

Abstracting movement as motion graphs also lends itself to the way robots already communicate. Each robot can communicate with nearby neighbors when an agent has left the observer’s sensing range, and the neighbors that now sense the agent can use this information to generate an edge in the graph. This avoids the need to sense and constantly convey relative coordinates. As such, this method not only abstracts motion but reduces communication requirements.
Figure 3.4: Generating multiple edges at once. When the robot reaches the point on the path shown by the red dot, it leaves the range of $C$, but remains in range of $A$ and $B$. Because of this, edges $(C, A)$ and $(C, B)$ are created.

With this said, there are a few situations that require a little extra care, especially when choosing the number of observers and their range of vision. In some cases, multiple edges can be generated at once, as shown in Figure 3.4, where edges are generated from one observer to two other observers. Ideally, any method we use to analyze motion graphs will keep issues like this in mind. For example, if we can only extract a single representative path for a robot, it might make sense to remember the longer of two paths, since this gives a more fine representation of the path.

Ignoring situations where edges may not be generated can also cause issues. As an extreme example, if every robot can sense the entire environment, no motion will be recorded since observed agents never leave any observer’s sensing range. While this seems like a significant problem, all it indicates is that there is a point where there are too many robots for this model to work effectively. From a practical perspective using actual sensing ranges, this is unlikely to be an issue, but requires more thorough testing.
Chapter 4

Classification Methods

4.1 Preliminary Tests

While determining the feasibility of the problem, we ran a much smaller set of tests to determine whether classifying motion was possible. Part of the motivation for these early tests was determining which features would be important, so we initially examined features general to all graphs, but not necessarily related to the structure of the graph. This included properties such as the maximum and minimum degrees of each vertex, as well as the mean, total, and variance of the number of edges. The motivation for this was partially because these values are straightforward to compute while still representing features of the graph as a whole.

We tested four of the five behaviors we describe in more detail in Section 5.2. Briefly, the behaviors tested were idling, also known as static behavior, random motion, resource collection, and coverage, also known as dispersion. There were twenty examples of each behavior, with half used for training and the other half used for testing. After gathering features from the training set, we manually built a decision tree as shown in Figure 4.1 without using the testing set. The results of this are shown in Table 4.1. Using this tree, we correctly predicted 37 out of 40 examples in the testing set, achieving 92.5% accuracy. While this is still a toy example, it proved that classification was at least possible and that better classifiers were worth looking into.
Table 4.1: Confusion matrix for early tests. The only mistakes were there examples which were classified as coverage instead of random motion.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>Idle</td>
<td>1</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
</tr>
<tr>
<td>Resource</td>
<td>0</td>
</tr>
<tr>
<td>Coverage</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.1: The decision tree generated for our proof of concept. Ellipses represent decisions while rectangles show classified behaviors.
4.2 k-Graphlets

In order to properly distinguish different graphs, feature vectors must be designed to meaningfully represent the graphs. If features are only built around classifying a handful of behaviors, a classifier will not be general enough to handle graphs not explicitly represented by the training set. Because of this, generated features need to work with all motion graphs. However, since motion graphs fundamentally abstract motion, it makes sense to design features specific to motion graphs, but not necessarily for all graphs. For example, if robots mostly move in a local region, a motion graph is more likely to have many cycles. Similarly, robots moving between two waypoints will likely create longer paths with fewer short cycles. In some cases, there may be parts of the environment where no observed agents travel. Because of this, we try to make use of features that consider local structures while keeping in mind the motion of the system as a whole.

Graphlets have also been shown to be a sufficient statistic of graphs and practical to compute [1]. By sampling vertices, using graphlets can act as a proxy for density, as well as a measure for frequency of local cycles. For motion graphs in particular, it is important to use a kernel that not scales well to a large number of nodes but also accurately measures similarity between graphs. Because graphlets are simple to implement and lend themselves well to sampling, we decided to use k-Graphlets as a baseline to test whether we could classify motion graphs at all. In particular, we chose to focus on 4-Graphlets to give enough expressiveness in computed feature vectors, while still be computationally fast.

Given a graph $G = (V, E)$, a vertex-induced subgraph of size $k$, or a k-Graphlet is the graph containing vertices $v_1, ..., v_k \in V$ and all edges adjacent to only these vertices. To generate k-Graphlet features, we represent motion graphs as an adjacency
Figure 4.2: An example of a feature vector generated using 3-graphlets. The graph above contains four vertices and three edges. Each subset of three vertices is isomorphic to one of graphs on the bottom. Three subsets of vertices have only one edge, while one subset of vertices contains three edges. Looking at these as a histogram of counts returns the feature vector $(0, 3, 0, 1)$.

Matrix. All edges in the motion graph are transformed into undirected edges, such that entry $(i, j)$ is the number of edges between vertices $v_i$ and $v_j$. Since there are many more possible isomorphic subgraphs with directed edges, counting features is simplified. For every set of $k$ vertices, a $k$-Graphlet feature vector is determined by finding which of the possible graphs with $k$ vertices is isomorphic to the induced subgraph. The resulting histogram of counts is then returned as a feature vector. An example of this is shown in Figure 4.2.

Finding isomorphism between graphs is in NP, but in some cases, such as planar graphs [59] or for small enough $k$, such as $k \leq 5$, isomorphism is relatively easy to determine. To simplify finding isomorphic subgraphs, we use an elegant trick for $k = 4$. By counting the degree of each vertex in a simple graph with four vertices, we can generate a set. A simple example for graphs with three vertices is shown...
Figure 4.3: Sets of degrees of vertices for each possible graph containing three vertices. Each possible set is unique, so finding isomorphic graphs from a subgraph is straightforward.

Figure 4.4: Sets of degrees of vertices for each possible graph containing four vertices. Each subgraph also maps to a unique set. (Adapted from Shervashidze (2009) [1])

in Figure 4.3. Figure 4.4 shows how every possible simple graph with four vertices maps to a unique set. Since each set is unique, finding features equates to generating the set and using it as a key in a dictionary to store the number of occurrences of a subgraph.

However, this ignores the fact that motion graphs are multigraphs. To alleviate this issue, we find which one of the possible graphs is isomorphic to the subgraph. We then calculate how many possible graphs are isomorphic to the underlying simple graph assuming only one edge can be chosen between two vertices, as shown in Figure 4.5. If there is frequent local motion, this method will reflect this by taking into
account multiple edges generated over time. As a technical detail, due to issues with representing integers, we add the logarithms of the multiplied values to make these counts manageable.

\subsection*{4.3 Path Lengths}

A major issue with the 4-Graphlet kernel is that a lot of information is lost for the sake of simplifying computation. Additionally, creating a distributed algorithm is not straightforward and may be communication intensive. Because of this, we utilize a method that works naturally with how we generate motion graphs. In many behaviors, robots will exhibit some patterns in their motion. For example, if a group of robots is trying to form a uniform distribution across a space, a single robot will only move in some local area. On the other hand, if robots are only traveling directly from one point to another, there will be relatively few or very long loops in the motion graph.

While random walks measure similarity in a simple way and exploit properties such as sparsity, low effective rank, and Kronecker product structure\footnote{Kronecker product structure}, we want to make use of properties unique to motion graphs. Taking inspiration from random
Figure 4.6: Two graphs with different walks. The upper graph contains a cycle generated by the first three edges. In the algorithm, a path of length 3 would be recorded in the feature vector, and the path would be reset. The lower graph would not record anything, since the walk contains no cycles. This separates short local motions from longer global motions.

walks, we build features from path lengths to abstract motion. A path $P$ in a graph $G = (V, E)$ is a sequence of vertices $P = v_0, v_1, \ldots, v_n \in V$ such that $(v_i, v_{i+1}) \in E$ and no $v_i$ is repeated, or simply, a path is a walk with no cycles. An example of this is shown in Figure 4.6.

Algorithm 4.7 describes how to generate path length feature vectors. In a distributed system, each robot can maintain a local list of path lengths to later be aggregated in some central location before classifying. If we must use a central processor, aggregation can be accomplished by creating a spanning tree and passing counts up to the root, so there should not be a significant amount of overhead.

The idea of the path length algorithm is to determine the distribution of local and
**Input:** List of edges $E$ from *Motion Graph* $G$ sorted by edge times

Max path length $k$

**Result:** Path Length Feature Vectors

1. for each $v \in V$ do
   2. $v.paths \leftarrow []$;
   3. end

4. for $i \leftarrow 1..k$ do
   5. $counts[i] \leftarrow 0$;
   6. end

7. for each edge $(x, y) \in E$ from robot $r$ do
   8. $currentPath \leftarrow x.paths[r]$;
   9. if $y \in currentPath \text{ or } len(currentPath) == k$ then
      10. $counts[len(currentPath)]++ = 1$;
      11. $y.paths[r] \leftarrow [y]$;
   12. end
   13. else if $len(y.paths[r]) \leq len(currentPath)$ then
      14. $y.paths[r] \leftarrow x.paths[r].append(y)$;
   15. end
   16. $x.paths[r] \leftarrow []$;
17. end
18. return $counts$;

Figure 4.7: Algorithm for generating path length features
Figure 4.8: On the left, a sample path for a robot. On the right, the motion graph for this path. Note that there are two cycles representing a single motion. Following the algorithm, both cycles will be recorded as paths of length two, resulting in a possible overcount.

Global motion. Resource collection will have few cycles, for example, while random motion will have many. The path length algorithm also makes full use of times associated with edges and does not need to simplify directed edges into undirected edges, so no important information is lost.

In more detail, the algorithm takes a motion graph as input and a max path length, $k$. A list of currently sensed robots is maintained for each vertex in the graph. This list contains the path of observers that each agent has passed, and is initially empty. For each edge $(x, y)$ in the graph, sorted by time, the path for robot $r$ is copied from vertex $x$ to vertex $y$, with $y$ appended to the path. If either the max path length is reached or a cycle is detected, the path length is written to the feature vector and the path for that robot is reset to the current observer. If multiple paths end at the same vertex, the longest path is stored. Once all edges have been seen, the feature vector is returned as a histogram of counts.

The time complexity of the path length algorithm is fairly reasonable. For each
edge, the algorithm must check whether a cycle exists, which can be calculated using a linear scan through the path to check if vertex $y$ has already been seen. Since paths are at most length $k$, the time complexity of the entire algorithm is $O(kE)$. Since $k$ is usually very small, we can consider this a constant, so the algorithm is $O(E)$.

An important design choice in developing this algorithm was attempting to reduce message sizes as well as the memory required to store the state of the system. By choosing the longest path, only the most information rich path is maintained. This means that the only paths that observers need to store at any given moment are the robots with its range of vision. From a practical perspective, this means the number of paths maintained by each observer will be very small.

Additionally, only paths of at most length $k$ are considered. In a distributed system, this helps reduce message sizes and memory requirements, but also gives a tunable parameter to match each system’s needs. Using path lengths also abstracts short term memory, which should ideally make the system more robust.

Fortunately, for each observer, there is no way for a robot to both enter and leave the sensing range of the observer in the same time step. Even if two edges are created with the same associated time, the path length will still be measured correctly. However, path lengths may overcount when two cycles end at the same time, as shown in Figure 4.8. One path will detect a cycle and the path length will be recorded in the feature vector. After this, the other path is seen, but appears as another cycle. It is possible for observers to remember when robots were recently seen to mitigate overcounting, but informal experiments show it does not seem to be necessary.
Chapter 5

Simulations

Since we wanted to first prove that classification was possible, we aimed to explore the strengths and weaknesses of the proposed data structure and classifiers without introducing unnecessary complexity. Because of this, we chose to model a variety of distributed behaviors that would test the limits of the data structure without necessarily being pathological. Although it is not the primary focus of this thesis, we anticipate the need to deal with hardware issues such as mitigating the need to manage noise in both sensing and communication. Code for simulation and analysis was written in python. For classification, we used scikit-learn [62], which provides python bindings for LIBSVM [63].

5.1 Assumptions

In this problem, we observe some group of uniquely identifiable agents moving in a swarm. Agents might be robots in the case of swarm robotics, or something like mobile phones in a sensor network. In many cases, differentiating one object from another is a challenging problem [60], but we assume we can identify individual objects. With this said, identifying and localizing robots is possible [61], assuming a robot can be identified as a robot as opposed to some other object.

Simulations were run in a bounded unit square with no obstacles as shown in Figure 5.1. While some behaviors display unique properties when obstacles are present,
Figure 5.1: A snapshot of the simulation environment. Blue dots represent observers while red robots represent agents. The behavior in this example is resource collection, with home base in the top left corner, and resources in the other three corners.

we tested in environments without obstacles for the sake of consistency and simplicity.

We aimed to solve the problem using minimal sensing abilities while also trying to find a fully distributed solution. Using GPS simplifies the classification problem greatly, but does not necessarily scale well. We assume individual robots have finite sensing and communication ranges, but that they also have a fully connected communication network so that any two robots may send messages to each other, as shown in Figure 5.2.

5.2 Behaviors

Five different behaviors were simulated to test whether clearly distinguishable behaviors were easily classifiable. In a few cases where the steady states of each behavior are very similar, such as dispersion and static motion, we also wish to understand when exactly behaviors are misclassified. For the sake of simplicity, the robots are
Figure 5.2: A possible connectivity graph for the observers. Note that a message can be passed from a robot to any other robot using a multi-hop routing protocol.

assumed to be holonomic so they can change direction instantly. This should not limit what is possible for classification using our methods, but only simplified writing algorithms to test behaviors.

- **Static Behavior**: Robots are idle. This is fairly common in many situations where robots are powered off or have reached a steady state, so being able to recognize and separate this from behaviors where robots are moving is important. Since there is no movement, motion graphs generated from this behavior contain no edges.

- **Random Motion**: Robots move at a constant velocity, but in a random direction at each time step. This is intended to test whether noisy behaviors can be distinguished from more meaningful behaviors. Since robots may frequently jitter between sensing ranges of two observers, this behavior also aims to explore whether it is straightforward to classify frequent local motions.
• **Dispersion:** Robots attempt to move out of each other’s sight. This behavior was inspired by coverage control algorithms [64], but is of particular interest because robots should travel across most of the space. It is also common enough in many swarm applications to need to disperse fairly evenly throughout an environment, so being able to classify this behavior is of practical use.

• **Contract:** Robots try to move closer to, or aggregate with other nearby robots. Aggregation is frequently seen in nature, but is also important for focusing robots in some area, or for managing movement of the entire group [65]. In our simulations, robots may overlap, usually resulting in several tight clusters. This directly tests whether features that should be obvious to a human observer are easily classified. Similar to dispersion, we also want to observe whether common behaviors are easy to classify.

• **Resource Collection:** Robots travel between waypoints. Resource collection is directly inspired by foraging behaviors seen in many insect swarms [4, 14, 66]. We use a simplification where robots simply travel between waypoints and a home point. This leads to many long motions without very many local cycles.

5.3 Validation Techniques

We used 10-fold cross validation to measure accuracy, and took the median of a small number of independent tests to measure time. For 4-Graphlets, we tested classification using various number of samples including 1, 5, 10, 25, 50, 100, 500, 1000, 5000, 10000, and 20000. To test path lengths, we varied the maximum length between 1 and 10, inclusive. While these tests are helpful for understanding the relationship between the variables described in this section, accuracy, and time, they
do not help fully explain why certain behaviors may be more difficult to classify than others.

In order to understand where kernels succeed, we split the entire data set into two even sets so that each data set contains similar data. We first consider learning with half the data and training on each subset of the data. This helps identify which behaviors are difficult to classify in general. We also consider training and testing on the same subset of data for each subset. Doing this helps understand whether confusion between behaviors is due to the dataset as a whole or properties of certain subsets of the data.

We generated data while varying the number of observers and agents, the range of vision, and how long simulations were run. We vary the number of observers and the number of agents to range between 40 and 140 robots. The motivation for this is to expose any weakness in scalability. If too few observers are used, it is possible there won’t be enough information to classify with. If too many observers are used, it may take too long to classify.

Range of vision for agents was varied between 0.06 and 0.15 units, since this seemed to be an optimal range for most behaviors. Range of vision for observers was varied between 0.15 to 0.25 units, as this range seemed to most frequently result in connected graphs. Varying the range of vision attempts to expose potential engineering issues, especially if it is required that vision be a certain range. Trivially, if all observers can see the entire space, then no edges will be generated in the motion graph, which implies that some ranges of vision should be better for classification than others.

The amount of time in each simulation was varied between 5 and 30 seconds to test whether methods of extracting features scaled with the number of samples. In the best case, increasing the amount of time spent observing should increase accuracy. If
spending more time measuring behaviors decreases accuracy, there is likely an issue. This may also expose issues in classifying some behaviors, such as dispersal, where robots quickly reach a steady state. Again, for the purpose of engineering, this may simply inform how long samples need to be taken to get reliable results.

For each of the 5 behaviors, 100 graphs were generated for each parameter varied above. This resulted in 500 graphs for each behavior, in addition to the 80 graphs from the original tests, for a total of 2580 graphs in our dataset.
Chapter 6

Results

In this chapter, we briefly describe the overall results of our simulations and then dig deeper into the strengths and weaknesses of k-Graphlets and path lengths. While we examine accuracy and timing results, we also look into what variables cause each method of generating feature vectors to succeed or fail, such as the number of robots, their sensing range, and the amount of time sensing the environment.

6.1 Overall Results

Accuracy and timing results for the 4-Graphlet and path length methods are shown in Figure 6.1. Both 4-Graphlets and path lengths were classified with around 85% accuracy. The path length kernel reached 85% accuracy when $k = 4$, which may validate the strategy of separating short from long cycles to distinguish motion graphs. Even so, using a path length of 1, or what is essentially an edge count, still resulted in 73% accuracy, showing that at least for our data, an abstraction of the amount of motion is enough to separate most behaviors. This also clearly surpasses the results of early tests which reached at most 78% accuracy using 3-Graphlets on simple directed graphs as described in Section 4.1.

Since 4-Graphlets rely on sampling, the amount of time spent generating features increases linearly with the number of samples. With this said, reasonable results can be achieved between 1000 and 5000 samples, but this still takes at most 330 seconds.
to generate features and 250 seconds to train. Since both generating features and training took a significant amount of time, using 4-Graphlets seemed infeasible for practical use.

In comparison, using path lengths will always use time linear in the number of edges irrespective of the max path length. For this data set, it always took around 90 seconds to generate features for the entire dataset. Perhaps surprisingly, the training time reduced from 18 seconds for \( k = 1 \) to around 0.2 seconds for \( k \geq 3 \), which is at least 1200 times faster than using 4-Graphlets and, more importantly, well suited to a real swarm of robots.

Because classifying with path lengths works so quickly, this also indicates that features generated using path lengths may be easily separable. While this should not be true for all behaviors, it is at least promising for future work. In particular, knowing the distributions of feature vectors for a wider variety of behaviors will help validate path lengths as a useful method.

Precision and recall data for each variable are shown in Table 6.1 for both 4-Graphlets and path lengths. The entire data set is split in half for training and testing. For results labeled all data, a classifier is generated from the training half and is trained on the entire testing set. This helps act as a baseline to understand where classifiers may have issues, with respect to each variable.

For each variable we also classify exclusively on the subgroups of data for that variable to understand which for variables the classifiers are more robust. To examine this further, we also train exclusively on the data from each subgroup and then test on the other half of the data in that same subgroup. We specifically want to discover which variables might be most prone to issues with overfitting.

Overall, precision and recall range between 0.8 to 0.9 for all subsets, but not for the
Figure 6.1: Timing and Accuracy results for 4-Graphlets (top) and path lengths (bottom). From left to right are tables for accuracy, amount of time spent generating features, and training time. Results for 4-Graphlets are shown with respect to the number of samples, while results for path lengths are shown with respect to the maximum path length.
Table 6.1: Precision and recall for all subsets of data for 4-Graphlets using 10000 samples and path lengths using max lengths of 10. No values are included for all data sub-groups since this is already equivalent to training on all data.

<table>
<thead>
<tr>
<th>Subset of Data</th>
<th>Testing Samples</th>
<th>All 4-Graphlets Precision</th>
<th>All 4-Graphlet Sub-groups Precision</th>
<th>All Path Lengths Precision</th>
<th>Path Length Sub-groups Precision</th>
<th>Recall</th>
<th>All 4-Graphlets Recall</th>
<th>All 4-Graphlet Sub-groups Recall</th>
<th>All Path Lengths Recall</th>
<th>Path Length Sub-groups Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Data</td>
<td>1290</td>
<td>0.83</td>
<td>-</td>
<td>0.85</td>
<td>0.85</td>
<td>-</td>
<td>0.83</td>
<td>-</td>
<td>0.85</td>
<td>-</td>
</tr>
<tr>
<td>Vary Observers</td>
<td>250</td>
<td>0.91</td>
<td>0.91</td>
<td>0.85</td>
<td>0.85</td>
<td>0.84</td>
<td>0.82</td>
<td>0.85</td>
<td>0.85</td>
<td>0.82</td>
</tr>
<tr>
<td>Vary Agents</td>
<td>250</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>Vary Observer Vision</td>
<td>250</td>
<td>0.80</td>
<td>0.79</td>
<td>0.86</td>
<td>0.86</td>
<td>0.84</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>Vary Agent Vision</td>
<td>250</td>
<td>0.85</td>
<td>0.84</td>
<td>0.82</td>
<td>0.82</td>
<td>0.83</td>
<td>0.82</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td>Vary Sample Time</td>
<td>250</td>
<td>0.83</td>
<td>0.83</td>
<td>0.86</td>
<td>0.86</td>
<td>0.91</td>
<td>0.90</td>
<td>0.92</td>
<td>0.91</td>
<td>0.91</td>
</tr>
</tbody>
</table>

same variables in each method. From this, we can conclude that varying the number of agents or their range of vision negatively impacts accuracy. To fully understand why this is the case, we examined the variables more closely to find for what values each method succeeds or fails.

6.2 4-Graphlets

Features generated using 4-Graphlets were successfully classified with a mean accuracy of 85% and a standard deviation of 5.5% using 10-fold cross validation. Figure 6.2 shows a confusion matrix for a classification of the entire set of testing examples. In general, static behavior was classified perfectly while resource collection was only occasionally classified incorrectly. Dispersion and random motion were the most likely to be classified incorrectly with around 60% accuracy in the worst case.

At a high level, we can see in Figures 6.3 and 6.4 that 4-Graphlets classified relatively well regardless of which subset of graphs are used for training. With this
Figure 6.2: Confusion matrix for 4-Graphlet feature vectors. Training and testing data were taken from all subsets of data. Overall, 4-Graphlets classified behaviors with 85% accuracy, but showed some difficulty with random motion. Labels in this figure are the same as Figures 6.3 and 6.4.
Figure 6.3: Confusion matrices for 4-Graphlets using a classifier generated from the entire data set. Labels for these matrices are the same as in Figure 6.2. The subsets of data tested for each matrix were, (a) the entire data set, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the amount of time the simulations ran, (f) varying the range of vision for agents or observers, (g) varying vision for only observers, (h) varying vision for only agents.

said, it is clear that in some cases, such as varying the number or range of vision of agents, behaviors were not classified nearly as well. However, this still does not give an intuition for why there may be confusion between behaviors.

To find when exactly confusion in classification occurs, we examined how predictions vary with respect to each variable as shown in Figure 6.6. Each stack graph is separated into a subgraph for each behavior, and the predictions for each behavior are separated evenly into five segments, ranging from lowest to highest value for that
Figure 6.4: Confusion matrices for 4-Graphlets using a classifier generated from each subset of varied parameters. The subsets of data tested for each matrix were, (a) varying the amount of time the simulations ran, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the range of vision for agents or observers, (f) varying vision for only observers, (g) varying vision for only agents.
Figure 6.5: An example stack graph to show how predictions vary with respect to the number of agents. Data points are separated evenly to show when classifications for each behavior succeed or fail.

Figure 6.5: An example stack graph to show how predictions vary with respect to the number of agents. Data points are separated evenly to show when classifications for each behavior succeed or fail.

variable. We show an example stack graph for the disperse behavior when varying the number of agents in Figure 6.5. The number of agents in each motion graph ranges from 40 to 140, so the data is split into bins containing 40-60, 60-80, 80-100, 100-120, and 120-140 observers. The predicted classifications for each range of values are shown for each bar. In this particular example, accuracy decreases as the number of agents increases. Graphs in Figure 6.6 are joined into a single graph for the same variable to more easily compare how classification accuracy changes with respect to each variable.

In the cases where behaviors are difficult to tell apart, robots often reached a steady state that appears similar to another behavior. For example, dispersion is
most affected by changing the number or range of vision of agents. As either of these values increase, agents will not have enough space to separate from other robots and will start to jitter. Random motion exhibits the same type of jitter, so dispersal will appear like random motion.

Varying the range of vision of the observers also causes confusion between the contract, random, and resource collection behaviors. If the range of vision is limited, both random motion and resource collection look similar to the contract behavior. On the other hand, if observers have a larger range of vision, the contract behavior looks more like random motion. One of the major consequences of a larger range of vision is that edges in the motion graph will span across more space. Since any motion will be seen by more observers in the system, this jitter may be overrepresented in the motion graph and reduce classification accuracy.

Ideally, more observations result in non-decreasing accuracy. Unfortunately, increasing the number of observations led to random motion being frequently classified as resource collection. It may simply be that the most important feature for resource collection is some measure of dense graphs. With more time, random motion may also end up building many local dense subgraphs, so the two behaviors appear similar.

These observations indicate that a possible weakness of using 4-Graphlets is differentiating frequent local motions. Since graphlets only really give a measure of local structure in the graph, the features may not be rich enough to distinguish global differences. Many of the difficulties in classification were due to jittery behaviors all appearing similar while ignoring any other emergent features in the rest of the system.
Figure 6.6: Comparison of predictions using 4-Graphlets for each variable (Number of observers or agents, range of vision of observers or agents, and amount of time). Each chart shows classifications of each of the five behaviors. For example, increasing the range of vision of the agents causes dispersal to appear like random motion. This may be because agents jitter a lot while dispersing with a high range of vision, which also occurs in random motion.
6.3 Path Lengths

Using path lengths as features with a maximum length of 10 also classified with 85\% accuracy and a standard deviation of 5\% using 10-fold cross validation as seen in Figure 6.7. In the worst case, random motion and dispersal were correctly classified around 56\% of the time. On the other hand, resource collection and static behavior were always classified correctly.

Figure 6.7: Confusion matrix for path length feature vectors. Training and testing data were taken from all subsets of data. Path lengths were classified with 85\% accuracy overall, but showed some confusion between dispersal and random motion. Labels in this figure are the same as Figures 6.8 and 6.9.
Figure 6.8: Confusion matrices for path lengths using a classifier generated from the entire data set. The subsets of data tested for each matrix were, (a) the entire data set, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the amount of time the simulations ran, (f) varying the range of vision for agents or observers, (g) varying vision for only observers, (h) varying vision for only agents.
Figure 6.9: Confusion matrices for path lengths using a classifier generated from each subset of varied parameters. The subsets of data tested for each matrix were, (a) varying the amount of time the simulations ran, (b) varying the number of agents or observers, (c) varying only the number of observers, (d) varying only the number of agents, (e) varying the range of vision for agents or observers, (f) varying vision for only observers, (g) varying vision for only agents.
Confusion matrices for all subsets of data are shown in Figures 6.8 and 6.9. Although path lengths were roughly as accurate as 4-Graphlets, there seemed to be a higher amount of confusion between dispersion and random motion. Perhaps even more surprisingly was the false positives for static behavior that were only seen when varying the range of vision or number of agents. However, this may be a quirk of the path length algorithm. Since the algorithm does not output its current state at the end, a path that never becomes a cycle or reaches the max length is not written to the feature vector. Informal testing showed that accuracy did not change significantly with this change, so it was not included in these tests. However, if it is important to differentiate static from non-static behavior, it may be worth further research into the effects of including this modification to the algorithm.

We examined how predictions using path lengths vary with respect to each variable in Figure 6.10. Increasing the number of observers or the range of their vision seemed to improve accuracy, which may simply come from using a finer grain of features. The only case where this was not true is when the number of observers was increased while agents disperse, which only resulted in around 50% accuracy. It is difficult to understand why this may be the case, but a possible explanation is that using more robots might lead to many of the same issues as 4-Graphlets, where nodes in the motion graph are connected to too many other nodes, so motion is not abstracted as well.

Fortunately, accuracy did not seem to be significantly affected by the amount of time observing agents when using path lengths. Since the path length algorithm does not typically represent long term motion, the amount of time observing should not affect accuracy, assuming there are enough samples. Since this seems to be an issue with 4-Graphlets, this is a definite advantage of using path lengths.
Figure 6.10: Comparison of predictions using path lengths for each variable (Number of observers or agents, range of vision of observers or agents, and amount of time). Each chart shows classifications of each of the five behaviors. For example, decreasing the range of vision of the agents causes dispersal to appear static.
6.4 Kernel Tests

To provide a comparison to previously studied methods, we also measured the performance of a variety of kernels which were previously described in Section 2.4. We used an implementation written by Nino Shervashidze to extract kernel matrices from adjacency matrices [67].

Since these kernels work with adjacency matrices, they do not naturally operate on multigraphs. As such, we modified our dataset to simplify each graph to its underlying directed graph. For each graph and for all \( v_k \in V \), the weight of the edge between vertices \( v_i \) and \( v_j \) was set to the count of the number of edges in the (motion graph) from \( v_i \) to \( v_j \). We then classified using this altered data structure.

10-fold cross validation was used to find the accuracy of each kernel. The results of these tests are shown in Table 6.2. Since the implementation by Shervashidze computes the kernel matrix directly, we measured the amount of time spent generating the kernel matrix as well as the amount of time spent training. Because kernel matrices already handle most of the computation, training times were fairly consistent, typically taking around 30 seconds.

Compared to 4-Graphlets and path lengths, the tested kernels were relatively less accurate. The reason for this may simply be that the kernels only consider the structure of the graphs while ignoring edge weights. By simplifying the motion graphs, that key information may simply be lost. Regardless, these tests seemed to show that measuring local motion or local structures is important.

We can still see that of these kernels, graphlets seem to be most effective. Counting all 3-Graphlets seems to be both fast and accurate enough to be usable, although it may not lend itself well to a distributed algorithm, since it requires global information of the graph.
Table 6.2: Time and Accuracy for preliminary kernel tests sorted by mean accuracy.

Counting all 3-Graphlets was the fastest and most accurate.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Mean Accuracy</th>
<th>Standard Deviation of Accuracy</th>
<th>Time (sec) for Kernel Matrix</th>
<th>Time (sec) for Training</th>
</tr>
</thead>
<tbody>
<tr>
<td>All 3-Graphlet</td>
<td>77.75</td>
<td>2.79</td>
<td>10.11</td>
<td>29.33</td>
</tr>
<tr>
<td>Sampling 5-Graphlet</td>
<td>75.35</td>
<td>2.49</td>
<td>8734.59</td>
<td>31.28</td>
</tr>
<tr>
<td>Sampling 4-Graphlet</td>
<td>73.26</td>
<td>2.78</td>
<td>267.50</td>
<td>30.65</td>
</tr>
<tr>
<td>Sampling 3-Graphlet</td>
<td>70.16</td>
<td>3.54</td>
<td>231.92</td>
<td>29.30</td>
</tr>
<tr>
<td>Weisfeiler-Lehman</td>
<td>68.68</td>
<td>3.82</td>
<td>63.55</td>
<td>27.78</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>63.18</td>
<td>3.47</td>
<td>12.95</td>
<td>30.06</td>
</tr>
<tr>
<td>Connected 3-Graphlet</td>
<td>61.43</td>
<td>4.42</td>
<td>18.27</td>
<td>27.60</td>
</tr>
</tbody>
</table>
Chapter 7

Conclusions and Future Work

In the field of swarm robotics, there is a gap in commonly agreed methods to understand what swarms are actually doing. In some applications, such as diagnostics, researchers directly observe robots or use metrics specific to a particular problem. For other problems, such as automatic deployment or task allocation, we assume that we know what tasks need to be accomplished ahead of time. Classifying behaviors of robot swarms helps solve challenges for all of these applications.

In this thesis, we showed that it is possible to classify various behaviors using robots as distributed sensors. We also provided a novel method of embedding motion of groups robots into a discrete graph to abstract their behavior. Using motion graphs to model our data, we generated feature vectors with 4-Graphlets and path lengths to make use of the inherent structure of the graph. After evaluating 4-Graphlets and path lengths, we found that both methods gave reasonable accuracy, but using path lengths was much faster and easier to implement as a distributed algorithm. To the best of our knowledge, this work is the first to attempt classifying motions of groups of robots in a distributed fashion. The methodology presented in this thesis naturally leads to a fully distributed implementation, which is part of our ongoing work.

An important part of this research is orienting ourselves to be able to deal with real engineering issues. Although training can be done offline, we use the results of this thesis as a proxy to understand how effective the methods presented are. With this said, it is important to be mindful of number and distribution of robots in an
environment. This may simply depend on factors such as what sensors are used, the size of robots, and the shape or size of the environment itself. This work set out to prove that classifying one group of robots with another was feasible and understand what issues there might be before moving towards implementing this on real robots.

Of course, since this research uses simulated data, an obvious next step is running experiments on actual hardware. This will also expose any sensing or communication issues that our assumptions may ignore. In particular, motion graphs attempt to avoid issues with noise in observing the environment. Instead of measuring exact locations, observers only need to detect when nearby robots are within sensing range. Since our methods strive to reduce the impact of sensing and noise issues, using real robots as validation is important.

While the methods presented lend themselves to a distributed system, our research still classifies using a central processor. Even though aggregating collected data is straightforward, this is not enough to make the system fully distributed. Ideally, each robot in a swarm will only require some local view of the environment, but requiring a central processor makes this impossible.

Some classification methods, such as distributed support vector machines, have been developed to solve this problem [68]. Using consensus based methods [69] solves the issue of ensuring robots agree on an action plan, but it is still necessary to test whether these methods work on very large networks. Some generic algorithms for distributed classification have even been developed in sensor networks [12], which show that this is promising even for relatively large networks. As future work, looking into these methods would help figure out the minimum requirements of the network and understand whether this is feasible for a real system.

This thesis already exposes some strengths and weakness of using motion graphs,
4-Graphlets and path lengths for classification, but designing a solution around pragmatic needs will expose more fundamental issues with these methods. The motivation of this work was both to build diagnostic tools for measuring behaviors of multi-robot systems as well as automatically generating classifications of behaviors for problems like automatic task allocation. Classification tools could be used to help understand whether or not a wide variety of canonical behaviors are easy to classify, especially on real systems. As future work, we would like to examine taking classifications and creating action plans to accomplish practical tasks such exploration or transporting objects. Doing so will put us one step closer to accomplishing fully automated and distributed tasks.
Bibliography


