2018

Semi-Informed Multi-Agent Patrol Strategies

Chad E. Hardin
Nova Southeastern University, cehardin@gmail.com

Semi-Informed Multi-Agent Patrol Strategies

by

Chad Hardin

A dissertation submitted in partial fulfillment of the requirements for the degree

of Doctor of Philosophy in

Computer Science

College of Engineering and Computing

Nova Southeastern University

2018

Approval

We hereby certify that this dissertation, submitted by Chad Hardin, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Michael J. Laszlo, Ph.D.
Chairperson of Dissertation Committee

May 14, 2018
Date

Francisco J. Mitropoulos, Ph.D.
Dissertation Committee Member

May 14, 2018
Date

Sumitra Mukherjee, Ph.D.
Dissertation Committee Member

May 14, 2018
Date

Approved:

Yong X. Tao, Ph.D., P.E., FASME
Dean, College of Engineering and Computing

May 14, 2018
Date

College of Engineering and Computing
Nova Southeastern University

2018

Abstract

The adversarial multi-agent patrol problem is an active research topic with many real-world applications such as physical robots guarding an area and software agents protecting a computer network. In it, agents patrol a graph looking for so-called critical vertices that are subject to attack by adversaries. The agents are unaware of which vertices are subject to attack by adversaries and when they encounter such a vertex they attempt to protect it from being compromised (an adversary must occupy the vertex it targets a certain amount of time for the attack to succeed). Even though the terms adversary and attack are used, the problem domain extends to patrolling a graph for other interesting noncompetitive contexts such as search and rescue.

The problem statement adopted in this work is formulated such that agents obtain knowledge of local graph topology and critical vertices over the course of their travels via an API ; there is no global knowledge of the graph or communication between agents. The challenge is to balance exploration, necessary to discover critical vertices, with exploitation, necessary to protect critical vertices from attack.

Four types of adversaries were used for experiments, three from previous research – waiting, random, and statistical - and the fourth, a hybrid of those three. Agent strategies for countering each of these adversaries are designed and evaluated. Benchmark graphs and parameter settings from related research will be employed. The proposed research culminates in the design and evaluation of agents to counter these various types of adversaries under a range of conditions.

The results of this work are agent strategies in which each agent becomes solely responsible for protecting those critical vertices it discovers. The agents use emergent behavior to minimize successful attacks and maximize the discovery of new critical vertices. A set of seven edge choosing primitives (ECPs) are defined that are combined in different ways to yield a range of agent strategies using the chain of responsibility OOP design pattern. Every permutation of them were tested and measured in order to identify those strategies that perform well. One strategy performed particularly well against all adversaries, graph topology, and other experimental variables. This particular strategy combines ECPs of: A hard-deadline return to covered vertices to counter the random adversary, efficiently checking vertices to see if they are being attacked by the waiting adversary, and random movement to impede the statistical adversary.

Acknowledgements

I want to thank all my professors at Nova Southeastern University as well as the dissertation committee members Dr. Laszlo, Dr. Mukherjee, and Dr. Mitropoulos; all of you provided me a very valuable education. I would especially like to thank my dissertation chair, Dr. Laszlo, for his patience and guidance as I worked through this challenging achievement. I also could not have done this without the love and support from my wife Sigrid, thank you so much, Bunny. I need to thank my children Christina (Sebastian) and Riley for your understanding and sacrifice. I know it wasn't easy for you, either. Lastly, I want to thank my employer Koverse and those co-workers for their support, especially my good friend Gadalia O'Bryan.

Table of Contents

List of Tables

List of Figures

Chapter 1.

Introduction

Background

In multi-agent adversarial patrol problems, agents patrol a graph whose vertices are subject to attack by a set of adversaries. In this problem, the number of vertices on the graph exceeds the number of agents and so the agents must have a strategy for effectively patrolling a large number of vertices with a limited ability to monitor all of them at any moment in time. Practical applications of this problem include physical and network security. An example of physical security would be robots protecting some area from intrusion by biological or robotic intruders, such as in a prison, storage complex, or military base. The problem can also be adapted to network security as mobile software agents inspecting computers on a network. The goal of the agents in that problem are to detect and prevent further intrusions or probes on locations that an adversary is attacking.

Problem Statement

The problem is how one or more *agents* that exist on a weighted undirected connected graph can best protect as many vertices as possible on that graph from one or more *adversaries* attempting to *attack* a proper subset of those vertices. The agents *patrol* the graph in an attempt to *thwart* the attacks of adversaries. Each adversary is assigned a single *target* vertex to *observe* and *attack*. The ta*rget vertices* are not known by the agents in advance, though they may be discovered as attacks are thwarted. This problem is a version of the multi-agent patrol problem (Machado, Ramalho, Zucker, & Drogoul, 2002).

In discrete timesteps, each agent and adversary can perform one action. An agent can either transition (or continue to transition) from one vertex along an edge to an adjacent vertex or remain on its current vertex. An adversary can either attack (or continue to attack) its target vertex, or not attack. Whether an adversary attacks or not, it is aware if an agent is occupying its target vertex.

Each edge of the graph has an integer valued weight value w that denotes how many timesteps it will take an agent to transition from one of its endpoints to the other. While transitioning, the agent is occupying the edge and cannot make any further actions until it reaches the adjacent vertex a total of w timesteps later.

An adversary attacks its target vertex by occupying it for K consecutive timesteps, where fixed integer K, known as the *attack interval*, is known to all agents and adversaries (Basilico, Gatti, & Amigoni, 2009). If an adversary's target is visited by an agent during an attack, the attack is thwarted. On the other hand, if the adversary occupies its target vertex for K timesteps without being disrupted by the arrival of an agent, the attack is successful. Once an adversary begins an attack, it must occupy the vertex until its attack is either thwarted or successful.

When an attack by an adversary is thwarted by the arrival of an agent to that vertex, the agent then knows that the vertex is an adversary's target (i.e., subject to attack). The set of target vertices known by the agents is referred to as the *critical vertices*. Note that if an attack by an adversary on a target vertex is successful, the agents are unaware that the attacked vertex is a target and thus the vertex will not be added to the critical vertices set. Such a vertex is considered compromised (i.e., the adversary has won) and is no longer subject to subsequent attack. The critical vertices are available to all agents at the

timestep in which they are discovered. Adversaries do not have knowledge of the critical vertices and share no information; they operate independently from each other.

When an attack is thwarted, its adversary 'retreats' but may attack this vertex (its target vertex) subsequently. An agent never knows if it has discovered all the target vertices and so must continue to patrol no matter how many critical vertices have been discovered. An agent's strategy is a blend of protecting critical vertices and exploring the graph for other vertices that may be subject to attack.

To decide what vertex to move to next, agents use a service interface, the *agent API,* from which it can obtain information in order to make its decision. The agents use this information to implement their strategies; they have no access to any other information, including the graph topology. The functions of the agent API are:

- current_timestep() – Returns the number of timesteps that have elapsed.
- attack_interval() – Returns the attack interval, *K*.
- incident_edges() – Returns the set of edges incident to the vertex that the agent currently occupies. This is an opaque and unique integer and does not provide any other information, such as the destination vertex. However, it does identify the edge direction from the current vertex to the destination vertex. Each edge identifier has a *reverse* method on it to give the agent another edge identifier which will identify the same edge from the destination vertex to the current vertex. The result is that each edge has two identifiers, one for each direction. Agents can, under some strategies, learn graph topology as they move about the graph.

- critical_vertices() – Returns the set of vertices known to be critical (i.e. subject to attack). Note that if the agent's current vertex is under attack and was just discovered to be critical, it will be returned in this set as well.
- discovered_critical() – Returns true if the current vertex is critical for the first time, meaning that the agent thwarted an attack on it before any other agent did.
- current_vertex() – Returns the vertex that the agent currently occupies. This is an opaque and unique integer and provides no further information about the vertex.
- dist_to_critical_vertex(e, c) – Returns the shortest distance from the current vertex to critical vertex *c* along paths starting from edge *e*, where *e* is incident to the current vertex.

There is an additional convenience function, best_dist_to_critical_vertex(c), which takes as input a critical vertex. Its purpose is to inform the agent which incident edge of its current vertex will get the agent to that critical vertex the fastest. This function only uses the previously mentioned API functions to ease agent development for this common use case. Internally, it calls dist_to_critical_vertex(e,c) for each incident edge and selects the edge with the shortest distance, returing it as part of a 2-tuple that also contains the distance.

Each agent implements a *decide* method that returns its action on every timestep when it occupies a vertex (i.e. not transitioning on an edge). All agents execute the same strategy but have independent state. The method's single input parameter is a reference to the agent API and its output is an optional edge to indicate movement to an adjacent edge

or not. If the agent strategy decides to stay at its current vertex, it simply does not return an edge. No other information is given to the agent's strategy (e.g. the graph topology, the states of other agents, or which strategy the adversaries are using).

Like agents, each adversary implements a *decide* method. This method has a boolean input indicating whether the target vertex is occupied by an agent and outputs the adversary's action of whether to attack. The method is invoked on every timestep when it is not attacking. All adversaries execute the same strategy but have independent state. If an adversary decides to attack, the method is not invoked again until the attack has succeeded or been thwarted. The inputs to the *decide* method calls are the only information the adveraries receive, they do not share any information with each other and they are not aware of which strategy the agents are using.

Agents and adversaries interact within a single *run*, which has an initial start state that defines it and then iterates over discrete timesteps where each agent and adversary may choose an action, as described in the above paragraphs. The start state is composed of the following constants: Graph topology and edge weights, attack interval K, number of agents and their strategy, number of adversaries and their strategy, initial agent locations, and the assignment of a target vertex for each adversary. All agents in each run adopt the same strategy, as do the adversaries. During the run, the conditions specified in the initial state do not change except for the the agent locations, critical vertices, and the states of individual agents and adversaries.

All adversaries will adopt one of three strategies identified by (Sak, Wainer, & Goldenstein, 2008): *random*, *waiting*, and *statistical*. Under the random strategy, an adversary attacks its target vertex at random. Under the waiting strategy, an adversary

observes its target vertex and attacks at the timestep after an agent has left the vertex. This implies that any target vertex left unvisited for *K* timesteps will always be compromised. Lastly, under the statistical strategy, an adversary observes its target vertex over time to construct a statistical correlation of how long after an agent leaves a target vertex and an agent arrives at that same target vertex. The correlation yields a probability that the target vertex will remain unvisited for *K* timesteps based on the observed history. When the adversary determines that the probability exceeds a certain threshold value under a minimum predicted statistical error, it initiates the attack.

Agent strategies compete with an adversary strategy with the goal of minimizing successful adversary attacks. Therefore, agent strategies must make their best effort to protect the critical vertices while also patrolling the graph to protect the unknown target vertices that may reside anywhere in the graph. To effectively patrol a graph with a limited number of agents against adversaries that are attempting to attack an unknown subset of vertices, the strategy employed by the agent is critical and worthy of research. Agent strategies that simply seek to uniformly cover the entire graph become susceptible to attack by adversaries that can predict their future movements throughout the graph. Therefore, agents should attempt to determine which vertices the adversaries are attempting to attack and then adjust their movement to protect them. These two opposing goals require agent strategies to strike a balance between protection and patrol. The experiment ends when every target vertex either has been compromised or has been discovered (i.e., added to the critical set).

Dissertation Goal

The goal was to create new heuristic agent strategies to specifically counter each of the three identified adversary strategies. Each of the agent strategies were designed to perform well against only one of the adversary strategies yet will be tested against all the adversary strategies. Additional agent strategies were designed to perform well against all three adversary strategies. This goal addressed the problem by discovering new methods to counter adversaries and minimize successful attacks by adversaries of differing levels of complexity.

Research Questions

The first research question was how to develop effective agent strategies that specifically target one of the three adversarial strategies. Notably, results from this work on targeted strategies will inform the design of a strategy effective against all three adversarial strategies.

The second research question was whether general agent strategies can be developed using ideas from existing strategies for solving related problems as well as new ideas. A general strategy would be one that performs equally well against any of the three adversaries. In other words, could a general agent strategy be designed that can effectively counter all three agent strategies.

The third research question was how well the agent strategies perform when measured against each other, including new general strategies against non-general strategies. The measures for comparison were based their ability to protect target vertices while having the same constant computational complexity. Specifically, the general strategies have the same level of resource requirements as the non-general strategies.

The fourth research question was to determine under what conditions the new agent strategies perform better. In what ways do such conditions as agent density, graph size, graph connectedness, and edge lengths affect performance? Of interest was how the conditions influence the effectiveness of the new agent strategies.

Relevance and Significance

The problem of agents interacting on a graph was originally formulated in (Parsons, 1976) as agents searching for other "lost" agents somewhere on a graph; this problem was termed pursuit-evasion. Later work determined how to calculate the minimum number of agents required to find and capture an adversary on a graph (Megiddo, Hakimi, Garey, Johnson, & Papadimitriou, 1988). The problem of agents defending against adversaries that can observe the agents to predict their movements and therefore exploit their predictability to successfully attack was explored and found that stochastic agent movements reduce the adversaries' success (Grace & Baillieul, 2005).

The problem of multi-agent patrolling was studied, resulting in a classification of many different variations of agent architectures, evaluation criteria, and experimental scenarios (Machado, Ramalho, Zucker, & Drogoul, 2002). However, the evaluation criteria were centered on a simple reduction of time that vertices are left unvisited; they did not consider the potential abilities of an adversary to predict and exploit predictable agent movements. An extension to that problem formulation took into consideration how an agent will perform against adversaries that can observe and exploit the agent's decisions on vertex movement while patrolling (Sak, Wainer, & Goldenstein, 2008). This extension introduced a new variation of the problem, termed the probabilistic patrolling problem, by the creation of two new types of adversary strategies: waiting and statistical.

The multi-agent patrol problem was identified as a research topic and analyzed by (Machado, Ramalho, Zucker, & Drogoul, 2002). They formulated the problem, identified many types of agent strategies to solve the problem, and created the software needed to conduct experiments and collect empirical data. The problem was structured and simplified with an unweighted directed graph that agents move about on in discrete timesteps, where each vertex is equally important regarding patrol frequency. The criteria chosen to evaluate the performance of the different strategies were based on the concept of idleness and exploration time. Idleness is the amount of time a vertex is left unvisited by an agent. The three types of idleness of concern were the *instantaneous graph idleness*, the *graph idleness*, and the *worst idleness*. All three are defined in (Machado, Ramalho, Zucker, & Drogoul, 2002, p. 157) and their explanation follows: The instantaneous graph idleness is "the average instantaneous idleness of all nodes in a given cycle", where cycle is synonymous with timestep; the graph idleness is "the average instantaneous graph idleness over n-cycle simulation"; and the worst idleness is "the biggest value of instantaneous node idleness occurred during the whole simulation". Exploration time is "the number of cycles necessary to the agents to visit, at least once, all nodes of the graph" (Machado, Ramalho, Zucker, & Drogoul, 2002, pp. 157-158).

Multi-agent patrol strategies that can solve the problem were identified by the definition of four criteria of the strategy: Reactive or cognitive, communication type, how the next node of traveling to is chosen, and a coordination strategy (Machado, Ramalho, Zucker, & Drogoul, 2002). Reactive agents make decisions based only upon information available from their current timestep and location while cognitive agents pursue a goal that is followed for multiple timesteps. Communication between agents can occur in three

types of ways: flags, blackboard, and messages. There are four categories of how the next node is chosen by an agent strategy, which are based on the two dimensions of the *field of vision* and *choice method*. The two fields of vision are local and global while the two choice methods are random or heuristic, resulting in the four categories of local-random, local-heuristic, global-random, and global heuristic. The last criteria of coordination strategy determine if the agent behavior results from some central coordination mechanism or is emergent (no central coordination). By combining the various criteria, we can see that there are $2 \times 3 \times 4 \times 2 = 48$ combinations, although only seven are actually named and examined: Random Reactive, Conscientious Reactive, Reactive with Flags, Conscientious Cognitive, Blackboard Cognitive, Random Coordinator, and Idleness Coordinator (Machado, Ramalho, Zucker, & Drogoul, 2002, p. 158).

For communication types, flag-based communication occurs by altering the environment (e.g. writing information to and reading information from the graph's vertices or edges). For this communication type, agents have access to information stored at their current location. Blackboard-based communications allow agents to read and write information to a globally available data store. Lastly, with message-based communication, agents pass information to each other directly, there is no global or graph-based storage of information (Machado, Ramalho, Zucker, & Drogoul, 2002, p. 158).

Through experimental results, the seven strategies are classified into three groups: random group, non-coordinated group, and top group (Machado, Ramalho, Zucker, & Drogoul, 2002, p. 168). The random group consists of two of the strategies that patrolled in a random fashion (Random Reactive and Random Coordinator), which performed the

worst. The non-coordinated group consists of different two strategies where the agents worked together in an emergent manner (Reactive with Flags and Blackboard Cognitive), which performed better than the random group. Finally, the highest performing group consists of the three remaining strategies: Conscientious Reactive, Conscientious Cognitive, and Idleness Coordinator.

The best performing group consists of a mix of the four criteria: random and cognitive, different communication types, global and local information, and both emergent and centralized coordination. It is important to note that as the number of agents increase, the performance of all strategies tends to converge. Additionally, the architectures of the random and non-coordinated group can outperform the top group when they contain more agents. In other words, the higher number of agents can offset the lack of coordination and sophistication the better performing groups possess.

Follow-on work expanded the experimental methodology and knowledge of the performance of difference architecture by creating new graph topologies and new agent strategies that outperform the previous ones by using reinforcement learning and agent negotiation or bidding (Almeida, et al., 2004). A theoretical analysis of multi-agent patrol strategies found that a theoretical optimal strategy can be used as a tool for analyzing actual strategies in differing classes of graph topologies (Chevaleyre, 2004). Agent strategies can be further separated into the two different classes of cyclic and partition-based, meaning whether the agents are responsible for a sub-graph or share the entire graph. The theoretical optimal performance of both classes was proven. Later work showed that cyclic strategies can perform just as well as partitioning strategies unless a

graph topology contains one or more “tunnels” of vertices (Chevaleyre, Sempe, & Ramalho, 2004).

The problem of using agents on a graph against adversaries was specified in (Hespanha, Kim, & Sastry, 1999), which identified a greedy agent strategy which pursued adversaries on the graph. The strategy was probabilistic in that the agents chose the next vertex to move to increase the probability of thwarting an attack. Further work extended that simple strategy to one where the agents cooperate via global communication in order to optimize their movements on the graph (actually a grid) to increase the efficiency at finding adversaries among the vertices; the agents make individual local decisions that increase the global goal of maximizing the number of adversaries found (Flint, Polycarpou, & Fernandez-Gaucherand, 2002). Work on agent strategies to predict adversary attacks and thwart them using cooperating agents found that modeling adversary behavior using Markov chains is possible (Subramanian & Cruz, 2003). Further work showed how a Markov Decision Process approach using reinforcement learning can enable agents to adapt to adversary behavior as well as respond to changing adversary behavior (Santana, Ramalho, Corruble, & Ratitch, 2004).

The problem of adversaries in the multi-agent patrol problem was further narrowed and analyzed to measure the effects of differing agent and adversary strategies when pitted against each other (Sak, Wainer, & Goldenstein, 2008). It was found that when an adversary can model the agent behavior, unpredictability is necessary to thwart their attacks; this is quite different from previous non-adversarial approaches to the multi-agent patrol problem, where higher regularity results in higher performance because it minimizes the interval at which any vertex is left unvisited. Higher regularity is sufficient

when the adversary is targeting all vertices with equal importance, and the agent considers all vertices to be equally worth protecting. However, agent regularity is a detriment if the adversary is formulating the optimal time to attack because the adversary can easily determine when an agent will not thwart an attack. Therefore, agent unpredictability is important because it reduces an adversary's ability to formulate a successful strategy from observations and predictions on agent behavior. Their problem formulation heavily influenced the problem formulation of this paper. Their identification of three classes of adversary strategies (random, waiting, and statistical) represents a whole range of possible intruder strategies to conduct experiments with.

The random strategy simply chooses when to begin an attack at random. The waiting strategy observes when agents leave a vertex and attack immediately after. The statistical strategy is the most complicated; it observes the timesteps that an agent visits its target vertex and takes note of the intervals between each visit. Every time an agent leaves the target vertex, the adversary looks at previous intervals and calculates the probability that an agent will return to that vertex at least K timesteps later. If the probability of not returning is above a minimum threshold, the adversary will initiate the attack. Also, their research revealed exactly which types of agent strategies perform best against each of the adversary strategies under a wide variety of experimental variables such as the number of agents and K.

Similar work which focuses on perimeter patrol rather than area patrol has also resulted in many contributions that apply to the problem of this paper. Examples of this are agent strategies with adjustable amounts of non-determinism to maximize patrol efficiency while maintaining a high probably of thwarting attacks (Agmon, Kraus, &

Kaminka, 2008) (Agmon, Sadov, Kaminka, & Kraus, 2008), dealing with malfunctioning agents and uncertainty of information on the adversary (Agmon, Kraus, & Kaminka, 2009) (Agmon, Kraus, Kaminka, & Sadov, 2009), expanding the types of events in the experiment beyond an attack by an adversary as a boolean event to that based on the time taken for the agents to detect the attack (Agmon, 2010), and finally combining the above using a Markov model to create perimeter patrol agents under a wide variety of agent correctness of behavior, sensing of adversary actions, and perimeter topologies (Agmon, Kaminka, & Kraus, 2011).

Game-theoretic approaches to the single-agent patrol problem with adversaries has resulted in strong mathematical models for agent strategies where the agent can solve the problem by reducing it to a hierarchy of sub-problems within basic linear, ring, and star graph topologies (Amigoni, Gatti, & Ippedico, 2008). Follow-on work created a more general mathematical model and improved the ability for agents to respond to changing topologies by sensing adversaries beyond those in directly adjacent vertices (Amigoni, Basilico, & Gatti, 2009). Later work introduced uncertainties in the agent's ability to sense adversaries in vertices beyond those that are immediately adjacent and introduced a strategy that stochastically chooses a non-direct path to adversaries so as to make it more difficult for adversaries to model and predict agent movement, all with a computationally efficient algorithm (Basilico, Gatti, & Rossi, 2009) (Basilico, Gatti, Rossi, Ceppi, & Amigoni, 2009).

A deterministic algorithm for thwarting attacks by adversaries based upon the concept of deadlines for vertex visitation (similar to *K*, but different for each vertex), where the agent must visit each vertex often enough to make a successful attack

impossible, was formulated as a Constraint Satisfaction Problem (CSP) (Basilico, Gatti, & Amigoni, 2009).

The best agent strategy for a random intruder is one where the agents follow a TSP path with equal intervals and traveling in the same direction (Sak, Wainer, & Goldenstein, 2008). However, calculating the TSP path with large graphs is intractable, so some heuristic is necessary. A novel heuristic was created based on Newton's law of gravitation, where each vertex has a mass that increases the longer it is left unvisited by an agent, which performs well relative to other TSP heuristics as the number of agents increase (Sampaio, Ramalho, & Tedesco, 2010). Another heuristic for solving TSP is to use *ant colony optimization*, where agents travel within the graph and deposit pheromones on the edges they travel on as a form of communication with each other. The amount of pheromones on an edge indicates to the other agents how much time has passed since an edge was last traversed. When these agents arrive at a vertex, it senses the amount of pheromones at each incident edge and greedily chooses the edge that has not been visited for the longest amount of time. This results in an emergent behavior where approximate solutions to a TSP of a graph are gradually achieved as the ants move about the graph (Dorigo & Gambardella, 1997).

For agents patrolling a graph with critical vertices of weighted importance, an agent strategy of calculating the probability of such vertices being successfully attacked (the risk) at each timestep, where the probability increases the longer an agent does not visit a vertex, has resulted in two well performing two-phase based heuristics (Park, Kim, & Jeong, 2012). In this two-phased strategy, the first phase calculates the agent paths, taking into consideration the initial risks of all critical vertices. In the second phase,

during subsequent timesteps the agent will calculate alternate paths using the current calculated risks and switch to the path that reduces the risk if one exists. Further work with the weighted importance of critical vertices introduced an agent communication capability of passing messages to each other when they are directly adjacent, where a single agent is identified as a leader that coordinates the behavior of the others through such message passing (Pasqualetti, Durham, & Bullo, 2012).

Barriers and Limitations

A limitation of this study was that only three adversarial strategies and six graphs were used during experimentation. The adversaries were identified and used previously by (Sak, Wainer, & Goldenstein, 2008); they represent three classes of complexity and sophistication for adversary strategies. The simplest class is based on random behavior, a slightly more sophisticated class is the waiting strategy, and the most sophisticated is the statistical strategy. The six graphs also come from previous research (Almeida, et al., 2004) and represent several classes of problem domains where the graphs take the form of rings, corridors, islands, grids, and otherwise complex environments. The implementation of the strategies is detailed in the Experimental Design section of the chapter. The graphs are shown in Figure 15 of the same chapter.

A limitation of this problem formulation was that the only communication or coordination among the agents is the sharing of which vertices have been discovered to be critical. Limited communication and decentralized decision making among the agents is a common theme among the current research as it tends to more closely model real world constraints for agents (Flint, Polycarpou, & Fernandez-Gaucherand, 2002), (Iocchi, Marchetti, & Nardi, 2011), (Franco, López-Nicolás, Sagüés, & Llorente, 2015), (Alam, Edwards, Bobadilla, & Shell, 2015), and (Yan & Zhang, 2016). Such constraints apply to agents patrolling the physical world, whether the agents are physical robots or software programs traversing a computer network from computer to computer.

Summary

The multi-agent patrol problem was introduced, and a problem statement was defined in suitable detail to formulate the dissertation goal of creating new heuristic agent

strategies. Four research questions were identified and described to guide the research in the attainment of this goal. Prior research was identified and described to prove the relevance and significance of this proposed research. Finally, the barriers and limitations of this proposed research have been delineated so that the scope of the research is clear.

Chapter 2.

Review of the Literature

This review includes research concerning the multi-agent area-patrol problem with multiple adversaries. The Relevance and Significance section of the Introduction chapter includes many of the same literature reviewed in this section; it is a broad overview of the problem. This chapter, in contrast, is focused on literature that supports the dissertation goal, research questions, and methodology. This section includes multi-agent area-patrol problems that are single-agent or single-adversary oriented, as long there is an adversary. Several domains of this problem will be excluded, however. The related perimeter-patrol version of the full problem is excluded. Research more focused on physical robots in the application of patrol problems is also excluded as they tend to be more concerned with robot sensors and effectors. Lastly, simpler versions of the problem that are patrol related but without adversaries is excluded except for (Chevaleyre, 2004), as such research tends to be focused solely on maximizing graph coverage or reducing vertex idleness.

A theoretical analysis of the multi-agent patrol problem which organized the agent strategies from previous research found some interesting results (Chevaleyre, 2004) (Chevaleyre, Sempe, & Ramalho, 2004). First, it was found that good performance could be achieved with very simple agents who are simply reactive in nature and with minimal or non-existent inter-agent communication. Second, the agents performed better when the graph was partitioned such that an agent only patrolled within an assigned partition. However, this research did not take into consideration the negative effects that adversary strategies have on the agents' patrol effectiveness. For example, an adversary can attack as soon as an agent leaves a vertex and would have an advantage because it could

complete the attack in the interval that the vertex is left unvisited. Instead, the research simply focused on measuring and minimizing the idleness of the vertices during agent patrol, which is only beneficial for thwarting attacks from a simple adversary strategy that does not take into consideration the agents' behavior.

The research of (Paruchuri, Pearce, Tambe, Ordonez, & Kraus, 2007) introduced the multi-agent patrol problem with adversaries. It structured the problem as a Bayesian game with a new heuristic to efficiently find an optimal agent strategy to counter an unknown adversary strategy. The adversary can observe the agents' movements in the graph and determine if there are any vertices that the agent does not visit often during its patrol. If such a vertex is found, the adversary begins the attack. If the agent does not return to that vertex in time, the attack is successful. Once the adversary decides to attack, it must continue to attack for a fixed number of timesteps until it is either successful or has been thwarted. Thus, this research introduced the concept of an adversary that observes the agents and can exploit the agents' actions, necessitating agent strategies that take the adversary's strategy into account. Note that the problem was formulated where the agent was aware of the adversary observing its movements, so their strategies were constantly updated back and forth in response to each other's behavior. This is applicable to the dissertation goal of designing a new agent that can counter any of the three adversary strategies. They modeled the scenario as a Bayesian game which has a well-known problem of being NP-hard to solve because the situation becomes a Stackleberg game (Conitzer & Sandholm, 2006) where the agent and adversary take turns changing their strategy and therefore causing the other's strategy to change in response. Praveen et. all designed a heuristic method for approximately solving this problem in a tractable

manner. However, the research did not identify or measure the results of multiple types of agent and adversary strategies when competing against each other and once an attack is successful or thwarted, the game ends.

Game-theoretic models were applied to the multi-agent patrol problem with adversaries in (Amigoni, Gatti, & Ippedico, 2008), where the graph can change dynamically while the game is underway. Note that this proposed research does not allow the graph to change while the agents and adversaries are running but the approach taken by Amigoni, Gatti, & Ippedico is interesting because it removes the agents' ability to monitor the adversary. However, a major deficiency of the research is that, unlike this proposed research, the game only runs until a single attack by an adversary is successful or thwarted.

The research of (Sak, Wainer, & Goldenstein, 2008) described three distinct adversary strategies of random, waiting, and statistical, which will be used in this research. Several agent strategies were created to counter these adversaries based on total random walking and modified shortest paths of all vertices with random permutations. The strategies could optionally have the agents responsible for separate graph partitions. This work provided a framework for measuring the performance of agents against adversaries using randomly generated graphs. However, those agent strategies required pre-computation (e.g. graph partitioning, k-means clustering, or TSP route calculation) and performed poorly when competing against any other adversary strategy than the one each was designed to counter.

A non-game approach was taken by (Basilico, Gatti, & Amigoni, 2009) where the goal was to design an algorithm that generated a static deterministic patrol route where

each vertex was guaranteed to stay protected from successful attack. To do this, the route must visit each vertex before the attack interval has elapsed since the last visit. They argue that if this goal can be achieved for a graph, it is fundamentally superior to a non-deterministic agent strategy, since it would become irrelevant what strategy the adversary was using. One problem of this approach is that it requires the pre-computation of the route, which would require some type of central coordinator to prepare the agents and place them in their starting positions prior to patrolling. A possible improvement would be to satisfy the same goal with de-centralized and minimally communicating agents. Related follow-on work added an interesting adversary requirement of having to navigate the graph to reach the target vertex and limiting the ability of the adversary to only observe agents near the adversary's current location (Basilico, Gatti, Rossi, Ceppi, & Amigoni, 2009). However, that work did not take into consideration a multi-agent system and was again centrally coordinated.

Generating a nondeterministic patrol path for agents against an adversary that has complete knowledge of the agent's positions can be done linear time (Agmon, Kaminka, & Kraus, 2011). However, their algorithm is limited to a perimeter patrol rather than area patrol, which is much simpler. Their previous work also focused on perimeter patrol only (Agmon, 2010). It may be possible to extend this body of work to adapt it to the area patrolling, which was done by (Alam, Edwards, Bobadilla, & Shell, 2015). They created algorithms using Markov chains in a distributed manner, without requiring centralized coordination or agent communication.

Chapter 3.

Methodology

Introduction

The methodology answers the following four research questions: Can effective agent strategies be developed from existing strategies for solving related problems? Can effective agent strategies be developed for countering any adversary? How do these developed agent strategies perform in relation to each other when measured? How do the problem variables affect agent strategy performance? The following sections of this chapter explain how the research questions on developing agent strategies were answered, how the strategies performed, and an examination of the dependence of performance on program parameters.

Adversary Strategies

Four adversary strategies were created: random, waiting, statistical, and hybrid. At each timestep, these strategies are informed whether an agent occupies their target vertex and uses the information to decide whether to start an attack at that timestep. As stated previously, once an agent initiates an attack, the agent is committed to the attack for K timesteps; an attack is only successful if an agent never occupies the vertex for the entire K timesteps. Note that none of these adversaries were implemented to attack if the target vertex is occupied, as that would be pointless since it would result in the attack being immediately thwarted. The following paragraphs will discuss the design of each adversary strategy, in order from least to most sophisticated.

The random adversary is the simplest and is the only one to not maintain any state between timesteps. At each timestep, the random adversary will initiate an attack with a $1/K$ probability; it will not attack if the vertex is occupied.

The waiting adversary, which is slightly more complex, will only start an attack if the target vertex was occupied at the previous timestep and is currently unoccupied. Therefore, it will not attack during the first timestep or any subsequent time step until an agent occupies the target vertex at least once. This adversary only keeps a single variable in state between timesteps: Was the target vertex occupied or not during the previous timestep?

The statistical adversary is the most complex one in that it attempts to predict the optimal time to attack by observing agent occupation of the target vertex over time, it attacks when it predicts that there is a chance of success. Say a maximal unoccupied interval (MUI) is a longest interval of time that a given vertex is unoccupied by an agent. This adversary will track the total number L of MUIs for its target vertex, and the number C of MUIs of duration at least K timesteps. The adversary attacks if and only if $C \geq \frac{L}{2}$. Intuitively, it attacks if there appears to be at least a 50% chance that the vertex will remain unoccupied for at least K timesteps, based on history.

The hybrid adversary is a combination of the previous three, in which each target vertex is assigned one of those three adversary strategies. The distribution of these adversaries among the target vertices are approximately uniform. This strategy is a more complicated scenario for an agent because an approach that is good for one target vertex will be bad for another (because they could have different adversaries). Therefore, an agent must be more general in nature when countering the hybrid strategy.

Agent Strategies

The research method taken for the first two questions was to develop agent strategies and quantitatively evaluate them with respect to each of the four adversary strategies. Two categories of agent strategies were created: control and covering. The control strategies are relatively simple compared to the covering strategies and serve as a baseline to determine if the more advanced capabilities of the covering strategies result in improvements. When an agent arrives at a vertex, the only decision its strategy must make is which incident edge of its current vertex to choose next. The agent will then begin traversing the edge and after arriving at the endpoint of the edge, will chose another edge, repeating the cycle. The agent cannot remain on the vertex, therefore an edge must be chosen.

Control Agent Strategies

The three control strategies are named random (*control-rnd*), least-recently-used-edge (*control-lue*), and probabilistic-least-recently-used-edge (*control-plue*). The *control-rnd* strategy simply chooses the next edge to move to in a completely random manner.

```
control_rnd_decide = function (api)
  edges = api.incident_edges().toList();
  return edges[random(0, edges.length())];
```

Figure 1 - control_rnd strategy

The *control-lue* strategy chooses the incident edge of its current vertex that the agent has not chosen for the longest amount of time, incident edges that have never been chosen are assumed to have been so for the duration of the entire simulation. If two or more incident edges have been unchosen for the same amount of time, one of them is picked randomly. After choosing an edge, the *control-lue* strategy stores the timestep that the edge was chosen, it will do this for all edges it chooses for the duration of the

simulation. Remember that the API only provides a unique and opaque identifier for each edge and does not provide information about the destination vertex and the identifier is unique to not only the edge but also the edge direction. For example, an edge that goes from vertices *a* and *b* will have two edge identifiers: one for *a* to *b* and another for *b* to *a*.

```
ects = map of edge ids to timesteps when chosen;
control_lue_decide = function (api)
  edge = api.incident_edges()
                    .map(e -> (ects.get(e).or(0),e))
                    .min(((ts1,e1),(ts2,e2)) -> ts1 >= ts2)
                    .map((ts,e) -> e);
  ects.put(edge, api.current_timestep());
  return edge;
```

Figure 2 - control_lue strategy

The *control-plue* control strategy is like the *control-lue* one but differs slightly by choosing the incident edge probabilistically, where the edges that have been unchosen the longest have a higher chance of being chosen than the others. Specifically, the number of timesteps which each incident edge has been unchosen are summed as $S = \sum_{e \in E} ts(e)$, where *E* is the set of incident edge identifiers and *ts* is a function that returns the number of timesteps since the edge was last chosen or the number of timesteps elapsed during the simulation if the edge has never been chosen. The probability of an edge being chosen is the number of timesteps it has been unchosen divided by the *S*: $P(e) = \frac{ts(e)}{S}$.

```
ects = map of edge ids to timesteps when chosen;
control_plue_decide = function (api)
  ts = api.current_timestep();
  edges = api.incident_edges()
              .map(e -> (ts - ects.get(e).or(0),e))
              .flatMap((ts,e) -> n_copies(ts, e))
              .toList();
  edge = edges[random(0, edges.length())];
  ects.put(edge, api.current_timestamp());
  return edge;
```

Figure 3 - control_plue strategy

Covering Agent Strategy

The covering agent strategy differs from the control strategies by: Using a new concept of critical vertex covering, learning the graph topology, and being composed of an edge choosing primitives (ECP) that is specified when the agent is instantiated. Each topic will be described in the following paragraphs and then the pseudocode code of the strategy will be presented and explained.

Critical vertex covering is where each agent takes sole responsibility for the protection of a subset of the critical vertices. When an agent arrives at a vertex, the API provides information on if the agent was the first to thwart an attack on that vertex. If that is the case, the agent will place that vertex into its covered vertex set, which is a subset of the critical vertex set. The agent can deduce if a critical vertex is covered by another agent by checking if it is not in its covered vertex set. Knowing this information, combined with learning the graph topology, allows the agent to also avoid the critical vertices that are covered by other agents.

```
covered_vertices = {}; the critical vertices covered by this agent
uncovered_vertices = {}; the critical vertices covered by other agents
covered_vertex_visit_ts = a map of covered vertices to last visit time
covering_decide = function(api)
  v = api.current_vertex();
  ts = api.current_timestep();
  if api.discovered_critical() then
    covered_vertices.add(v);
  uncovered_vertices = api.critical_vertices() - covered_vertices;
  covered_vertex_visit_ts.put(v, ts);
  edge = some logic to choose an edge
  return edge;
```

Figure 4 - Covered Vertices Discovery

The graph topology is learned by keeping state when travelling on an edge and comparing it to the information after arriving at the destination vertex. Specifically, before travelling on an edge, the current timestep and chosen edge are stored. After arriving at the destination vertex, the current timestep is subtracted from the stored

timestep to calculate the amount of time it takes to traverse that edge. Additionally, the current vertex is noted and the fact that the previously chosen edge has the current vertex as its destination is stored.

```
edge_lengths = map of edge id to edge timestep traversal time lengths
edge_distinations = map of edge id to its destination vertex
prev_edge = null;
prev_ts = null;
prev_valid = false;
covering_decide = function(api)
  if prev_valid then
    edge_lengths.put(prev_edge, api.current_timestep() – prev_ts);
    edge_distinations.put(prev_edge, api.current_vertex());
  edge = some logic to choose an edge
  prev_edge = edge
  prev_ts = api.current_timestep();
  prev_valid = true;
  return edge;
```

Figure 5 - Graph Topology Discovery

An ECP is what the covering agent delegates to in order to choose the next edge to travel on. When delegating to an ECP, the strategy provides it the agent API, the covered vertex set, and the information that the agent has deduced: the set of vertices covered by other agents, the edge lengths, and the edge destinations. There are many possibilities for ECP design and it is not necessarily the case that just one could be written that would perform well. Instead, it is possible to arrange multiple ECPs in a chain, giving each one an opportunity to choose an edge or not. This represents a layering and prioritization of strategies rather than a monolithic design. Thus, there is a special type of compound ECP which is composed of two ECPs and asks the first one to choose an edge and if it does not, asks the second. This compound ECP can be composed of itself recursively to support any number of ECPs in a chain. However, one of the ECPs (ideally the last in the chain) must choose an edge.

```
ecp_choose_1 = primary ecp
ecp_choose_2 = secondary ecp
compound_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    return ecp_choose_1(
      api,
      covered_vertices,
      uncovered_vertices,
      covered_vertex_visit_ts,
      edge_lengths,
      edge_destinations)
    .or(
      ecp_choose_2(
        api,
        covered_vertices,
        uncovered_vertices,
        covered_vertex_visit_ts,
        edge_lengths,
        edge_destinations));
```

Figure 6 - Compound ECP

The covering agent strategy is thus a combination of the aforementioned behavior and an ECP, the pseudocode code for the covering agent strategy is the combination of the previously described pseudocodes.

```
covered_vertices = {}; the critical vertices covered by this agent
uncovered_vertices = {}; the critical vertices covered by other agents
covered_vertex_visit_ts = a map of covered vertex to last visit time
edge_lengths = map of edge ids to edge lengths
edge_distinations = map of edge ids to destination vertex
prev_edge = null;
prev_ts = null;
prev_valid = false;
ecp_choose = the ECP choose function that the agent strategy will use
covering_decide = function(api)
  v = api.current_vertex();
  ts = api.current_vertex();
  if api.discovered_critical then
    covered_vertices.add(v);
  uncovered_vertices = api.critical_vertices() - covered_vertices;
  covered_vertex_visit_ts.put(v, ts);
  if prev_valid then
    edge_lengths.put(prev_edge, ts - prev_ts);
    edge_distinations.put(prev_edge, v);
  edge = ecp_choose(
      api,
      covered_vertices,
      uncovered_vertices,
      covered_vertex_visit_ts,
      edge_lengths,
      edge_destinations);
  prev_edge = edge
  prev_ts = api.current_timestep();
  prev_valid = true;
  return edge;
```

Figure 7 - Covering Agent

There are many types of ECPs, which will be discussed shortly, but first it is important to describe a *decisive* ECP, which will always choose an edge. Because the covering agent strategy delegates to a ECP to choose an edge and an edge must be chosen, at least one ECP in a chain, preferably the last, must choose an edge every time it is asked to do so. There are three types of decisive ECPs, each of which is closely related to the three control agent strategies but take advantage of the information that the covering agent strategy provides them. These ECPs are random (*rnd*), least-recently-used-edge (*lue*), and probabilistic-least-recently-used-edge (*plue*). Each will be described in the following paragraphs, one for each.

The *rnd* ECP will choose an incident edge randomly but will avoid edges whose endpoint is known to be covered by another agent. Thus, it is like the *control-rnd* agent but is more sophisticated by avoiding occupying critical vertices that are already protected by another agent. This gives each agent using the *rnd* ECP more time to protect the vertices that it itself covers. Specifically, this ECP takes the incidence edge set from the agent API and culls all edges that are known to go to the covered vertex of another agent from consideration for choosing. If after culling those edges, there are no edges left to choose (which means all edges go to such vertices), this ECP will switch to a fall back behavior of randomly selecting any of the incident edges. It does this because it is a decisive ECP, which must always choose an edge.

```
rnd_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    all_edges = api.incident_edges();
    culled_edges = all_edges
                .remove(e -> uncovered_vertices
                                .contains(edges_destinations.get(e))
                .toSet();
    edges = if culled_edges.isEmpty() then all_edges else culled_edges;
    return edges[random(0, edges.length())];
```

Figure 8 - rnd ECP

The *lue* ECP will, like the *control-lue* agent strategy, deterministically choose the edge that it has not chosen for the longest time. However, incident edges that the covered agent reports have a destination vertex that is covered by another agent are treated as if they were chosen exactly one timestep in the past, which makes them much less likely to be chosen. If an edge has never been chosen, it is assumed to have been unchosen for the number of timesteps that have elapsed in the simulation. Thus, each edge is given a score equal to how much time has passed since the ECP last chose it. The edge with the highest

score is chosen and if there are multiple such edges, one them is chosen randomly. After choosing an edge, the ECP stores the timestep that the edge was chosen so that another edge will be chosen the next time the agent occupies the same vertex. This agent is decisive and will always choose an edge, even if all edges have a destination vertex that is covered by another agent.

```
ects = map of edges to timesteps when chosen;
lue_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    ts = api.current_timestep();
    edge = api.incident_edges()
                .map(e ->
                  If uncovered_vertices
                              .contains(edge_destinations.get(e)) then
                    return (ts – 1, e);
                  else
                    return (ects.get(e).or(0), e);
                .min((ts1, e1),(ts2,e2)) -> ts1 >= ts2);
    ects.put(edge, ts);
    return edge;
```

Figure 9 - lue ECP

The *plue* ECP, like the *control-plue* agent strategy, chooses incident edges that it has not travelled on for the longest time, but in a probabilistic manner. Each edge is given a probability weight greater than one, which is equal to the number of timesteps that have elapsed since it was last chosen. An exception is that incident edges that the covered agent reports to have a destination vertex that are covered that are known to go to the covered vertex of another agent are given a minimum possible weight of the value one. Conceptually, the edges are chosen by a roulette wheel selection and is implemented as follows. First, an array with a length equal to the sum of all weights is created and each edge is inserted into this array as many times as its weight. Finally, a random number from zero to the size of array is chosen and the edge at that index in the array is chosen.

While this implementation could encounter run time memory allocation errors for very large edge weights, number of vertices, and number of agents (because the array would have large length), such errors did not occur during this experiment because the edge weights and number of vertices are relatively low. Additionally, the ratio of agents to vertices is relatively high. Thus, the array that was created at run time was never large enough to cause an error.

```
ects = map of edges to timesteps when chosen;
plue_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    ts = api.current_timestep();
    edges = api.incident_edges()
                .map(e ->
                  if uncovered_vertices
                            .contains(edge_destinations.get(e)) then
                    return (1, e);
                  else
                    return (ts - ects.get(e).or(0), e));
                .flatMap((w,e) -> n_copies(w, e))
                .toList();
    edge = edges[random(0, edges.length())];
    ects.put(edge, ts);
    return edge;
```

Figure 10 - plue ECP

Three covering agents are defined that are each composed of exactly one of the decisive ECPs: *covering-rnd*, *covering-lue*, and *covering-plue*. These represent the simplest possible covering agent strategies and are complementary to the three control agents *control-rnd*, *control-lue*, and *control-plue* but differ by avoiding the covered vertices of other agents.

In addition to these decisive ECPs, there are four *indecisive* ECPs that do not always choose an edge. These are designed to be used in a chain of ECPs by using one or more compound ECPs that ends with one of the decisive ECPs. These four ECPs are hard-limit

(*hl*), two variants of soft-limit (either vertex or edge focused) named *sl-vertex* and *sl-edge*, and peek-back (*pb*). Each will be described in the following paragraphs.

The *hl* ECP determines if the agent must return to any one of its covered vertices so that it arrives at that vertex before *K* timesteps have elapsed since last occupying it. To do this, it uses the deduced edge weights from the covering agent to heuristically determine how many time steps past this deadline will be remaining for each covered vertex after the agent can arrive at it from its current vertex. The heuristic to calculate the maximum estimated distance for each vertex uses the maximum learned incident edge weight W_I and the maximum global learned edge weight W_G. The maximum learned incident edge weight is calculated as $W_I = \max_{i \in I} w(i)$, where *I* is the set of incident edges and the function *w* returns the integer weight of that edge if deduced by the covering agent or the maximum value for the data type (e.g. 32-bit integer) if unknown. The maximum learned global edge weight is calculated as $W_G = \begin{cases} MAX\_INTEGER, & |E| = 0 \\ \max_{e \in E} v(e), & |E| > 0 \end{cases}$, where *E* is the set of all edges that the covering agent has deduced the weight of and the function v returns the integer weight value for that deduced edge weight; *MAX_INTEGER* is the maximum value for the data type (e.g. 32-bit integer). For each covered vertex, the strategy calculates the deadline timestep $D(v) = p(v) + K$, where *v* is the covered vertex, the function *p* returns the last timestep that the agent occupied a vertex or zero if never occupied (it is never negative), and *K* is the attack interval. The number of timesteps remaining until the deadline is exceed is $R(v) = D(v) - ts$, where *ts* is the value of the current timestep of the simulation; the output is a positive number if there is time remaining until the deadline and negative if the deadline has passed. Next, a heuristic of the estimated maximum number of timesteps needed to reach a covered vertex v for *any*

chosen incident edge of the current vertex is $M(v) = W_I + W_G + d(v)$, where the function d is the agent API call (best_dist_to_critical_vertex) to return the distance of the shortest path to that vertex. The value L for each covered vertex is calculated as $L(v) = R(v) - M(v)$, if L is positive then it is estimated that the agent can reach the covered vertex before the vertex is susceptible to a successful attack. Next, all covered vertices with a value for L that is negative are discarded from further consideration, as they cannot be reached in time. If any covered vertices are remaining, then this ECP will choose the one with the lowest value of $d(v)$ and use the agent API to choose the incident edge that is the first step on that shortest path. Otherwise, this ECP will decline to choose an edge, giving the next ECP in the chain an opportunity to choose one.

```
hl_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    ts = api.current_timestep();
    k = api.attack_interval();
    global_max = edge_lengths.values()
                   .max((length1,lenthg2) -> length1 >= length2)
                   .or(MAX_INTEGER);
    local_max = api.incident_edges()
                   .map(e -> edge_lengths.get(e).or(MAX_INTEGER))
                   .max((length1,length2) -> length1 >= length2);
    distances = covered_vertices
                 .map(v -> (v, api.best_dist_to_critical_vertex(v))
                 .toMap();
    return covered_vertices
             .map(v -> (v, covered_vertex_visit_ts.get(v).or(0)))
             .map((v,t) -> (v, t + k))
             .map((v,d) -> (v, d - ts))
             .map((v,r) -> (
               v,
               r - (local_max + global_max + distances.get(v).dist()))
             .remove((v,l) -> l < 0)
             .map((v,t) -> v)
             .map(v -> (v, distances.get(v).dist())
             .min(((v1,t1),(v2,t2)) -> t1 >= t2))
             .map((v,t) -> v)
             .map(v -> distances.get(v).edge());
```

Figure 11 - hl ECP

The two variants of the soft-limit ECPs differ from the *hl* ECP in that they allow the agent to return to a covered vertex *after K* timesteps have elapsed since last occupying it. These two soft-limit ECPs accomplish this in slightly different ways but neither will ever choose an edge whose other endpoint is a vertex covered by another agent.

The first way, which is *sl-vertex*, calculates a tuple for each vertex covered by the agent. This tuple is composed of the covered vertex, the edge which will get the agent to that vertex the quickest, and the score for that vertex. The score is calculated with the function $S(v) = \frac{(ts+d(v))-p(v)}{K}$, where *v* is a covered vertex of this agent. Intuitively, it is when the agent can arrive at covered vertex *v* subtracted by the last time it visited it, divided by the attack interval. Thus, it is the ratio of how many timesteps will have passed when arriving at that vertex since last visiting it, to the attack interval. Next, all covered vertices with scores less than the value one are removed from further consideration and the edge for the covered vertex that can be arrived at the soonest is chosen. If all covered vertices had scores less than the value one, no edge is chosen.

```
sl_vertex_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    ts = api.current_timestep();
    k = api.attack_interval();
    distances = covered_vertices
                .map(v -> (v, api.best_dist_to_critical_vertex(v))
                .toMap();
    return covered_vertices
             .map(v -> (v, covered_vertex_visit_ts.get(v).or(0)))
             .map((v,t) -> (v,t,distances.get(v)))
             .map((v,t,(e,d)) -> (v,t,e,d,ts + d))
             .map((v,t,e,d,a) -> (e,(a - t) / k), d))
             .remove((e,s,d) -> s < 1)
             .remove((e,s,d) -> uncovered_vertices.contains(e))
             .map((e,s,d) -> (e,d))
             .min((e1,d1),(e2,d2)) -> d1 >= d2)
             .map((e,d) -> e);
```

Figure 12 - sl_vertex ECP

The second way is *sl-edge*, which differs by choosing an edge based on calculating a cost of the incident edges rather than scoring the covered vertices. However, the only incident edges that have a cost calculated and considered are those that the covering agent reports do not have an endpoint that is the covered vertex of another agent (a "safe" incident edge). The cost of each such "safe" edge is calculated by considering each covered vertex as well. First, the amount of "time left" after a "deadline" for each covered vertex, when traveling though one of the edges, is calculated, formally as $L(e,v) = (p(v) + K) - \big(ts + d(e,v)\big)$, where *e* is an incident edge, *v* is a covered vertex, and the function *d* is the API call (dist_to_critical_vertex) that returns how many timesteps it will take to arrive at vertex v through incident edge e. In this formula, the "deadline" for the vertex is $(p(v) + K)$ and the arrival time is $\big(ts + d(e,v)\big)$. By subtracting them, it is determined how many timesteps will be left until the deadline time passes when travelling to that vertex through that edge. Thus, to determine a cost for each

edge, a vector of integers is created for that edge and each covered vertex using $F(e,v) = L(e,v)$ for all v with $L(e,v) > 0$; note that all combinations of incident edge and covered vertex that result in negative time being left at arrival are removed from the vector and are not considered any further. Next, any edge whose $F(e)$ invocation results in an empty vector is removed from consideration as a choice. Lastly, the sum of each element of a vector is calculated to arrive at the edge cost: $C(e) = \sum_{v \in V} \sum_{i \text{ of } F(e,v)} i$, where e is a "safe" incident edge and V is the set of covered vertices. Note that the cost is always a positive number. This ECP then chooses the edge with the lowest cost.

```
sl_edge_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    ts = api.current_timestep();
    k = api.attack_interval();
    distances = covered_vertices
                 .map(v -> (v, api.best_dist_to_critical_vertex(v))
                 .toMap();
    avoid_edges = edge_destinations
                    .remove((e,v) -> uncovered_vertices.contains(e))
                    .map((e,v) -> e)
                    .toSet();
    return api.incident_edges()
            .remove(e -> avoid_edges.contains(e)
            .map(e -> (
              e,
              covered_vertices
               .map(v -> (v, covered_vertex_visit_ts.get(v).or(0)))
               .map((v,t) -> (
                            t + k,
                            ts + api.distance_to_critical_vertex(e,v))
               .map((d,a) -> d - a)
               .remove(time_left -> time_left < 0)
               .toList())
            .remove((e,time_lefts) -> time_lefts.isEmpty())
            .map((e,time_lefts) -> (e,time_lefts.sum())
            .min((e1,s1),(e2,s2)) -> s1 >= s2)
            .map((e,vs) -> e);
```

Figure 13 - sl_edge ECP

The remaining ECP is *pb*, it checks each non-critical vertex it is occupying to determine if returning to it immediately after leaving it will thwart an attack on it. To do this, it randomly chooses an incident edge that does not go to another agent's covered vertex and then immediately returns to that same vertex along the same edge, in reverse. Each vertex is only checked once and when the current vertex has already been checked, this ECP declines to choose an edge. Eventually, all vertices will either be checked or be critical and this primitive will cease choosing edges altogether. It is designed to quickly find attacks in the beginning of the simulation and then stop as soon as possible.

```
vertices_checked = {};
edges_checked = {};
struct ReturnToCheck = {
  return_edge,
  check_vertex
}
return_to_check = null;
checking_vertex = null;
pb_ecp_choose = function(
  api,
  covered_vertices,
  uncovered_vertices,
  covered_vertex_visit_ts,
  edge_lengths,
  edge_destinations)
    v = api.current_vertex();
    chosenEdge = null;
    vertices_checked.addAll(api.critical_vertices());
    if checking_vertex != null then
      if v == checking_vertex then
        vertices_checked.add(checking_vertex);
      checking_vertex = null;
    if return_to_check != null then
     if api.incident_edges().contains(return_to_check.return_edge) then
       chosen_edge = return_to_check.return_edge;
       checking_vertex = return_to_check.check_vertex;
     return_to_check = null;
    if chosen_edge == null then
      if !vertices_checked.contains(v) then
        avoid_edges = edge_destinations
                        .remove((e,v) -> uncovered_vertices.contains(e))
                        .map((e,v) -> e)
                        .toSet();
        chosen_edge = api.incident_edges
                .remove(e -> edges_checked.contains(e))
                .map(e -> (e, edge_destinations.get(e).or(null)))
                .map((e,d) -> (
                     e,
                     d.map(v -> covered_vertices.contains(v).or(false))
                .remove((e,v) -> v == true)
                .remove((e,v) -> avoid_edges.contains(e))
                .map((e,v) -> e)
                .any();
        if chosen_edge != null then
          return_to_check = new ReturnToCheck {
            return_edge = chosen_edge.reversed(),
            check_vertex = v
          };
    if chosen_edge != null then
      edges_checked.add(chosen_edge);

    return chosen_edge;
```

Figure 14 - pb ECP

By combining these ECPs in different ways (but always ending with a decisive ECP), many varying behaviors can be created that interact with adversaries in diverse ways. Many combinations (198 in total) were created with three being just the terminal primitives alone and with different combinations of all others without duplicates, always ending with a terminal primitive. However, only a few were assumed to perform best against the different adversaries, which will be explained in the following paragraphs.

Each of the indecisive ECPs were designed to be effective against different adversaries. The *hl* ECP was designed to be effective at countering the waiting adversary because any critical vertex that remains unvisited for longer than *K* timesteps will definitely be compromised. The two soft-limit ECPs (*sl-vertex* and *sl-edge*) were designed to counter the statistical and random adversaries; where it is not vital to return to vertices within *K* timesteps because it is unlikely that an attack on them will begin as soon as an agent leaves them. The *pb* ECP was also designed to counter the waiting adversary because its goal is to find vertices being attacked immediately after an agent leaves them.

The decisive ECPs were also designed with specific adversary strategies in mind, when used in a chain. The *rnd* ECP is designed to confuse the statistical adversary by being unpredictable. The *lue* and *plue* ECPs were designed to counter the random and waiting strategies.

Because there are so many possible combinations of covering agent strategies with different chains of ECPs, it is not practical to define them all. In fact, it was not entirely known if the ECP combinations that were assumed to be effective against certain adversaries would actually be so. However, some general assumptions are outlined in the

following paragraphs as to what the result will be for certain combinations against the adversaries. Ultimately, it was determined through experimentation of all possibilities as listed in **Error! Reference source not found.** and measuring their effectiveness which c ombinations actually perform best against which adversaries. Furthermore, it was assumed that the top performing combinations of ECPs for the covering agent strategy would outperform the control strategies.

When designing the ECPs, it was assumed that certain combinations would perform best against certain adversary strategies. For the random adversary, the combination of a chain of one of the soft-limit ECPs followed by either the *lue* or *plue* would outperform other combinations. For the waiting strategy, the triple combination of the *hl*, followed by *pb*, and terminating in either the *lue* or *plue* ECPs would outperform other combinations. Finally, for the statistical adversary, one of the soft-limit ECPs followed by the *rnd* ECP was designed to perform the best in comparison to other combinations.

For the third and fourth research question, the performance of the agent strategies will be analyzed based upon the variables of the problem such as graph topology, number of agents and adversaries, and *K.* The exact variables will be described in the next section.

Experimental Design

The experimental design was to run simulations of each agent strategy against adversary strategies under a range of different *scenarios*. Variables of each scenario were: One of six graph topologies as shown in Figure 15 from (Almeida, et al., 2004, p. 480); the ratio of the number of agents to vertices $N_1 \in \{5\%, 10\%, 15\%, 20\%, 25\%\}$ and the ratio of the number of adversaries to vertices $N_2 \in \{5\%, 10\%, 15\%, 20\%, 25\%\}$ from

Portugal & Rocha (2013, p. 330); and the attack interval based on the number of agents and the length of the *approximate* TSP cycle of the graph $K \in \left\{\frac{1}{8} \times \frac{T_{cycle}}{n}, \frac{1}{4} \times \frac{T_{cycle}}{n}, \frac{1}{2} \times \frac{T_{cycle}}{n}, 1 \times \frac{T_{cycle}}{n}, 2 \times \frac{T_{cycle}}{n}\right\}$, where T_{cycle} is the length of the approximate shortest TSP cycle of the graph (factoring in edge weights) and n is the number of agents (Sak, Wainer, & Goldenstein, 2008, p. 130). Thus, there will be $6 \times 5 \times 5 \times 5 = 750$ scenarios.

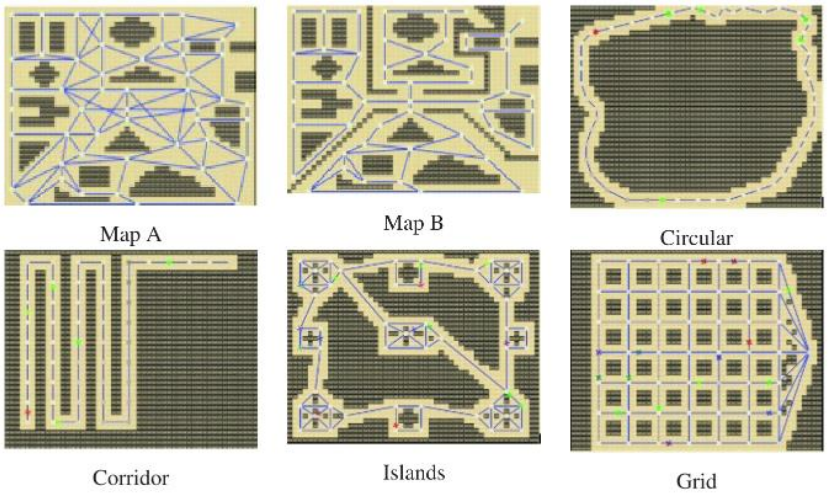

Figure 15 - Graph Topologies

The chosen graphs characterize six classes of environment for the agents. The Circular and Corridor graphs are the simplest and can represent patrolling a perimeter and hallway, respectively. The Grid graph can approximate a warehouse where much of the environment is uniform but with a small but complex area where coordination occurs for humans that work in the warehouse. The Islands graph can represent the computer network of a large corporation with corporate offices, a small country with cities, or the Internet backbones of the entire world connecting continents together. Finally, graphs Map A and Map B represent arbitrarily complex environments with Map B differing from Map A by in the inclusion of barriers to isolate the graph into four areas.

Information on these graphs have been computed and are listed in Table 1 below. Note that approximate TSP was calculated using the Nearest Neighbor Algorithm; finding an optimal TSP path is not critical for this research. The TSP length is given in both the number of vertices and the sum of the edge weights for the path.

Graph	**# Vertices**	**# Edges**	**Approximate TSP Length (# Vertices)**	**Approximate TSP Length (Edge Weights)**
Map A	50	105	63	380
Map B	50	69	73	512
Circular	50	50	51	178
Corridor	49	48	97	392
Islands	50	84	59	332
Grid	50	91	58	353

Table 1 - Graph Details

Each scenario gives rise to multiple *matches* over all combinations of the agent strategies and adversary strategies, where each of the agent strategies played against each of the adversary strategies. A match is composed of X=10 *games*, each of which has different randomly selected agent starting positions and target vertices. Each game runs for $R = 100 \times T_{cycle}$ timesteps (Sak, Wainer, & Goldenstein, 2008, p. 130).

Data Analysis

The effectiveness of agent strategies was calculated from the following measurements, all calculated from experiment outputs from each game. The two types of measurements are raw and calculated. The raw measurements are the number of attacks

and the number of attacks that were thwarted. There are five calculated measurements. The first calculated measurement in the *attack thwarted ratio*, which is the percentage of attacks there were thwarted. The next is *general effectiveness*, which is the percentage of target vertices that were not compromised at all. The *deterrence effectiveness* is the percentage of target vertices where no attack was attempted. The *patrol effectiveness* is the percentage of the target vertices that were discovered to be critical by the agents. Lastly, the *defense effectiveness* is the percentage of initially thwarted critical vertices that were never compromised, which differs from the general effectiveness by considering that some target vertices may never be attacked (i.e., how effective the strategy is at continuing to thwart attacks on a target vertex without it becoming compromised later).

The game measurements were then summarized into a match measurement, which is the average and standard deviation of each of the ten game measurements for a match. With each match measurement are the variables for that match and its enclosing scenario, combined. The graph, number of agents, ratio of agents to vertices, number of adversaries, ratio of adversaries to vertices, and K are from the scenario. The agent strategy and adversary strategy are from the match. The result is a multi-dimensional cube of measurements that were analyzed to determine how the agents perform under varying conditions.

Summary

The methodology to answer the research questions was to develop the adversaries, control agent strategies, and the covering agent strategy along with its ECPs. Experiments were run which evaluate the performance of the agent strategies under many varying

conditions. The design of the agent strategies has been described along with the new concepts of covered vertex covering and ECPs. The experimental design is clearly defined and backed up by prior research. Graphs used by previous research were obtained and evaluated for their properties such as their approximate TSP length and application to real world scenarios.

The analysis of the data relied on the defined measurements of raw data and effectiveness of an agent. The output of an experiment resulted in a n-dimensional cube in data that was analyzed across many different slices to determine when and why agents perform well or not.

Chapter 4.

Results

Introduction

The agents are named in a very particular way in the experiment results to describe their behavior. First, the control agent strategies are named *control_lue, control_plue,* and *control_rnd* for least-recently-used-edge, probabilistic-least recently-edge, and random, respectively. The covering agent strategy names being the prefix *control_* followed by the ordered list of the edge chooser primitive names, separated by the "_" character. For example, *covering_hl_rnd* is the covering agent strategy with the hard limit and random edge chooser primitives, in that order. The names, descriptions, and examples of these edge choosing primitives of the covering agent strategy are listed in Table 2.

Edge Chooser Primitive Name	**Description**	**Terminal?**	**Example**
lue	Least Recently Used Edge	Yes	*covering_lue*
plue	Probabilistic Least Recently Used Edge	Yes	*covering_plue*
rnd	Random	Yes	*covering_rnd*
hl	Hard Limit	No	*covering_hl_rnd*
pb	Peek Back	No	*covering_hl_pb_lue*
sl-vertex	Soft Limit (Vertex Focused)	No	*covering_sl-vertex-plue*

Edge Chooser Primitive Name	Description	Terminal?	Example
sl-edge	Soft Limit (Edge Focused)	No	*covering_sl-edge-rnd*

Table 2 - Covering Agent Edge Chooser Primitive Naming

Control Agent Strategies

The control agent strategies showed clear differences in their performance against the four adversaries, as can be seen in Figure 16. The *control-lue* agent strategy performed much better under the general effectiveness measure than the other two control strategies against the random and waiting adversaries. The *control-rnd* strategy did the best against the statistical adversary and worse overall against the other adversaries. The *control-plue* ranked in between the other two control agent strategies in all cases, which is not surprising considering that it functions like mixture of the deterministic *control-lue* and nondeterministic *control-rnd* strategies. Overall, the *control-lue* agent performed best.

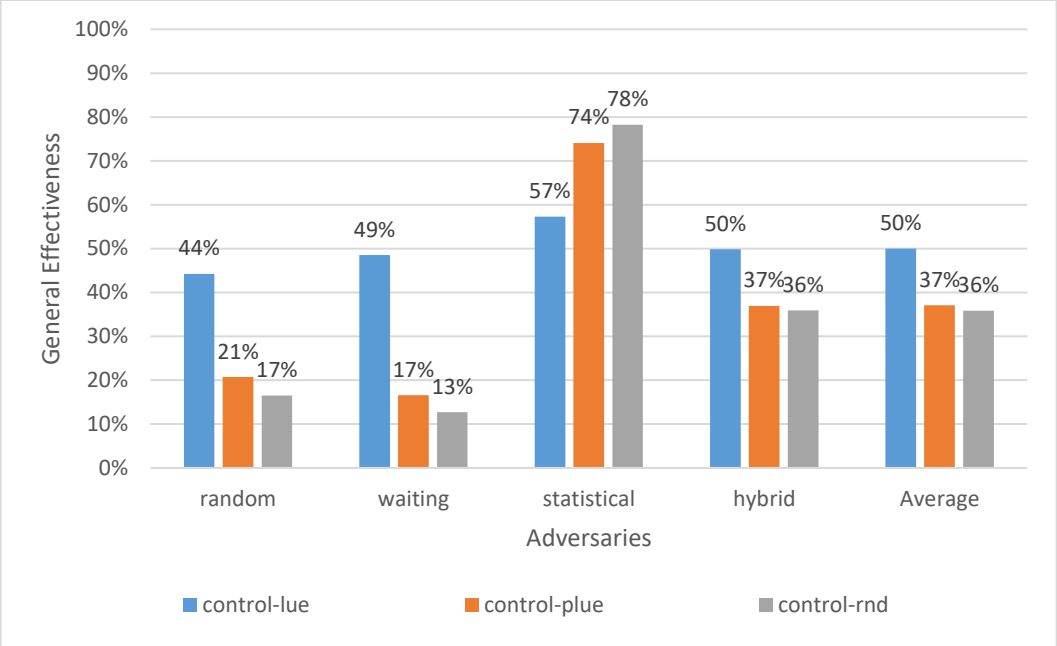

Figure 16 - General Effectiveness of the Control Agent Strategies

Basic Covering Agent Strategies

The basic covering agents are simply the covering agents with a single terminal edge choosing primitive. Their performance is illustrated in Figure 17. Like with the control agents, the least recently used edge variant (*covering-lue*) outperformed the probabilistic least recently used edge (*covering-plue*) and random (*covering-rnd*) variants when going against the random, waiting, and hybrid strategies. Unlike with the control agents, the *covering-plue* agent outperformed the *covering-rnd* when against the statistical adversary.

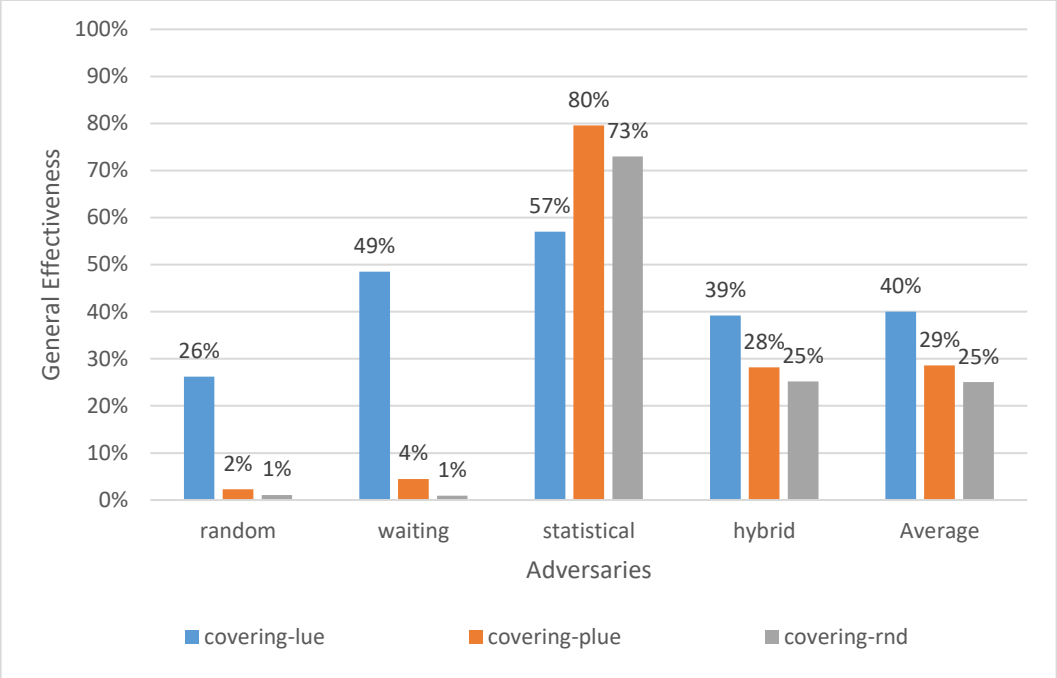

Figure 17 - General Effectiveness of the Basic Covering Agents

The basic covering agents did not perform as well as their control agent counterparts, as can be seen in Figure 18. The additional behaviors that the covering agents add is detrimental in all situations when only a single terminal edge choosing primitive is used.

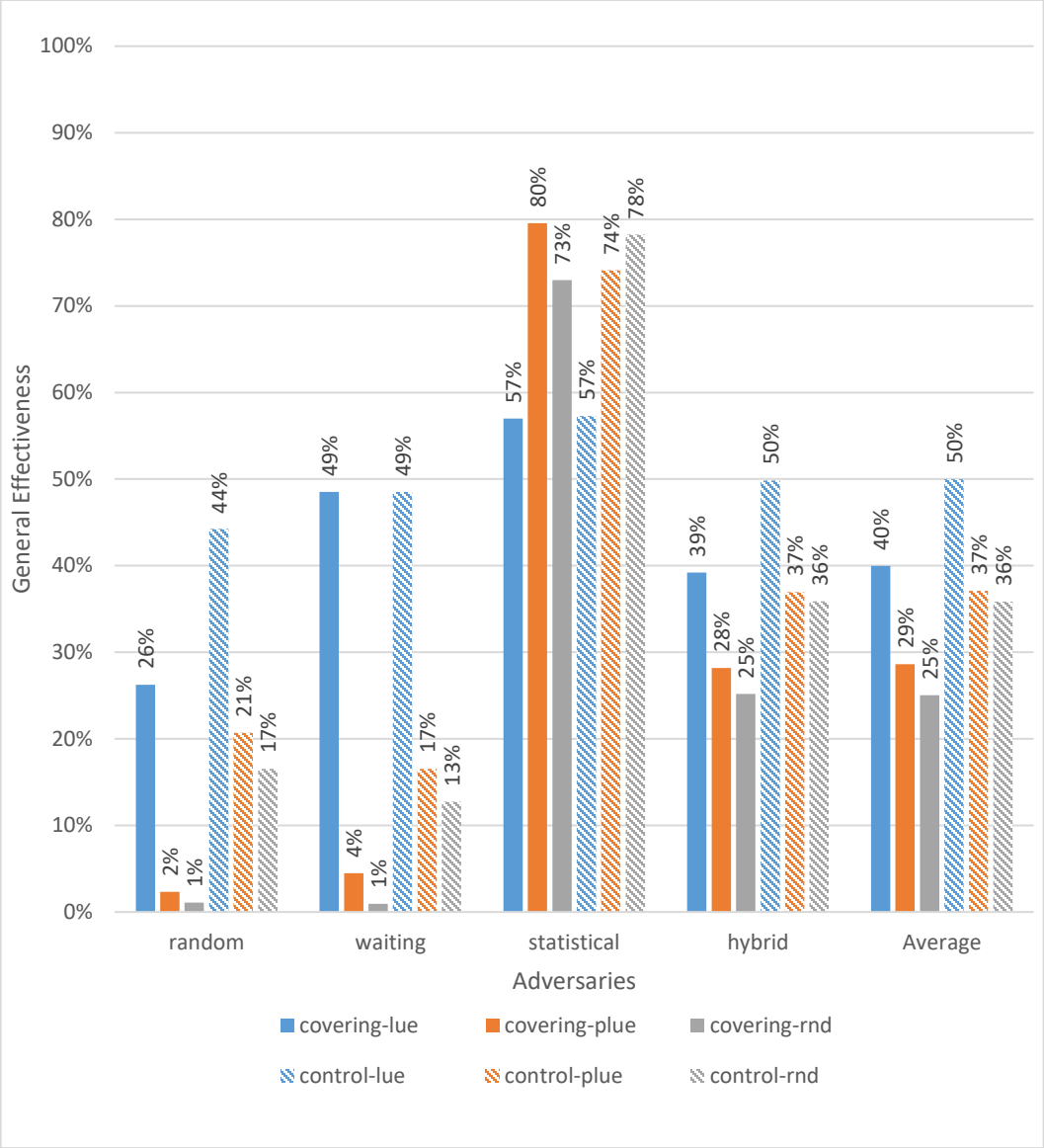

Figure 18 - General Effective of the Basic Covering and Control Agents

Chained Agent Strategies

Overall

The effectiveness measurements of all 198 agent strategies against all adversaries are illustrated in Figure 19, which has the agent strategies on the horizontal axis but are unlabeled. The chart sorts all agents in descending order of their general effectiveness average against all adversaries, left to right. The corresponding defense, patrol, and

deterrence effectiveness measurements are also shown (also the average against all adversaries), showing how they relate to general effectiveness. General effectiveness of all agents ranges from about 25% to about 83%, showing a wide disparity in performance among the agents overall. General effectiveness is inversely correlated with patrol effectiveness such that as patrol effectiveness rises, general effectiveness decreases. Defense effectiveness is somewhat noisy but in general lowers along with general effectiveness. Deterrence effectiveness is stable and not correlated with general effectiveness.

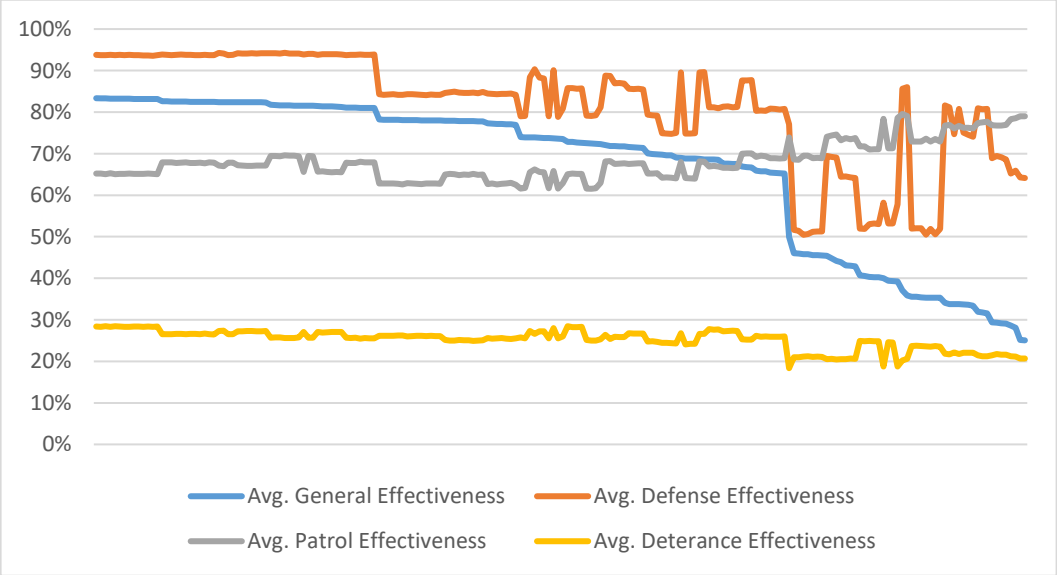

Figure 19 - Effectiveness Range Against All Adversaries

The top performing agents measured by general effectiveness are illustrated in Figure 20. This chart shows the agents who score is within 98% of the maximum general effectiveness score. There are a lot of variants in this list and while it is tempting to just choose the top performing agent as the best, such a strategy does not pick the ultimate essence of what combinations and orderings of edge choosing primitives is optimal. Therefore, a strategy was used to pick out the truly optimal agent by also considering the

number of edge choosing primitives an agent has, in addition to its general effectiveness. To facilitate this selection, a simple strategy of choosing the agent with the highest general effectiveness and the lowest count of edge choosing primitives was taken. The rational for this is that the top performing combinations of ECPs have general effectiveness scores that are very close to each other and the absolute highest performing agents can have a very large number of ECPs in comparison to similarly performing agents with lower numbers of ECPs. However, the larger numbers of ECPs do not necessarily mean that these agents are superior. Instead, their slightly higher performance is a statistical anomaly and does not capture the true essence of a superior performing strategy of an agent with very similar performance but less ECPs. In the case of overall agent performance against all adversaries, the *covering_hl_pb_rnd* combination of three edge choosing primitives was selected and is highlighted in Figure 20. While there are ten other agents that performed better, they all have four or five edge choosing primitives instead of three. Additionally, the general effectiveness of this agent strategy was 82.20%, which is within 99.82% of the maximum general effectiveness of 83.25% for the *covering_hl_sl-edge_pb_rnd* agent. Therefore, it is argued that the *covering_hl_pb_rnd* (explained in further detail in Appendix A) agent represents the true optimal agent strategy, with the chained combination of the Hard-Limit, Peek Back, and Random edge choosing primitives; the *sl-edge* ECP from the *covering_hl_sl-edge_pb_rnd* agent did perform better, but so did the *covering_hl_sl-vertex_pb_rnd* agent. Thus, it is argued that neither the inclusion of the *sl_edge* or *sl_vertex* ECPs are critical for identifying the most appropriate agent in terms of selecting the optimal combination and number of ECPs for an agent.

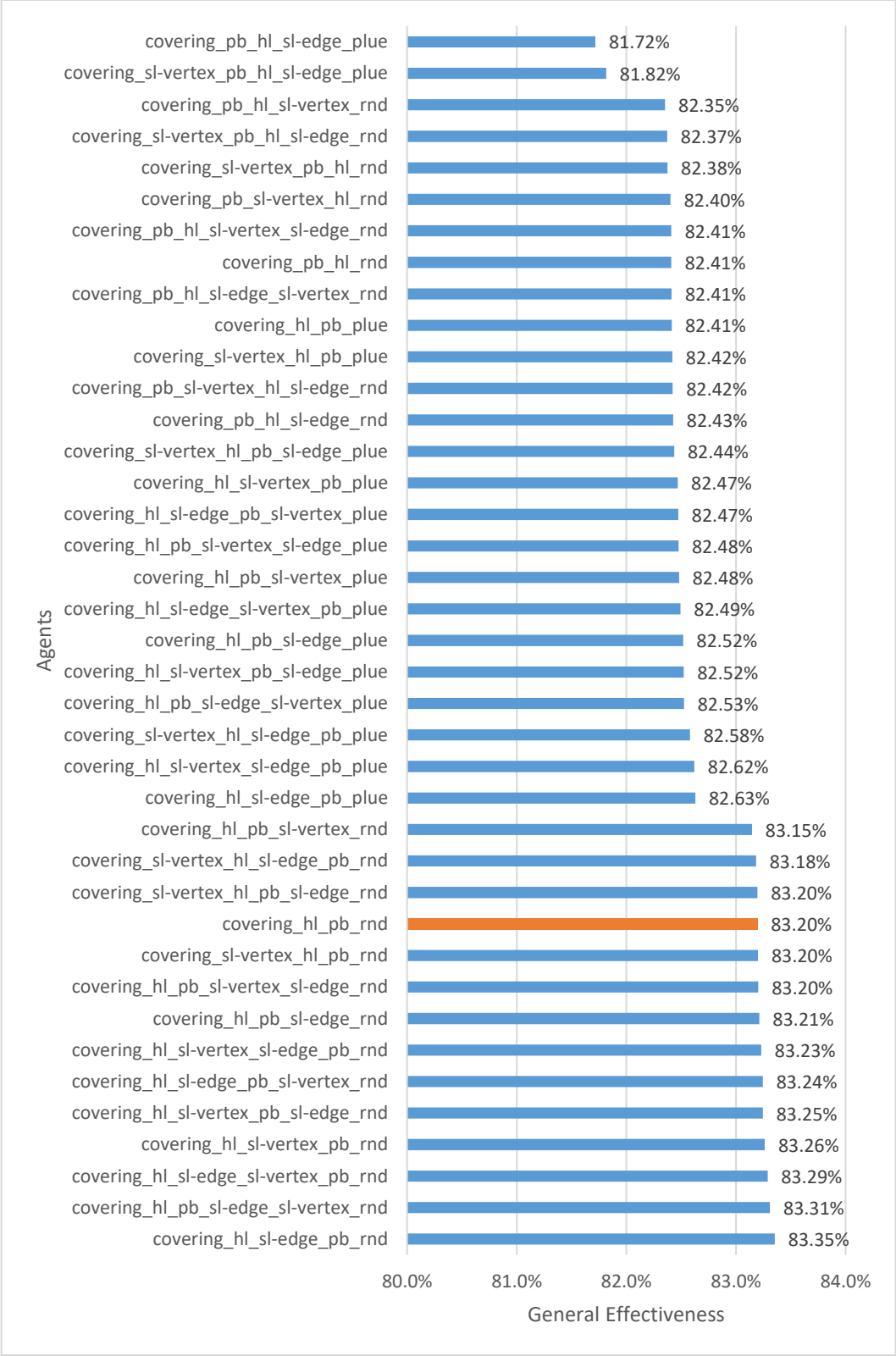

Figure 20 - General Effectiveness Against all Adversaries (Within 98% of Maximum)

Against the Random Adversary Strategy

Against the random adversary, the range of general effectiveness among all the agents varies considerably from about 1% to about 69% (see Figure 21). The deterrence effectiveness is always 0% for all agents because no behavior on the part of the agent will deter the random adversary from attacking. The patrol effectiveness is inversely correlated with general effectiveness because discovering more critical vertices (patrol effectiveness) comes at the cost of not being able protect critical vertices from even a single successful attack (general effectiveness).

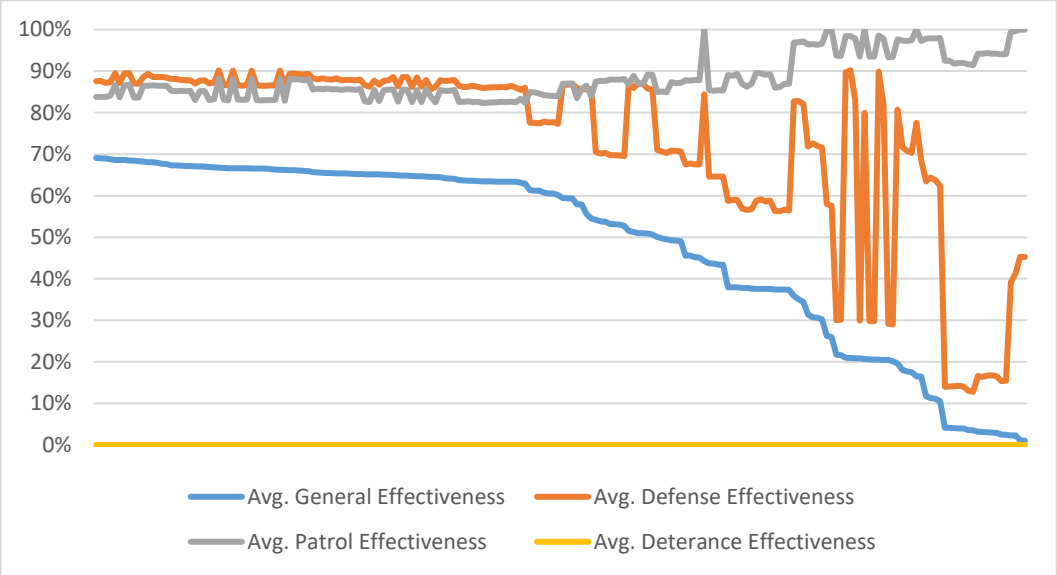

Figure 21 - Effectiveness Range Against the Random Adversary

The top performing agents, as measured by general effectiveness are *covering_hl_lue*, *covering_sl-vertex_hl_sl-edge-lue*, and *covering_sl-vertex_hl_lue* with scores of 69.0%, 69.0%, and 69.1%, respectively, as illustrated in Figure 22. Using the criteria to select the optimal strategy as described in the previous section, the best agent strategy is *covering_hl_lue*, which is a chain of the hard limit and least-recently-used-edge choosing primitives. Even though this agent is not the absolute best as measured, it

is simpler than the top two and within 99.85% of the top performing agent. It is not surprising that this agent performs well because it covers the graph uniformly and at regular intervals unless a critical vertex is in danger of being unvisited for longer than *K* timesteps, in which case it heads directly to that vertex as close as possible to exactly after *K* timesteps have elapsed. The least-recently-used-edge choosing primitive was created to counter the random adversary, so it is encouraging that the results indicate its effectiveness. Note that the simpler *control_lue* and *covering_lue* agents only had a general effectiveness score of 44.24% and 26.23%, respectively, showing that by adding vertex covering and the hard limit edge chooser, the performance of an agent can increase greatly (to 69%).

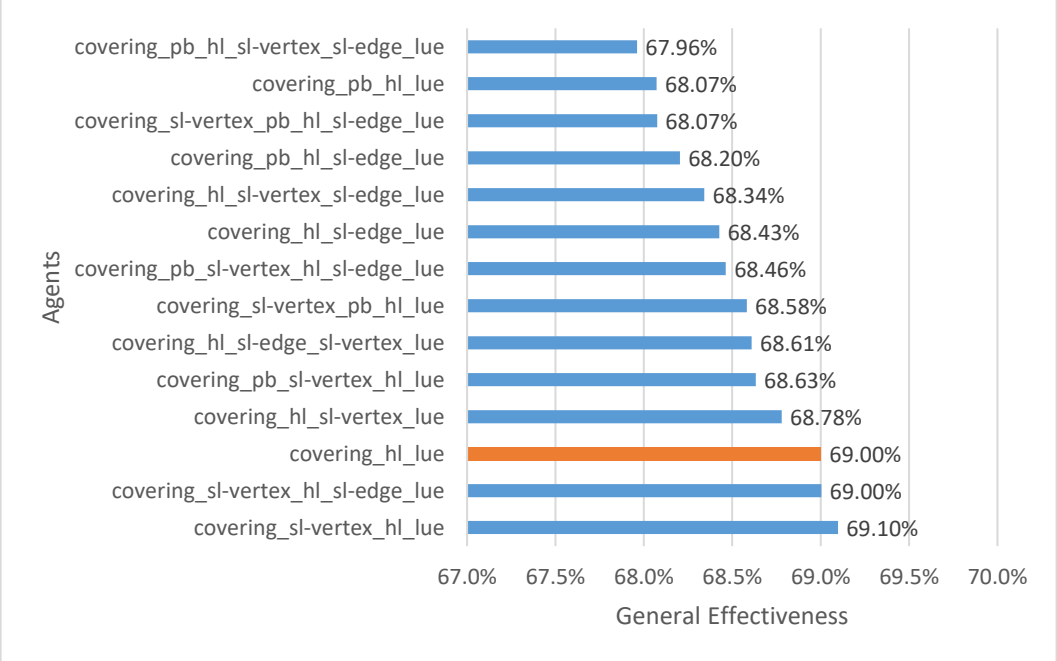

Figure 22 - General Effectiveness Against the Random Adversary (Within 98% of Maximum)

Against the Waiting Adversary Strategy

When going against the waiting adversary, agents have a correlation between general effectiveness and both defense and deterrence effectiveness (see Figure 23). Patrol

effectiveness is inversely correlated with general effectiveness. The general effectiveness measurements for all agents against the waiting adversary has a range of 0.84% to 94.30%. The *control_lue* control strategy performed best against the all the control strategies with a general effectiveness score of 49%. The base covering strategy *covering_lue* performed best against all the basic covering strategies with a general effectiveness score of 49%.

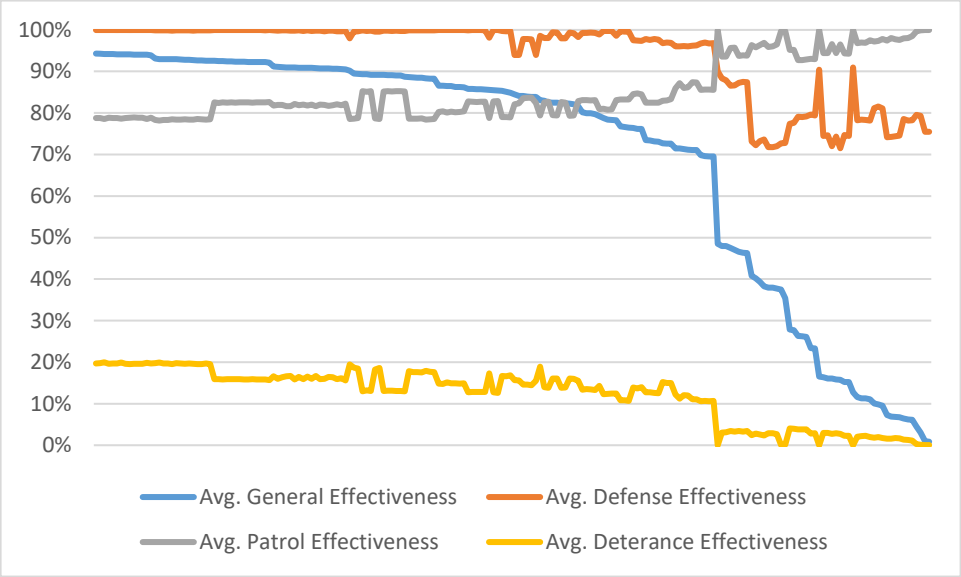

Figure 23 - Effectiveness Range Against the Waiting Adversary

An interesting result is the correlation of general effectiveness to attack count; general effectiveness decreases as the attack count decreases (Figure 24 and Figure 25). Because the waiting adversary only attacks after an agent leaves its target vertex, a low attack count indicates that the agent is not visiting target vertices.

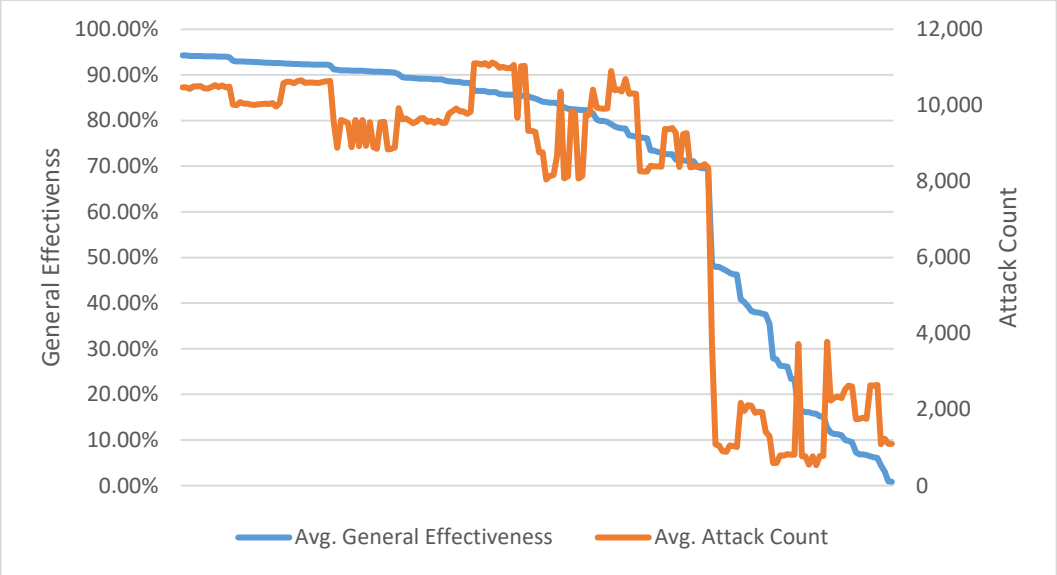

Figure 24 - General Effectiveness and Attack Count against the Waiting Adversary

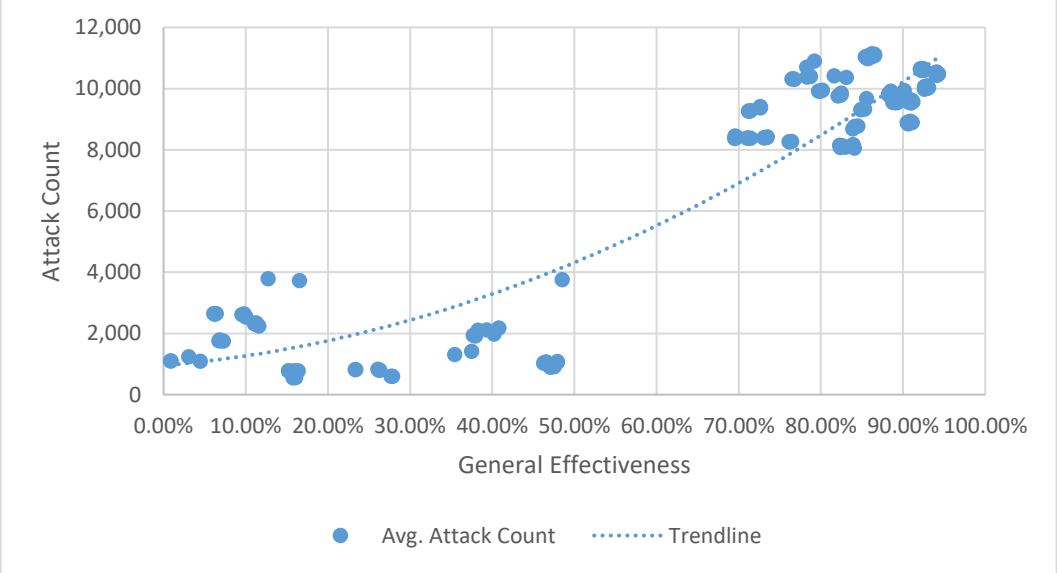

Figure 25- General Effectiveness and Attack Count Scatterplot for the Waiting Adversary

The top performing agents against the waiting adversary are illustrated in Figure 26, of which the optimal agent is *covering_hl_pb_rnd*. This agent is composed of the Hard-Limit, Peek-Back, and Random edge choosers, in that order. This agent has a general effectiveness score of 94.06%, which is within 99.75% of the top score of 94.30%. The Peek-Back edge chooser was designed specifically to counter the waiting adversary, so

its presence in the optimal and all top performing agents is expected and its contribution to countering the waiting adversary is proven.

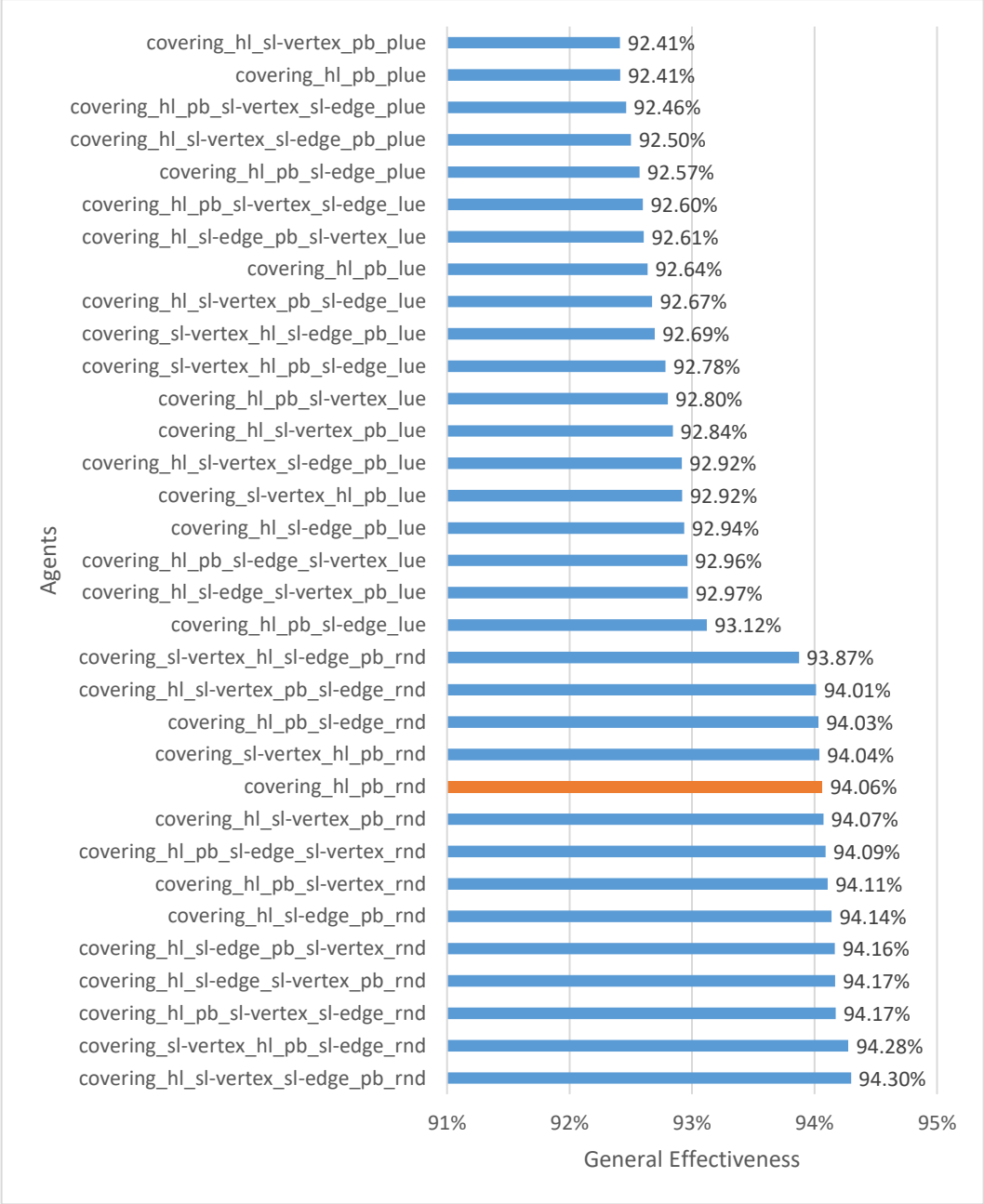

Figure 26 - General Effectiveness Against the Waiting Adversary (Within 98% of Maximum)

Against the Statistical Adversary Strategy

When going against the statistical adversary, the general effectiveness of an agent is correlated with the defense effectiveness. The general effectiveness for all agents has a range of about 57% to about 92.5% (Figure 26). General effectiveness is inversely correlated to attack count (Figure 28 and Figure 29), since the statistical adversary only attacks when it predicts it will be successful, a reduce attack count implies that the adversary is unable to predict success and hence does not attack. Since lower attack counts result in higher general effectiveness scores, an agent performs best by preventing attacks in the first place.

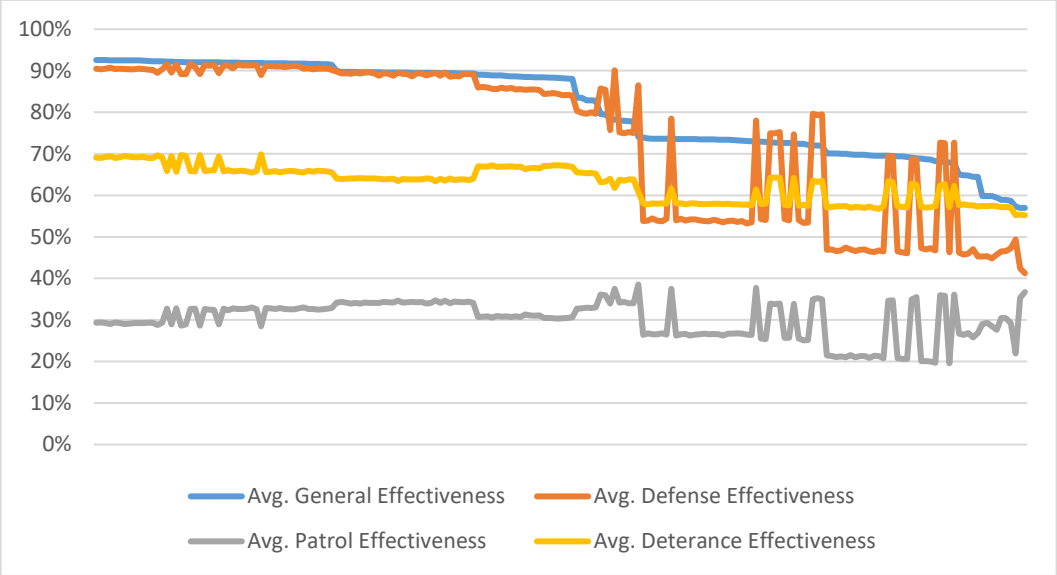

Figure 27 - Effectiveness Range against the Statistical Adversary

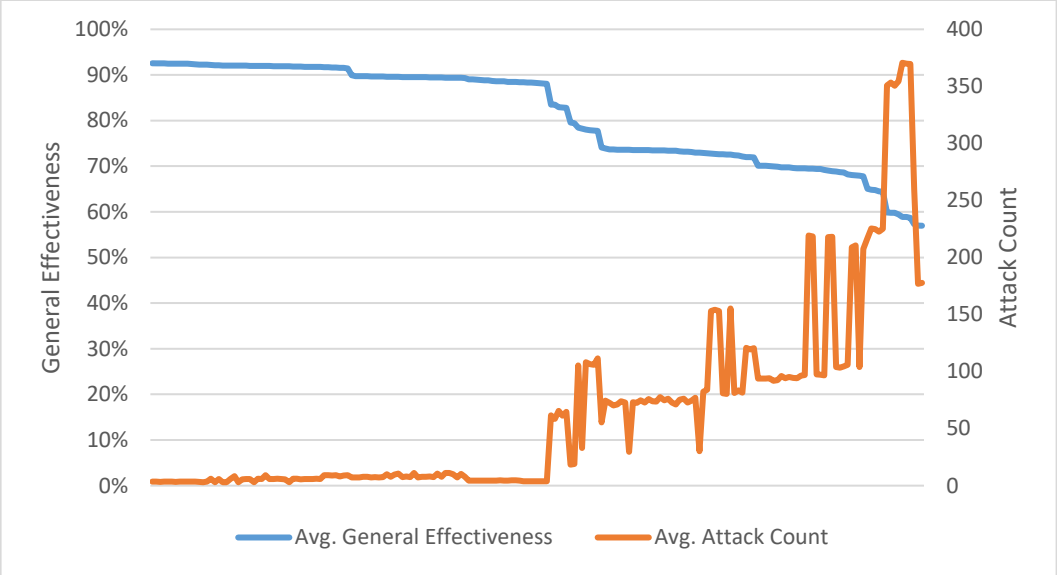

Figure 28 - General Effectiveness and Attack Count against the Statistical Adversary

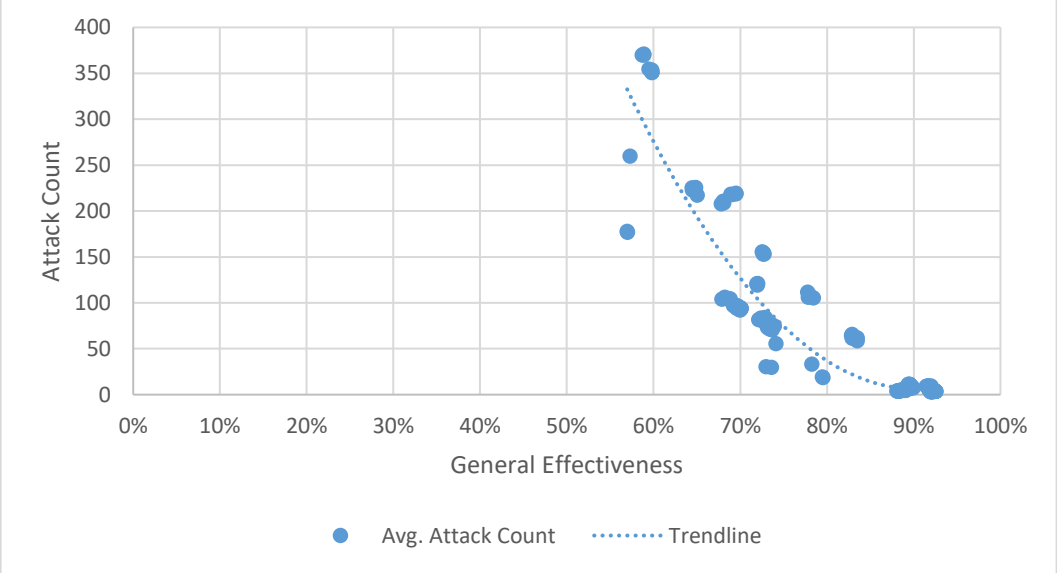

Figure 29- General Effectiveness and Attack Count against the Statistical Adversary (Scatter Plot)

The agents that score the highest general effectiveness scores (within 98% of the maximum score) and illustrated in Figure 30. Of those, the agent *covering_hl_rnd* is the simplest and is within 99.5% of the much more complicated maximum scoring agent. The *covering_rnd* and *control_rnd* agents scored only 79.6% and 78.2%, respectively. The much more predictable agents *covering_lue* and *control_lue* scored 56.99% and

57.29%, respectively, near the bottom of all agents. The only agent which performed worse was *covering_pb_lue* at 56.96%. These low scores are not surprising because these agents have very predictable patterns of movement.

Figure 30 - General Effectiveness Against the Statistical Adversary (Within 98% of Maximum)

Against the Hybrid Adversary Strategy

The general effectiveness scores against the hybrid adversary vary from 25.07% to 83.81 and are inversely correlated with patrol effectiveness (Figure 31). General effectiveness is correlated with the attack count; agents that evoke the adversaries to attack more have higher scores of general effectiveness (Figure 32).

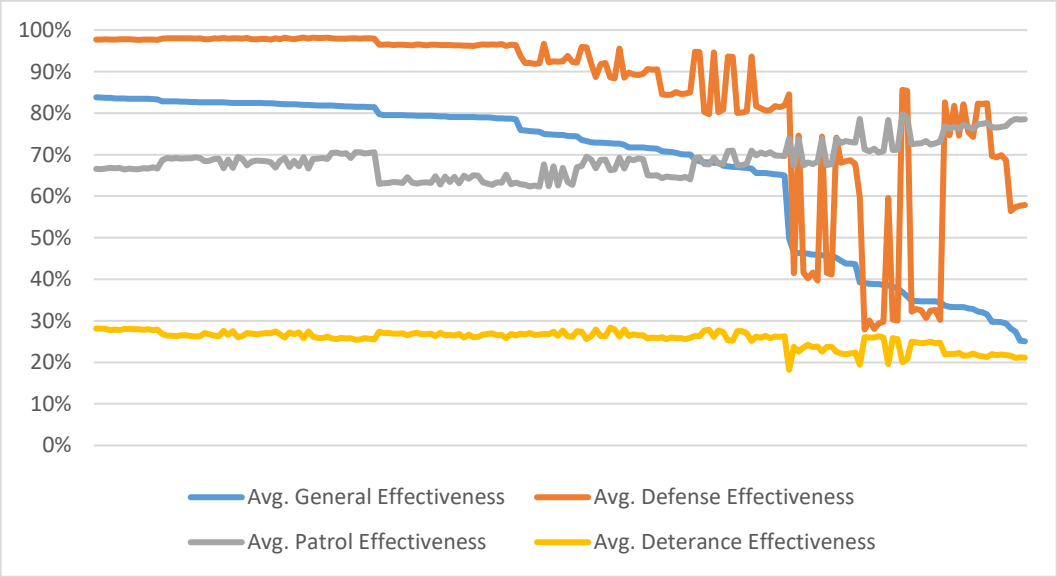

Figure 31 - Effectiveness Range against the Hybrid Adversary

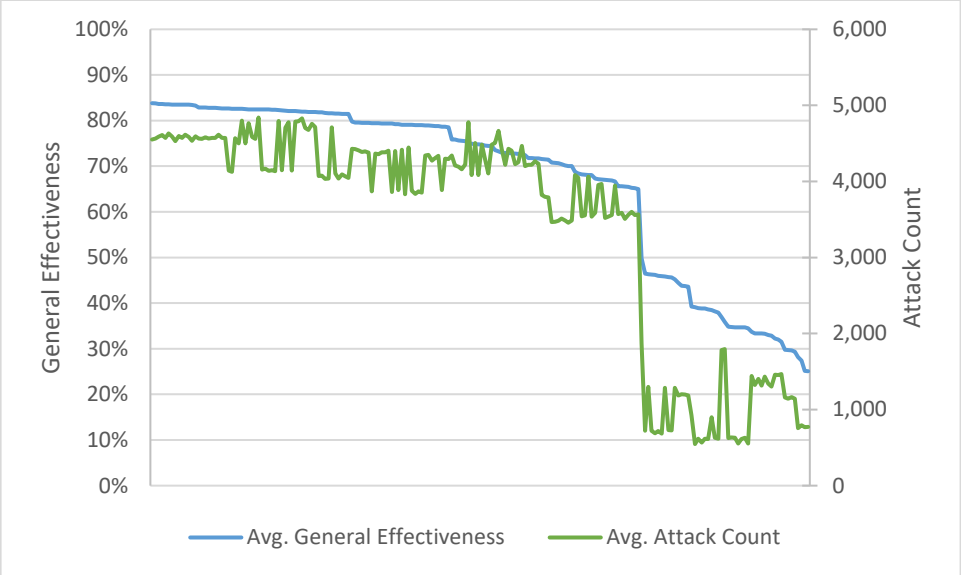

Figure 32 - General Effectiveness and Attack Count against the Hybrid Adversary

The simplest and highest performing agent as measured by general effectiveness is *covering_hl_pb_rnd* at 83.48%, which is within 99.6% of the maximum score of *covering_hl_pb_sl-vertex-rnd* at 83.81% (Figure 33).

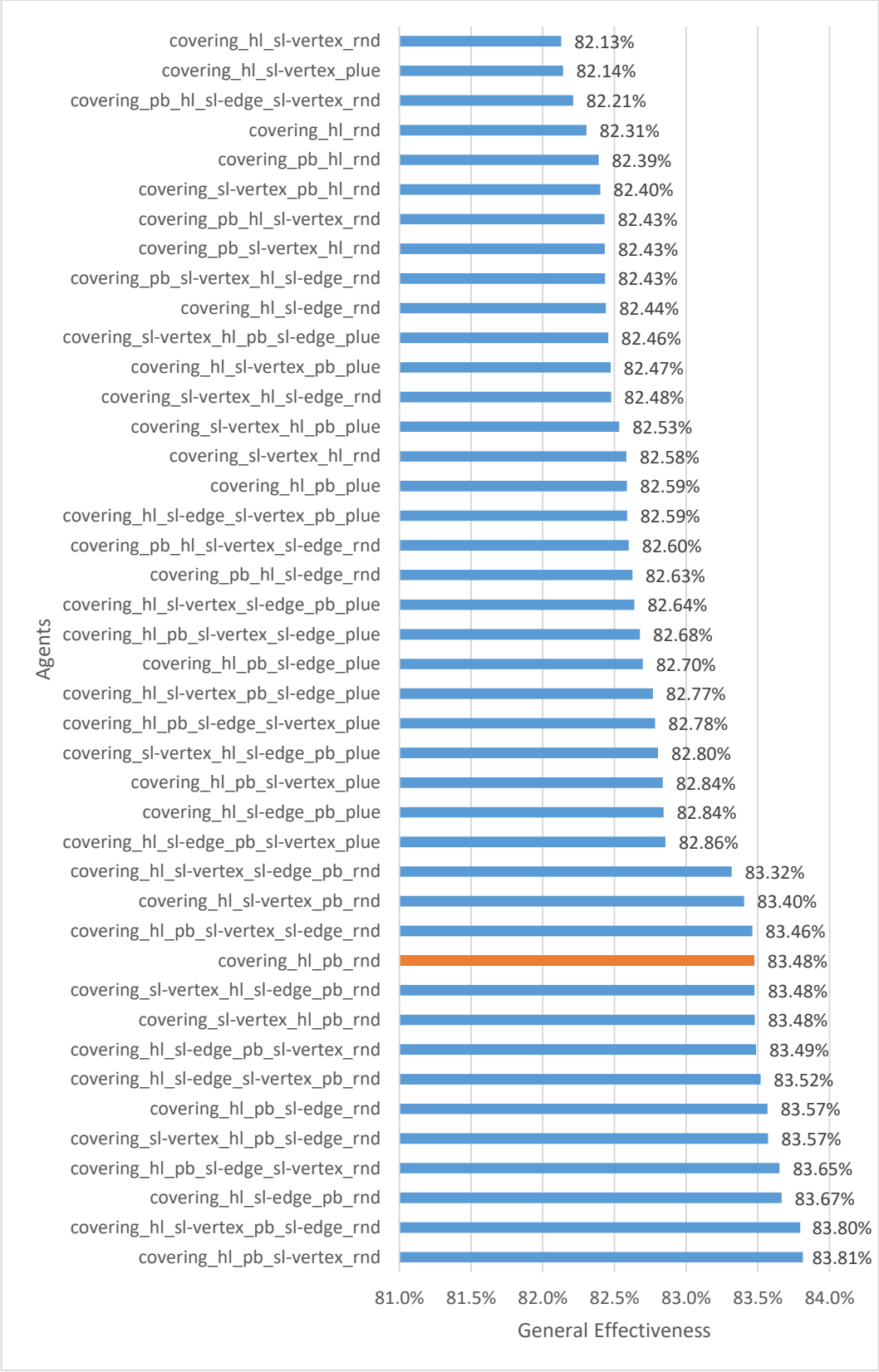

Figure 33 - General Effectiveness Against the Hybrid Adversary (Within 98% of Maximum)

Chapter 5.

Conclusions, Recommendations, and Summary

Conclusions

Overall

The top performing covering agent strategies were covering_hl_lue, covering_hl_pb_rnd, and covering_hl_rnd. The most effective agent against the random adversary was covering_hl_lue, against the statistical adversary is was covering_hl_rnd, and covering_hl_pb_rnd performed best against the waiting and hybrid strategies as well as best overall (Table 3).

Adversary	*Agent Name*	*Agent Description*	*General Effectiveness*
Random	*covering_hl_lue*	Hard-Limit, Least-Recently-Used- Edge	69.00%
Waiting	*covering_hl_pb_rnd*	Hard-Limit, Peek-back, Random	94.06%
Statistical	*covering_hl_rnd*	Hard-Limit, Random	92.06%
Hybrid	*covering_hl_pb_rnd*	Hard-Limit, Peek-Back, Random	83.48%
Overall	*covering_hl_pb_rnd*	Hard-Limit, Peek-Back, Random	82.20%

Table 3 - Top Performing Covering Agents

The top agent strategies do not employ all edge choosing primitives: only combinations of hard-limit, peek-back, least-recently-used-edge, and random were used by the top scoring agents. This means that the two soft-limit edge choosers, the vertex and the edge focused ones, in addition to the probabilistic-recently-used-edge primitive, were not a key factor for maximizing agent performance, as measured by general effectiveness. As described in Table 4, the hard-limit primitive was effective against all adversaries, the peek-back primitive was effective against the waiting and hybrid

strategy, and the random primitive was effective against waiting, statistical, and hybrid adversaries.

	Hard Limit (hl)	*Peek-Back (pb)*	*Least Recently Used Edge (lue)*	*Random (rnd)*
covering_hl_lue (Against Random)	X		X	
covering_hl_pb_rnd (Against Waiting & Hybrid)	X	X		X
covering_hl_rnd (Against Statistical)	X			X

Table 4 - Edge Choosing Primitive Usage in Top Covering Agents

When the results of all experiment variables are averaged, the top performing agent was *covering_hl_pb_rnd* with a general effectiveness of 83.20%, which is also the best agent for the hybrid adversary (Figure 34). Closely following that are the *covering_hl_rnd* and *covering_hl_lue* agents, with general effectiveness scores of 81.32% and 77.20%, respectively. The control agents scored below those and the basic covering agents scored even lower. Basic covering agents with only a single terminal edge choosing primitive perform worse than their corresponding control agents; the covering edge choosing agents only show high performance when used with multiple primitives.

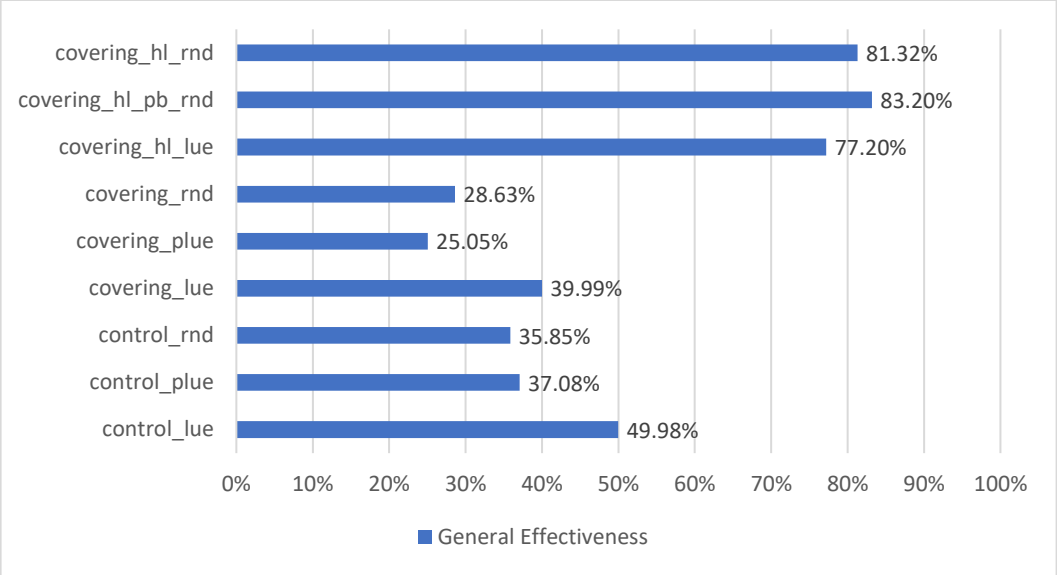

Figure 34 - Overall General Effectiveness of Control, Basic Covering, and top Covering Agents

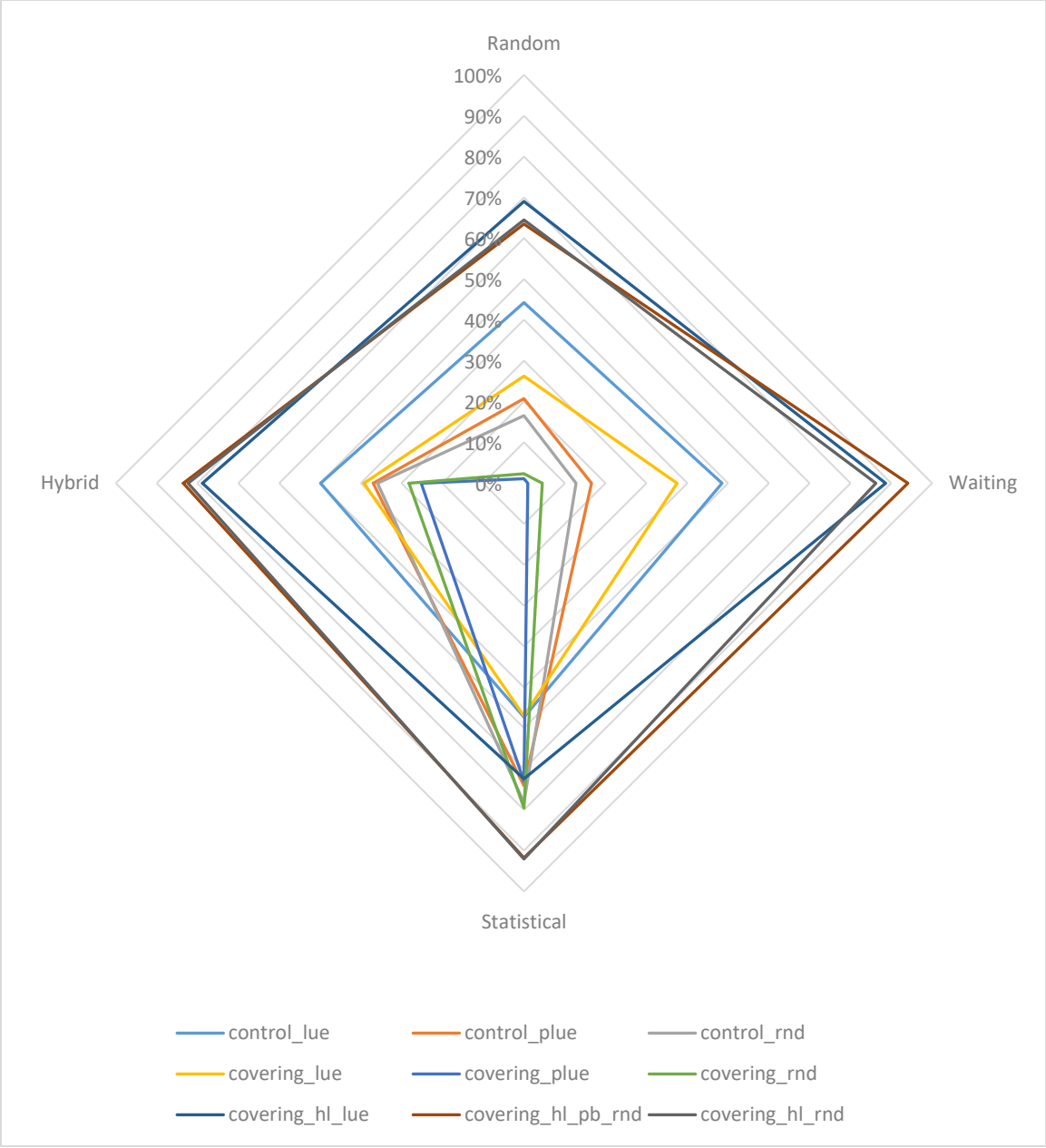

Figure 35 - General Effectiveness of Control, Basic Covering, and top Covering Agents against Adversaries

By Adversary

The *control_lue* agent shows roughly equal performance against all four adversaries, while the *control_plue* and *control_rnd* performed much better against the statistical adversary, less well on the hybrid strategy, and not well against the random and waiting adversaries (Figure 36). The basic covering agents (Figure 37) have a similar result, but

with lower scores of general effectiveness, a less equal performance of the *covering_lue* agent against all adversaries, and near zero scores of general effectiveness of the *covering_plue* and *covering_rnd* agents against the random and waiting adversaries.

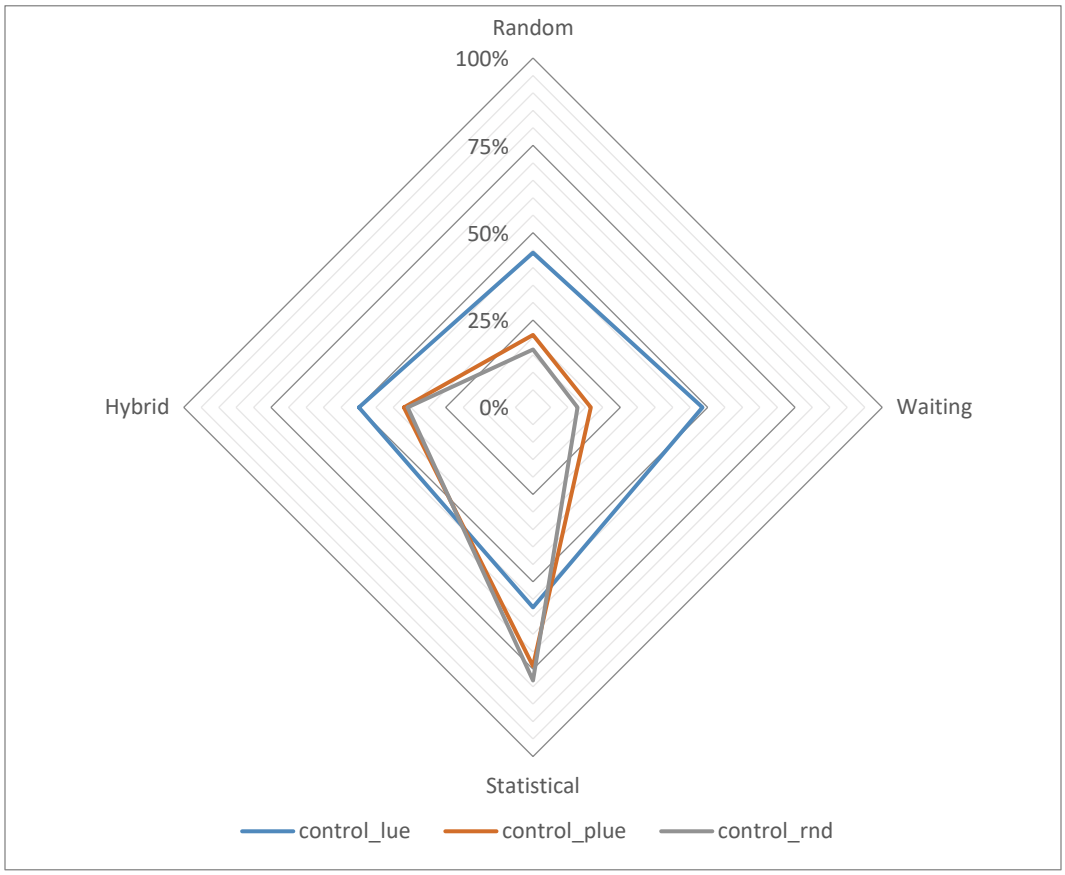

Figure 36 - General Effectiveness of Control Agents Against all Adversaries

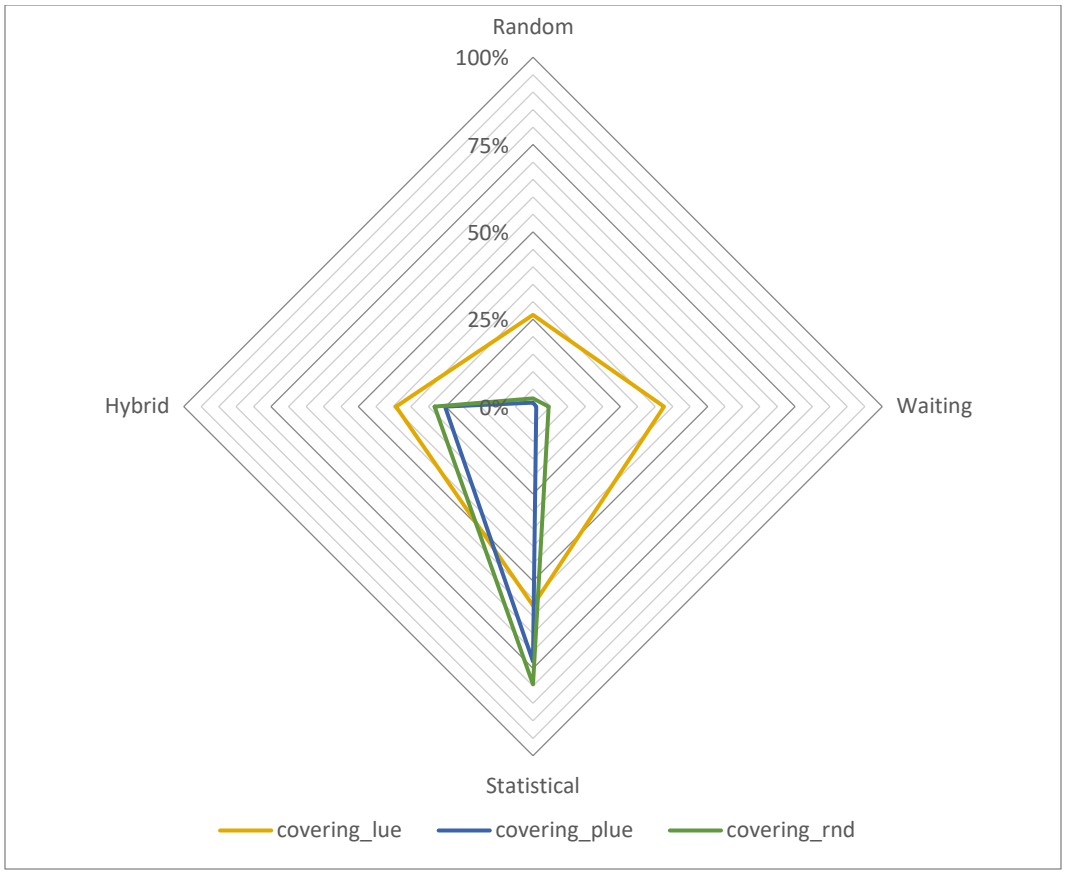

Figure 37 - General Effectiveness of Basic Covering Agents Against all Adversaries

In contrast, as can be seen in Figure 38, the top performing covering agents perform equally well against all adversaries. One notable difference is that *covering_hl_lue* agents performs worse against the statistical adversary in comparison to the *covering_hl_pb_rnd* and *covering_hl_rnd* agents. This is not surprising because the *covering_hl_lue* agent moves about the graph predictably while the other two agents do not.

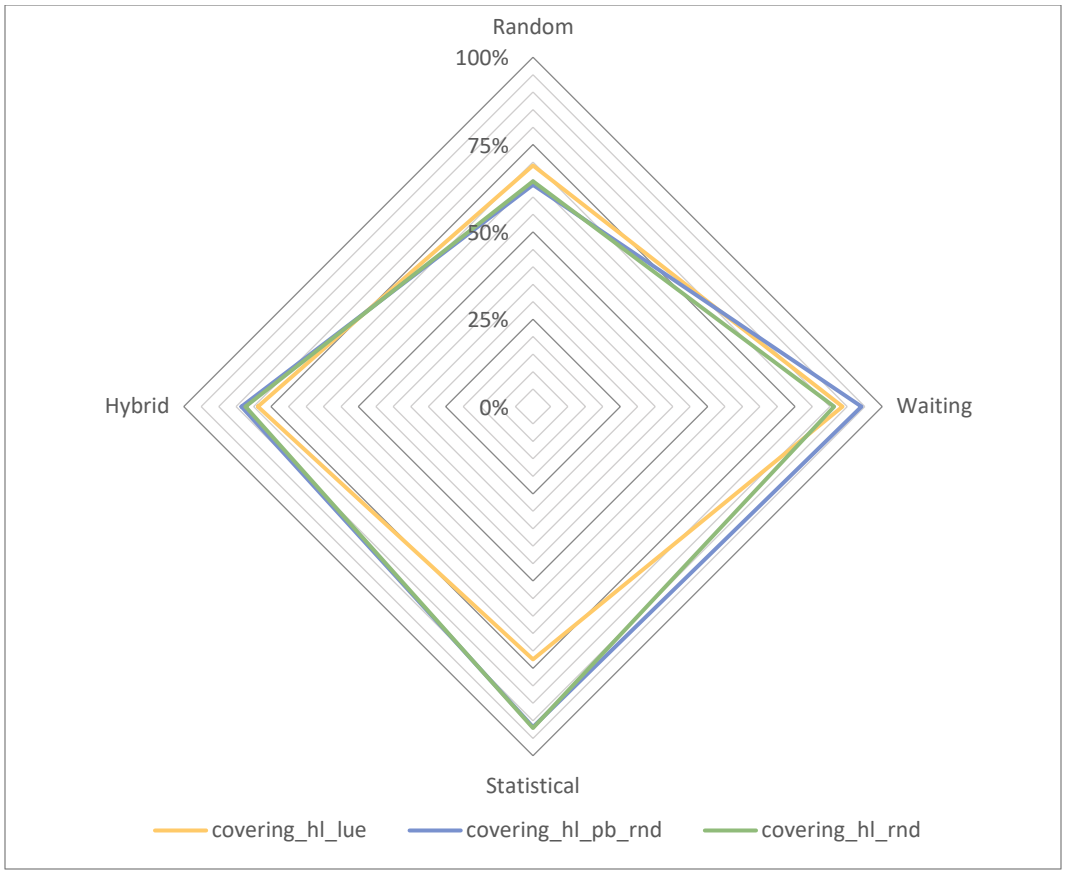

Figure 38 - General Effectiveness of Top Covering Agents Against all Adversaries

By Graph

To analyze the effect that graphs have on agent performance, the results from each game were grouped by graph and the general effectiveness against all adversaries were averaged. The analysis of the agents for the graphs are grouped into three categories: control agents, basic covering agents, and the top performing covering agents. The analysis shows that the top performing covering agents have both higher scores of general effectiveness as well are more consistent performance for all the graphs.

When comparing the general effectiveness of the control agents, the *control_plue* and *control_rnd* agents perform about equally no matter graph is used (Figure 39).

However, the *control_lue* performs much better on the corridor graph than the others; it also tends to outperform the other control agents on all graphs except for graph A.

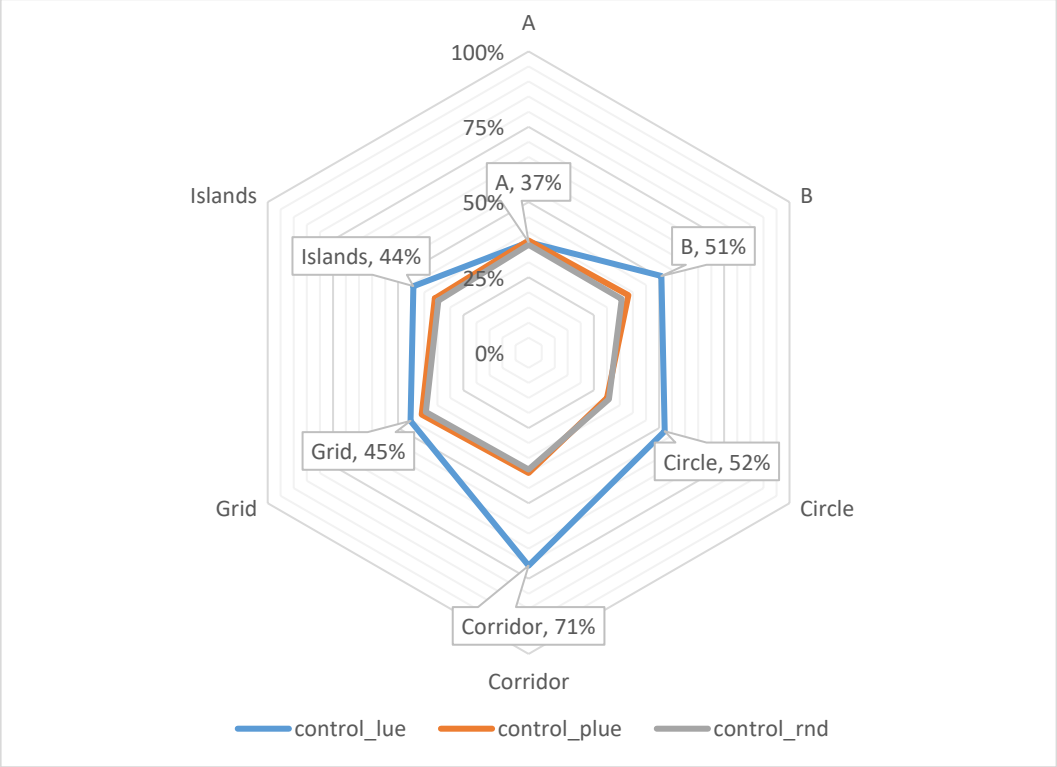

Figure 39 - Control Agent General Effectiveness by Graph

The basic covering agent strategies have the same general result, but with lower general effectiveness scores overall and a lower score specifically in the "Grid" graph (Figure 40). Thus, the control agent strategies perform better than the basic control agents, overall, regardless of the graph. Thus, there is a cost for the more complex behavior of the covering agents. However, it will be shown next that this cost is recovered and the general effectiveness scores are greatly increased when certain combinations of ECPs are used.

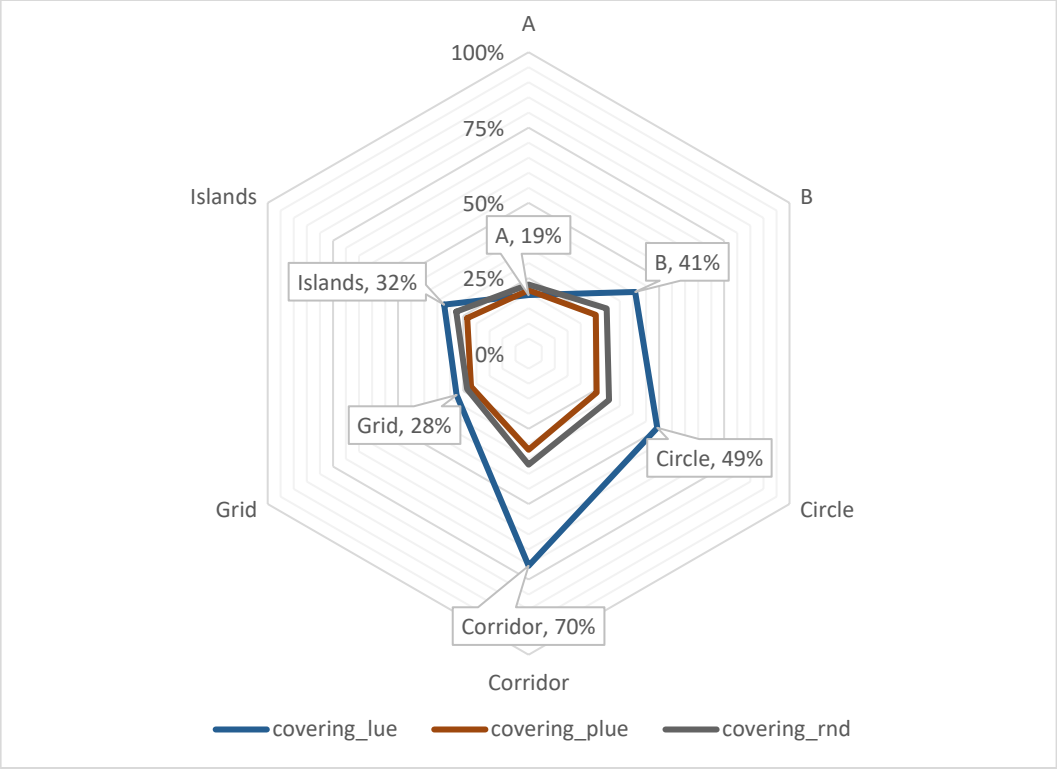

Figure 40 - Basic Covering Agent General Effectiveness by Graph

The top performing covering agents have a very different performance characteristic than the control and basic covering agent strategies; they show very uniform and much improved performance no matter which graph is being used (Figure 41). One slight deviation is that the *covering_hl_lue* agent underperforms in the "A" graph (and to lesser extend the Islands, B, and Grid graphs) than the *covering_hl_pb_rnd* and *covering_hl_rnd* do. As a whole, though, these top performing covering agents handle each of the graphs extremely well, much better than the control and basic covering agent strategies.

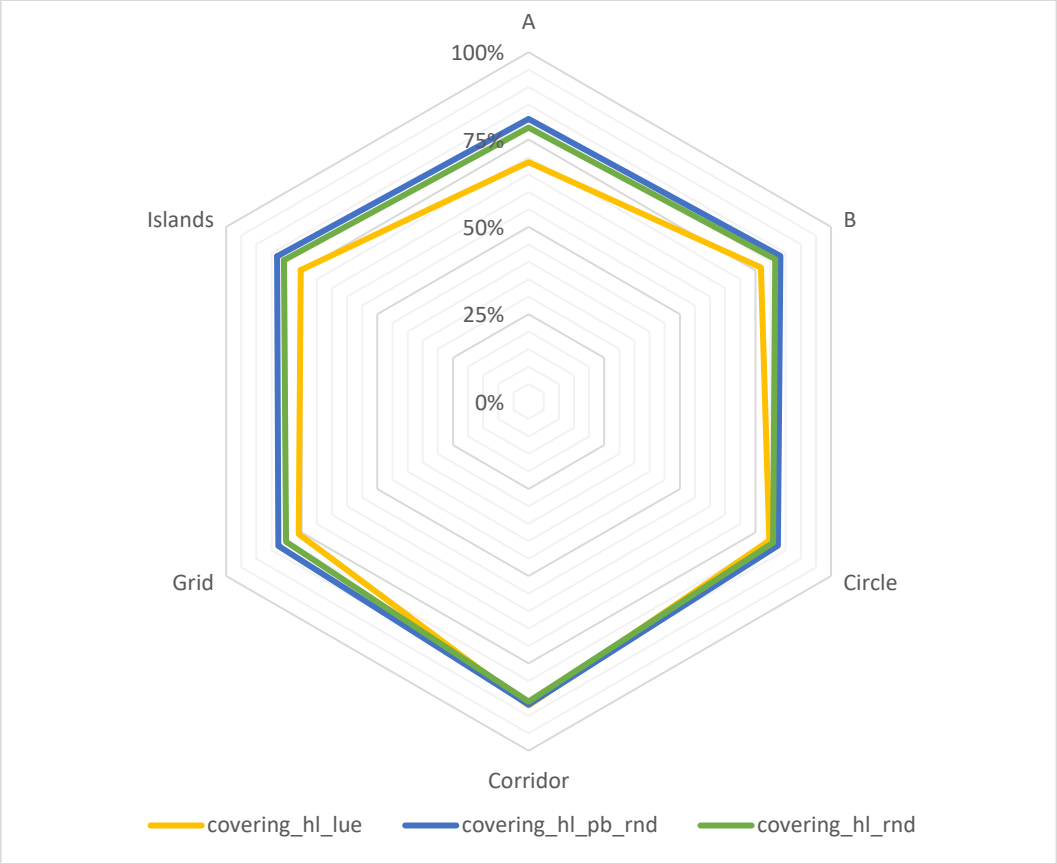

Figure 41 - Top Covering Agent General Effectiveness by Graph

Recommendations

Limitations

There are three major limitations in this work. The first limitation is that only four adversaries (random, waiting, statistical, and hybrid) were used. While the random, waiting, and statistical adversaries have their provenance in previous research (Sak, Wainer, & Goldenstein, 2008), other adversaries are possible that are more sophisticated than the random, waiting, statistical, and hybrid adversaries. The effectiveness of the proposed agent strategies against other adversary types has not been considered here.

The second limitation is that other ECPs could be designed that perform better than the ones used in this work. Each ECP of this work have low time and space requirements,

basically linearly as a function of either the number of edges or critical vertices. Thus, these ECPs execute efficiently in terms of time and space. It is feasible that other ECPs could be designed that perform better though they may be less efficient. An example may be an ECP that uses neural networks to monitor their target vertex and learn when the most optimal time to attack is.

The third limitation is the graphs. While the graphs are from previous research related to this problem (Almeida, et al., 2004), it is not well understood how agent performance depends on graph topology generally. The experiments have been limited to these select graphs. Questions of dependence of strategy effectiveness on graph topology has only been touched upon.

Problem Variations

A potential variation of the problem is to limit the agents' knowledge of the attack interval, *K*. In this work, the fixed attack interval was known to the agents. Instead, the attack interval would be given to the agents as a range as possible values, where each adversary's attack interval lies within this range (the present research considers the trivial range [*K*, *K*]). The attack interval for an adversary a is $K(a) \in [K_{low}, K_{high}]$, where K_{low} and K_{high} are fixed values that the agents know (they are experimental settings). Hence the agent does not know exactly how long it has before a targeted vertex will be compromised, though this time period is constrained. In all the agent designs in this work, a notion of a return-to-vertex deadline was used to inform the agent when it should return to a vertex to minimize successful attacks on it. In some agents this is a hard deadline while in others it is soft. In either case, the agent is not sure of which timestep it must return to a critical vertex to thwart an attack on it.

When a hard deadline is required, such as against the waiting adversary, the agent must return to the vertex at the lower bound of the range, to be certain that an attack is thwarted. However, doing so will not give the agent any information regarding the attack interval $K(a)$ for adversary a. Instead, the agent maintains a range $[K_{low}(a), K_{high}(a)]$ for each adversary *a* that it covers and may use strategies to narrow this range. As it does so, it can hone itself to better protect the vertices of each adversary. This approach will cause at least one attack to succeed; the agent should find just one critical vertex and play this game against it alone until the value of *K* is known. After that, the agent can explore the graph and find other critical vertices. However, this approach is not as simple as it may seem because agents cannot stay on a vertex for succeeding time steps, an incident edge must always be chosen. Thus, the agent strategy must account for the edge weights to know the actual durations it may be away from a critical vertex. So, playing this game against a single adversary on a particular critical vertex will not necessarily make *K* known. It would, however, allow the range of values of *K* to be reduced in length after succeeding games against multiple adversaries at their critical vertices. Because no ECPs which used the soft deadline were used by the top performing agents, the implementation with soft deadlines will not be considered.

Another problem variation is to further restrict the amount of communication between the agents. In this work, the only information that is shared between the agents are the critical vertices. This could be removed and replaced with a flag that an agent can leave at a critical vertex, which other agents take note of when arriving at that vertex. Thus, agents could only learn of new critical vertices by visiting a vertex and checking the flag. In practical terms, the original problem statement could be thought of each agent

broadcasting the critical vertices it has discovered by using electromagnetic emissions that are received by all agents as soon as a critical vertex is discovered. However, the assumption that this is possible in real world situations is not necessarily valid. Thus, in its place, physical markers could be left at vertices by the agents as they travel to them. The basic effect on the agents is the same, the agents learn of the critical vertices over time, but the rate at which they learn the critical vertices happens more slowly instead of instantaneously. This could have applications in practical scenarios such as when the graph is a computer network or a location where electromagnetic propagation is severely limited.

A problem variation in the other direction than above is for agents to know the graph topology from the start as well as which vertex each agent is at for each timestep. However, no other information is shared such as the critical vertex set or any intentions of the other agents. Thus, the agents would not directly know which vertices are critical or covered by other agents, such as in this work. Instead, this information would have to be inferred by the agents, such as by monitoring the behavior of other agents. The agents could also patrol the graph to thwart attacks and gain direct knowledge of which vertices are critical and apply this knowledge to their inferences. It is interesting to consider that the agents would monitor each other to infer each other's state and intentions and to cooperate effectively from the inferred knowledge.

Summary

This work on the domain of multi-agent adversarial patrol problems, which is applicable to interesting practical applications, resulted in the creation of novel agent strategies that outperform agent strategies of previous work. The information available to

the agents was purposely kept minimal, with the only shared information between them being the critical vertices. The agents had very limited access to the environment to reproduce the limited information that would be available to such agents in a practical setting.

The goal of creating new heuristic agent strategies to counter each of the adversaries was achieved. Additionally, a new universal (or general) agent strategy was designed and found through experiments and analysis of results to be capable of countering all adversary types in the graphs that were considered. The three research questions were answered on how to develop effective agent strategies against each of the adversaries: that a general agent strategy could indeed be created, how these agents perform under a variety of conditions, and which agent strategies performed better and why.

The methodology was derived directly from the research goals and questions, resulting in the design of three adversary strategies as well as a fourth that is hybrid of those three. This hybrid strategy is a new approach that drives more sophisticated agent strategy design. The problem formulation resulted in an approach to agent design that is based on a new chained component architecture, resulting in almost 200 new agent strategies by permutations of these components. Additionally, the new concept of agent critical vertex covering was introduced and designed into each of these new agent strategies. This covering capability allows the agent strategies to maximize protection of critical vertices while also maximizing patrolling for other vertices subject to attack in an efficient and emergent manner.

The experiment design was created with inspiration from previous research in this problem domain, to provide a clear lineage of the significance of this work. New

performance metrics were created that provide better insight into the nature of an agent's functioning under a very wide range of experiment scenarios. The wide range of experiment scenarios were broken down into sub-categories that enabled the results to be evaluated from many different viewpoints to provide insight under what conditions an agent does or does not perform well. To produce these results, millions of simulations were run, with each simulation having a duration of approximately tens of thousands of timesteps. The experiment took approximately 4 days to run a 32-core CPU machine; the experiment software was designed to execute in a highly parallel manner.

Three categories of agent strategies were created: control, basic covering, and chained covering. The three control agent strategies were designed to counter the random, waiting, and statistical adversaries. The three basic covering strategies use critical vertex covering and consist of only a single terminal ECP; each basic covering agent designed to counter the random, waiting, and statistical adversaries. However, the basic covering strategies did perform as well as the control strategies. It appears that adding critical vertex covering with only a single terminal ECP shows no advantage and in fact is detrimental. Lastly, the almost 200 chained covering agent strategies, each one a permutation of all possible ECP combinations.

Three of the chained covering agent strategies were notable as outperforming all others under all experimental variables. One of them was superior too all other agent strategies, including the control and basic covering ones. That agent strategy was covering_hl_pb_rnd, which is the chain of the following ECPs, in order: Hard-Limit, Peek-Back, and Random. This agent outperformed all other agents under the performance measures. It also performed roughly equally well no matter which adversary, graph, or

other experiment variables change. Thus, this agent strategy appears to be universal and consistently performant in all situations.

Appendix A

General Agent Strategy

The General Agent strategy of this work was the agent composition of ECPs that outperformed all other agent strategies in terms of general effectiveness. This agent uses vertex covering to assign each critical vertex to exactly one agent. Each agent takes sole responsibility for protecting its own critical vertices and avoids the critical vertices of other agents. The General Agent Strategy is composed of a chain with the following ECPs, in order: Hard-Limit, Peek-Back, and Random. This particular combination of ECPs (in this specific order) performs the best against all the adversaries: random, waiting, statistical, and hybrid. When this agent arrives at a vertex and must choose an incident edge to travel to next, it first asks the Hard-Limit ECP to choose an edge. If the Hard-Limit ECP declines to choose an edge, the Peek-Back ECP is then asked to choose an edge. If it also declines to choose an edge, the Random ECP will always choose an edge. The combination of these three ECPs performed the best among all other permutations of ECPs. Each ECP will be given a brief description, below.

The Hard-Limit ECP will only choose an edge if it heuristically determines that it must immediately begin travelling back to a covered vertex to reach it before K timesteps have elapsed since last leaving it. If that situation occurs, the ECP will pick the edge that will cause the agent to return to vertex the quickest. If there are multiple vertices that fit the criteria, it will pick the vertex that it can reach the soonest. If the criteria are not met, the Peek-Back ECP follows.

The Peek-Back ECP will check every vertex once by departing it along an incident edge and then, after arriving at the other vertex, immediately return back to the original

vertex along the same edge. It does this to catch a waiting adversary in the act of attacking the vertex after leaving it. Once a vertex has been checked, it is never checked again and so if the agent travels to a checked vertex again in the future, the Random ECP follows.

The Random ECP randomly picks an incident edge of the current vertex. However, it will avoid choosing an edge whose endpoint it knows to be a covered vertex of another agent. In the case where all incident edges have endpoints that go to a covered vertex of another agent, this ECP will choose one of them randomly.

References

Acevedo, J. J., Arrue, B. C., Maza, I., & Ollero, A. (2013). Cooperative Large Area Surveillance with a Team of Aerial Mobile Robots for Long Endurance Missions. *Journal of Intelligent & Robotic Systems, 70*(1), 329-345. doi:10.1007/s10846-012-9716-3

Agmon, N. (2010). On Events in Multi-Robot Patrol in Adversarial Environments Categories and Subject Descriptors. *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010). 2*, pp. 591-598. Toronto, Canada: International Foundation for Autonomous Agents and Multiagent Systems.

Agmon, N., Kaminka, G. A., & Kraus, S. (2011). Multi-Robot Adversarial Patrolling: Facing a Full-Knowledge Opponent. *Journal of Artificial Intelligence Research, 42*, 887-916. doi:10.1613/jair.3365

Agmon, N., Kraus, S., & Kaminka, G. A. (2008). Multi-Robot Perimeter Patrol in Adversarial Settings. *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (pp. 2339–2345). Pasadena, CA: IEEE Computer Society. doi:10.1109/ROBOT.2008.4543563

Agmon, N., Kraus, S., & Kaminka, G. A. (2009). Uncertainties in Adversarial Patrol. *AAMAS '09 Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems. 2*, pp. 1267-1268. Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems.

Agmon, N., Kraus, S., Kaminka, G. A., & Sadov, V. (2009). Adversarial Uncertainty in Multi-Robot Patrol. *International Joint Conference on Artificial Intelligence*, (pp. 1811-1817).

Agmon, N., Sadov, V., Kaminka, G. A., & Kraus, S. (2008). The Impact of Adversarial Knowledge on Adversarial Planning in Perimeter Patrol. *AAMAS '08 Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. 1*, pp. 55-62. Estoril, Portugal: International Foundation for Autonomous Agents and Multiagent Systems.

Alam, T., Edwards, M., Bobadilla, L., & Shell, D. (2015). Distributed Multi-Robot Area Patrolling in Adversarial Environments. *International Workshop on Robotic Sensor Networks.* Seattle, WA.

Almeida, A., Ramalho, G., Santana, H., Tedesco, P., Menezes, T., Corruble, V., & Chevaleyre, Y. (2004). Recent Advances on Multi-agent Patrolling. In *Advances in Artificial Intelligence - SBIA 2004* (pp. 474-483). Springer. doi:10.1007/978-3-540-28645-5_48

Amigoni, F., Basilico, N., & Gatti, N. (2009). Finding the Optimal Strategies for Robotic Patrolling with Adversaries in Topologically-represented Environments. *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on* (pp. 819-824). Kobe: IEEE. doi:10.1109/ROBOT.2009.5152497

Amigoni, F., Gatti, N., & Ippedico, A. (2008). A Game-Theoretic Approach to Determining Efficient Patrolling Strategies for Mobile Robots. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on, Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on 2008* (pp. 500-503). IEEE. doi:10.1109/WIIAT.2008.324

Basilico, N., Gatti, N., & Amigoni, F. (2009). Developing a Deterministic Patrolling Strategy for Security Agents. *WI-IAT '09 Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology. 2*, pp. 565-572. IEEE. doi:10.1109/WI-IAT.2009.212

Basilico, N., Gatti, N., & Rossi, T. (2009). Capturing Augmented Sensing Capabilities and Intrusion Delay in Patrolling-intrusion Games. *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games* (pp. 186-193). Milano: IEEE. doi:10.1109/CIG.2009.5286477

Basilico, N., Gatti, N., Rossi, T., Ceppi, S., & Amigoni, F. (2009). Extending Algorithms for Mobile Robot Patrolling in the Presence of Adversaries to More Realistic Settings. *WI-IAT '09 Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology. 2*, pp. 557-564. IEEE. doi:10.1109/WI-IAT.2009.211

Chevaleyre, Y. (2004). Theoretical Analysis of the Multi-Agent Patrolling Problem. *Intelligent Agent Technology, 2004. (IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on* (pp. 302-308). Beijing: IEEE. doi:10.1109/IAT.2004.1342959

Chevaleyre, Y., Sempe, F., & Ramalho, G. (2004). A Theoretical Analysis of Multi-Agent Patrolling Strategies. *Proceeding AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3* (pp. 1524-1525). New York, New York, USA: IEEE. doi:10.1109/AAMAS.2004.34

Conitzer, V., & Sandholm, T. (2006). Computing the Optimal Strategy to Commit to. *Proceedings of the 7th ACM conference on Electronic commerce - EC '06*, (pp. 82-90). doi:10.1145/1134707.1134717

Dorigo, M., & Gambardella, L. M. (1997, April). Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation, 1*(1), 53-66. doi:10.1109/4235.585892

Flint, M., Polycarpou, M., & Fernandez-Gaucherand, E. (2002). Cooperative Control for Multiple Autonomous UAV's Searching for Targets. *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on. 3*, pp. 2823-2828. Las Vegas, NV, USA: IEEE. doi:10.1109/CDC.2002.1184272

Franco, C., López-Nicolás, G., Sagüés, C., & Llorente, S. (2015). Adaptive Action for Multi-Agent Persistent Coverage. *Asian Journal of Control, 18*(2), 419-432. doi:10.1002/asjc.1152

Glad, A., Simonon, O., Buffet, O., & Charpillet, F. (2008). Theoretical Study of Ant-Based Algorithms for Multi-Agent Patrolling. *18th European Conference on Artificial Intelligence including Prestigious Applications of Intelligent Systems (PAIS 2008)* (pp. 626-630). Patras, Greece: IOS press. doi:10.3233/978-1-58603-891-5-626

Grace, J., & Baillieul, J. (2005). Stochastic Strategies for Autonomous Robotic Surveillance. *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on* (pp. 2200-2205). Seville, Spain: IEEE. doi:10.1109/CDC.2005.1582488

Hespanha, J. P., Kim, H. J., & Sastry, S. (1999). Multiple-Agent Probabilistic Pursuit-Evasion Games. *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on. 3*, pp. 2432-2437. Phoenix, AZ: IEEE. doi:10.1109/CDC.1999.831290

Iocchi, L., Marchetti, L., & Nardi, D. (2011). Multi-robot patrolling with coordinated behaviours in realistic environments. *IEEE International Conference on Intelligent Robots and Systems*, (pp. 2796-2801). doi:10.1109/IROS.2011.6048424

Machado, A., Ramalho, G., Zucker, J.-D., & Drogoul, A. (2002). Multi-Agent Patrolling: An Empirical Analysis of Alternative Architectures. *3rd International Conference on Multi-agent-based simulation II (MABS)* (pp. 155-170). Bologna, Italy: Springer. doi:10.1007/3-540-36483-8_11

Megiddo, N., Hakimi, S. L., Garey, M. R., Johnson, D. S., & Papadimitriou, C. H. (1988, January). The Complexity of Searching a Graph. *Journal of the ACM, 35*(1), 18-44. doi:10.1145/42267.42268

Park, C.-H., Kim, Y.-D., & Jeong, B. (2012). Heuristics for Determining a Patrol Path of an Unmanned Combat Vehicle. *Computers & Industrial Engineering, 63*(1), 150-160. doi:10.1016/j.cie.2012.02.007

Parsons, T. D. (1976). Pursuit-Evasion in a Graph. In Y. Alavi, & D. R. Lick (Eds.), *Theory and Applications of Graphs* (Vol. 642, pp. 426-441). Michigan: Springer Berlin Heidelberg. doi:10.1007/BFb0070400

Paruchuri, P., Pearce, J. P., Tambe, M., Ordonez, F., & Kraus, S. (2007). An Efficient Heuristic Approach for Security Against Multiple Adversaries. *AAMAS '07 Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems.* Honolulu, Hawaii: ACM. doi:10.1145/1329125.1329344

Paruchuri, P., Tambe, M., Ordóñez, F., & Kraus, S. (2006). Security in Multiagent Systems by Policy Randomization. *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 273-280). Hakodate, Japan: ACM. doi:10.1145/1160633.1160681

Pasqualetti, F., Durham, J. W., & Bullo, F. (2012). Cooperative Patrolling via Weighted Tours: Performance Analysis and Distributed Algorithms. *IEEE Transactions on Robotics, 28*(5), 1181-1188. doi:10.1109/TRO.2012.2201293

Portugal, D., & Rocha, R. P. (2013). Multi-Robot Patrolling Algorithms: Examining Performance and Scalability. *Advanced Robotics, 27*(5), 325-336. doi:10.1080/01691864.2013.763722

Praveen, P., Pearce, J. P., Tambe, M., Ordóñez, F., & Kraus, S. (2007). An Efficient Heuristic Approach for Security Against Multiple Adversaries. *AAMAS '07 Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems.* Honolulu, Hawaii: ACM. doi:10.1145/1329125.1329344

Sak, T., Wainer, J., & Goldenstein, S. K. (2008). Probabilistic Multiagent Patrolling. *Advances in Artificial Intelligence - SBIA - 19th Brazilian Symposium on Artificial Intelligence* (pp. 124-133). Savador, Brazil: Springer Berlin Heidelberg. doi:10.1007/978-3-540-88190-2_18

Sampaio, P. A., Ramalho, G., & Tedesco, P. (2010). The Gravitational Strategy for the Timed Patrolling. *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on. 1*, pp. 113-120. IEEE. doi:10.1109/ICTAI.2010.24

Santana, H., Ramalho, G., Corruble, V., & Ratitch, B. (2004). Multi-Agent Patrolling with Reinforcement Learning. *Proceeding AAMAS '04 Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. 3*,

pp. 1122-1129. New York, NY, USA: IEEE Computer Society. doi:10.1109/AAMAS.2004.180

Subramanian, S. K., & Cruz, J. B. (2003). Adaptive Models of Pop-up Threats for Multi-agent persistent Area Denial. *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on. 1*, pp. 510-515. Maui, HI, USA: IEEE. doi:10.1109/CDC.2003.1272614

Yan, C., & Zhang, T. (2016). Multi-robot patrol: A distributed algorithm based on expected idleness. *International Journal of Advanced Robotic Systems, 13*(6). doi:10.1177/1729881416663666