7-2020

Simulated Experince Evaluation in Developing Multi-agent Coordination Graphs

Andrew J. Watson

THESIS

Andrew G. Watson, Captain, USAF

AFIT-ENG-MS-20-J-015

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-J-015

SIMULATED EXPERIENCE EVALUATION IN DEVELOPING MULTI-AGENT COORDINATION GRAPHS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Andrew G. Watson, BS

Captain, USAF

July 2020

AFIT-ENG-MS-20-J-015

SIMULATED EXPERIENCE EVALUATION IN DEVELOPING MULTI-AGENT COORDINATION GRAPHS

Andrew G. Watson, BS

Captain, USAF

Committee Membership:

Dr. G. L. Peterson
Chair

Lt Col D. W. King, PhD
Member

Dr. G. B. Lamont
Member

Dr. J. S. Okolica
Member

AFIT-ENG-MS-20-J-015

Abstract

Cognitive science has proposed that a way people learn is through self-critiquing by generating 'what-if' strategies for events (simulation). It is theorized that people use this method to learn something new as well as to learn more quickly. This research adds this concept to a graph-based genetic program. Memories are recorded during fitness assessment and retained in a global memory bank based on the magnitude of change in the agent’s energy and age of the memory. Between generations, candidate agents perform in simulations of the stored memories. Candidates that perform similarly to good memories and differently from bad memories are more likely to be included in the next generation. The simulation-informed genetic program is evaluated in two domains: sequence matching and Robocode. Results indicate the algorithm does not perform equally in all environments. In sequence matching, experiential evaluation fails to perform better than the control. However, in Robocode, the experiential evaluation method initially outperforms the control then stagnates and often regresses. This is likely an indication that the algorithm is over-learning a single solution rather than adapting to the environment and that learning through simulation includes a satisficing component.

Table of Contents

List of Figures

List of Tables

SIMULATED EXPERIENCE EVALUATION IN DEVELOPING MULTI-AGENT COORDINATION GRAPHS

I. Introduction

Background to Research Problem

A reoccurring challenge of software design is determining a configuration of components and their interconnections to meet an architectural goal [1]. In these designs, the components themselves offer a set of desirable functionalities that, in the interest of software reuse, may be advantageous. A designer must select components for reuse and then determine how data moves between them. A trend of artificially intelligent systems is the combination of simpler sub-systems. Generative Adversarial Networks (GANs) are an example of the combination of components to create a larger system with greater capabilities than its members [2]. Often, these sub-systems are constructed by human designers. A method for designing these coordination interactions automatically could lead to reduced time spent on design, increased progress in design complexity, and the identification of otherwise unknown designs. An opportunity exists to lessen the human's design burden though self-organizing systems.

Coordination graphs identify agents in a multi-agent system and define the flow of information between the agents. In this research, the coordination graphs are stored as an adjacency matrix. By treating each component as an agent in a multiagent system, complex control problems can be framed as searching for an optimal coordination graph. Genetic programs can be useful to discover coordination graphs, but the fitness assessment can be costly to execute [3]. Genetic algorithms require a significant number of trial executions to develop a satisfactory solution.

Executions are often costly in terms of money, time, or safety. For example, the evaluation of a system controlling a self-driving vehicle is accompanied by costs not only of time and fuel, but also human safety. Reduction of the number of required executions can alleviate these costs. Genetic programs conduct learning over generations, which is a slow time scale for self-organizing. A means for a set of agents to learn faster is to incorporate cultural learning in which individuals learn from each other's experiences [4]. To accomplish this, the information gleaned from each test must be maximized and fully exploited. In classic evolutionary algorithms, the fitness assessment of each solution only determines that solution's probability to pass genetic material to the next generation. Once that probability is identified, the execution data is discarded. Use and reuse of execution data to inform development of AI systems has been explored for reinforcement learning problems using experience replay [5], [6], [7]. Alternatively, one-shot learning techniques attempt to enable a system to learn a task after only a single example [8]. The common goal of these techniques is to find ways to allow AI systems to learn more with less time spent in real-world execution.

This research hypothesizes that execution data, viewed as population experience, may be used to accelerate the development of solutions, and evaluates whether cultural learning in a generation based self-organizing system can speed the organizational process. By analogy, the genetic program guides the agents' genetic structure and the experiences, passed down through generations, provide their culture. The resultant agents are, then, a product of both genes and heritage. Could these agents guided by ancestral experiences develop more quickly or better than those which had no memory before their generation?

This research is focused on the development of a working method to automatically find effective coordination graphs for heterogenous multi-agent systems. Much of the research time was spent on designing and developing a graph-based coordination framework in which to construct and manipulate the coordination graphs such that they can be created and tweaked in interesting, flexible, and random ways. With the framework developed, the genetic program was designed to include a variety of evolutionary options. The coordination graph construct and genetic program were created with the intention that they can be adapted to multi-agent AI problems in other environments.

Evaluation used two experimental domains. The first domain is a simple sequence matching problem. In this environment, the participant graph's state is the index of the character to be guessed and the returned action is the character the graph guesses should be in that index. The graph is provided each index one at a time from 0 to 25. A complete output of this process will be a string of 26 lower case characters. The graph is scored on how well this string of characters approximates the alphabet of letters 'a' thru 'z'. For example, the output string 'bbeeeeehlllllmmmmpuuuuxxx' achieves a score of 20.2, while 'muezpcvelpaqmmdcneqiqjmps' receives only 11.8 (score calculation is described in sections on Action Distance and Score Calculation in Chapter IV). Individual agents in the sequence matching environment can produce a constant character given a constant index, produce a random character, or evaluate between characters based on priority.

The second experimental domain, Robocode, is a 1-v-1 2D tank game. The tanks much scan the environment for, track, target, and fire upon each other while also dodging enemy bullets and avoiding collisions. Collisions with enemies, walls, and bullets result in degrees of damage. Each robot begins with 100.0 hit points (or energy) and explodes when their energy is less than 0. Energy is expended when firing a bullet, but some energy is regained upon a successful hit. Component agents in this environment can examine the state, recommend an action, evaluate between multiple recommended actions, relay and process information about the state, or manipulate a single aspect of a recommended action.

This research attempts to utilize subjective experiences captured by genetically evolving agents to accelerate solution development. Coordination graphs of agents are constructed and manipulated through genetic programming to find fitter configurations. Memories of state and action pairs are recorded during fitness assessment and retained in a global experience bank based on the magnitude of change in the multi-agent system's score and age of the experience. Experiential simulated evaluation is introduced to the genetic program. This technique extracts cultural experiences from the population’s fitness assessment and uses the experiences to evaluate candidate behaviors for the next generation. Between generations, candidate agents revisit stored experiences. Candidates that respond similarly to good experiences and differently from bad experiences are included in the next generation.

Problem Statement

The problem this research seeks to solve is the automated development of coordination graphs for agents that facilitates collections of simple agents to achieve relatively difficult goals. A coordination graph developed after fewer iterations of fitness assessment is considered a better solution due to the potential cost of execution. A solution to this problem, could then, benefit from more efficiently using the data created during execution for more purposeful search in the solution space.

This research hypothesizes that the problem of agent combination can be solved in fewer rounds of fitness assessment by using experiential memories from previous execution phases to predict the relative fitness of candidate solutions prior to their actual execution. Specifically, this research operates in the context of genetic programming. Solutions are generated and tested against an environment. The data from the environment tests are saved and used in an experiential simulated evaluation of the next generation of solutions to guide generational crossover and mutation.

Purpose of the Study

The purpose of this study is to develop an evolutionary algorithm augmented such that the experiences gained during previous rounds of fitness assessment improve the rate of development in the population. This research developed a graph-based genetic program that searches through the space of coordination graphs of a set of relatively simple agents. Experiential simulated evaluation is added to evaluate cultural learning. Experiential simulated evaluation uses recorded experiential data from the collection of agents to run simulations which approximate the ability of a candidate solution to solve the problem. The algorithm is described generically to apply to any problem given an environment with state and action objects, a running performance score, and a set of agents to be combined into a coordination graph.

Research Questions

This research asks what is the effect of experiential simulated evaluation on the progress of development in an evolutionary algorithm? Further, it asks what is the most effective method for creating and retaining the experiences used for simulated evaluations? And, during evaluation, is it more beneficial to explore previously unsearched areas of the solution space, to exploit the already known good areas or the solution space, or a balance between both methods?

Evaluation leverages four performance measures: terminal score, inflection point, rate of change and rate of change to inflection point to capture how the experiential learning affects both the speed and performance of the self-organizing coordination graph. A set of secondary questions to the inclusion of experience based cultural learning are how memories should be retained and used. Since the method used to retain experiences in the buffer can influence the effects of experiential simulated evaluation on the population’s progress. Experimentation tested over several parameters including the buffer retention method, experience highlight extraction, and exploitation or exploration tradeoffs.

Results

A. Significance of the Study

Results showed that, experiential evaluation affects the rate of progress differently depending on the environment. In sequence matching, experiential evaluation succumbs to overlearning and fails to outperform the control. In the Robocode environment, experiential simulated evaluation accelerates the initial progress of an evolutionary algorithm. This is significant because it demonstrates that a boost may be provided through this technique to identify a solution in fewer execution cycles in some environments.

Depending on the required level of success for a solution, the method presented in this research can be used as the sole solution or may be combined with other genetic programming methods.

The graph-based coordination framework developed in this research and the associated genetic program can be used separately or with experiential simulated evaluation for applications in other environments. This research found that though successful behaviors converged to a similar strategy, their genotypic representation of the strategy was diverse. This demonstrates that the development of coordination graphs for complex behavioral solutions can be intrinsically diverse while achieving extrinsic similarity.

B. Limitations of the Study

In one environment, experiential simulated evaluation failed to outperform the control at all. This result was surprising since it was the simpler of the two environments where credit is more easily assigned. In a more complex environment, experimental results indicate that, compared to the baseline, experiential simulated evaluation initially outperforms the control, but it stagnates and even regresses. This is likely an indication that the algorithm is over-learning a single solution rather than adapting to the environment. A trade-off is observed between accelerated, unstable development with experiential simulated evaluation versus slower, consistent progress achieve with a baseline evolutionary algorithm.

Summary

The problem addressed in this research occurs commonly in the field of artificial intelligence. An automated solution could unlock faster development of combination-based systems. This research attempted to find a solution for developing coordination graphs in a reduced number of execution cycles through repeated reuse of previously gathered experiential data to guide the evolutionary trajectory. The two primary components of this research are the development of the graph-based coordination framework and the development of the experiential simulated evaluation. Although the method presented consistently stagnated prematurely, the accelerated progress during initial development has potential utility. The remainder of this document presents the background knowledge and prior work, the implementation of the graph coordination graphs and experiential simulated evaluation, and the results of experimentation with the algorithm.

II. Literature Review

The concept of emergence in multi-agent systems influenced the development of early subsumption architectures, which led to the Unified Behavior Framework (UBF) from which this work is adapted. Due to their weak emergence property, behavior-based systems can be difficult to develop and making this easier is an open research topic. Separate from behavior-based systems research; internal simulations have proven effective for improved decision-making in reactive robots.

This chapter provides the background for this research. Previous studies in artificial intelligence and robotic control are presented to offer foundational understanding. This chapter reviews the background works starting with the foundations of behavior-based systems for reactive control including the concept of emergence, previous implementations of multi-agent systems (MAS), subsumption architectures, and, specifically, the Unified Behavior Framework. This chapter finally examines previous techniques to autonomously develop other multi-agent and behavior-based systems.

Behavior-Based Systems for Reactive Control

A. Emergence

The principle of emergent behavior is discussed in disparate fields including biology, sociology, psychology, and philosophy in addition to artificial intelligence. Emergence is the arising of complex dynamics from the interaction of simpler dynamics [9]. The theory of weak emergence claims that the emergent behavior of any system can be understood through understanding of its component parts even if that understanding comes only by use of simulation. In contrast, the theory of strong emergence claims that emergent dynamics cannot be causally linked to the mere interactions of sub-components [10]. Strong emergence implies that emergence is an unknowable property of nature rather than an understandable, deconstructable, or applicable process.

It is helpful to introduce the concept of emergence in multi-agent systems with an analogy. Ant colonies are frequently used to explain emergence in a biological multi-agent system [11]. Artificial agents may be compared to the individual ants in the colony. In such systems, the simple actions taken by individuals in response to the local signals (such as pheromone trails) and environment sensing result in complex behaviors of the group such as shortest path following, brood maintenance, and colony defense. This natural elegance is a desirable characteristic for artificial multi-agent systems. If achieved, it allows the designer to focus on simple goals and responses. An algorithm working with this characteristic can realize a complex result through simple repeated calculations.

B. Multi-Agent Systems

In an ant colony, the many members specialize and arrange themselves naturally through simple rules. Optimal organizational arrangements are similarly required for multi-agent systems designed by humans [11]. Many researchers have sought to develop a self-organizing multi-agent system comprised of homogeneous or heterogeneous agents designed to sense the environment and determine a response action to control robotic behavior.

Robotic hardware and software development have often been influenced by natural designs [12]. Verschure, Kröse, and Pfeifer [13] developed a behavior structure between sensors and a motor actuator to control a simulated robot. The interactions between their nodes were learned with behavior conditioning. The nodes used and the structure of their connections remained static while the weighting of the connections was learned. Their results found that the weights were learned well enough to improve the robot’s navigation, but the final settings were still sub-optimal compared to a human design.

Yamashita and Tani [14] implemented a functional hierarchy of control behaviors in multiple time-scales to control a humanoid, box-manipulation robot. They structured the problem for the system to create a self-organization hierarchy both spatially and temporally. Their results indicated that the temporal component is essential for mimicking the interplay of primitive and complex behaviors in biological intelligences.

C. Multi-Agent Coordination

The fundamentally unique component of multi-agent systems as compared to the single agent alternative is computation distribution and the required coordination scheme. A subset of multi-agent coordination problems allows communication between all members. Self-evidently, systems in which each agent communicates with every other agent scales poorly. Coordination graphs are used to constructively limit the communication channels between agents to only the most beneficial. The problem of coordination optimization has been studied with multiple techniques.

The automated identification of the most beneficial communication channels is an open challenge. Kok and Vlassis [15] presented a method to using the max-plus algorithm to discover a near-optimal coordination graph as an alternative to variable elimination in densely connected cyclical graphs. In their application, robots communicated to maneuver as a team of soccer players.

Baum [16] implemented a model found on evolutionary economics called “The Hayek Machine” which was demonstrated to perform better than previous learning models on the block world game. In his work, information was traded as a finite resource forcing agents to determine the value of communication efficiently. Gu, Balasubramanian, and Norrie [17] constructed a multi-agent system for planning and scheduling based on a bidding process. Similarly, researchers have implemented bidding economies for energy distribution [18]. Sartoretti [19] proposed a leader-based multi-agent swarm where leader agents distribute reward to follower agents which best meet the leader’s goals. Kononen [20] introduced a similar approach in which the leader agents were given information about the decision processes of the follower agents and encouraged the followers to choose actions that benefit the leader.

Guestrin, Koller, and Parr [21] formulated the entire coordination problem as a Markov decision process (MDP) and approximated the joint value function using factored linear value functions. This approximation technique limited the explorable solution space. Yagan and Tham [22] continued this work by developing an algorithm to implement reinforcement learning on a coordination graph for a multi-agent system. The coordination graph is a key component of this work. It is a parameterized structure that defines which agents communicate with each other in the execution of the system.

D. Subsumption Architectures

Distributed behavior control for robotic systems is an ongoing research area. Brooks' subsumption architecture [23] prompted many studies of optimal composition methods of behavioral agents. In principle, subsumption architecture design sought to mimic the theory of layered competencies in the brain. Multiple behavioral logics were composed in a hierarchical structure such that higher behaviors could manipulate or override lower behaviors. Brooks implemented this concept for a mobile robot which demonstrated the ability to execute an overarching goal to carry an object across the room while also considering sub-goals such as obstacle avoidance.

Brooks' original design addressed limitations described by Rosenblatt and Payton [24]. A primary limitation included difficulties deciding between conflicting goals. Since in Brooks' work higher-order behaviors would completely subsume those below them, goal conflictions were simply decided in favor of the higher-ranking behavior. Rosenblatt and Payton purposed an alternative fine-grained control architecture composed of processing nodes which mapped many numerical inputs to a single numerical output. These nodes together could compose a complete behavior and, based on the weights of the connections, could be combined piecewise by higher nodes in the behavior structure.

Gat [25] argued for the division of controllers into three layers: controller, sequencer, and deliberator. The control layer determines reactive responses, the sequencer decides the order and priority of goals, and the deliberator identifies new goals sub-goals. The application of this research operates entirely in Gat's controller layer. Weyns [26] provides an extensive review of various distributed behavior controller for robots both theoretical and applied. The structures discussed by Weyns are all trees and the application of graphs in these systems is not addressed other than for mapping and navigation.

A popular application of multi-agent control for physical robots is the Robot Operating System (ROS) [27] open-source architecture which allows robotics researchers to write their own interoperable modules to operate their hardware. However, working with physical robots can be cumbersome if the research question is focused only on the composition of control. To avoid these physical limitations, many researchers have opted to experiment in simulated environments.

E. Unified Behavior Framework

Woolley and Peterson [28] developed the Unified Behavior Framework (UBF) to study the effectiveness of multiagent control structures for individual simulated robots in the Robocode environment. In the UBF, robotic control systems were structured as trees; the action recommendations originated at the leaves and were passed up through arbiters of various kinds until a single selected action emerged from the root. The agents at the leaves of the behavior trees sense features of the environment and recommend actions for the robot to take to their parent nodes. The internal nodes of the tree were arbitrating agents. These agents had differing methods to discriminate between the actions recommended to them and decide on the singular best action or best combined action to take and pass it up the hierarchy.

The UBF addressed many of Rosenblatt and Payton's concerns with the original subsumption architecture by implementing multiple types of arbiters. Some arbiters could imitate behavior subsumption, forwarding the recommended action as a unit, based on a pre-existing utility, or based on the confidence votes from the behaviors. Other arbiters could use these utilities or confidence votes to construct a new recommended action from pieces of its inputs. A Monte Carlo arbiter was included in the pool, which randomly decided between inputs. The variety of arbiters allowed for nuanced handling of conflicting goals. The UBF was not as granularly detailed as Rosenblatt and Payton's design. In the UBF, every behavior was an indivisible unit. The structure of the network could not influence the internal logic of the behaviors, only the emergent behavior of the robot.

Using this framework, Woolley combined the leave behaviors or ‘agents’ to create unique behavior structures for his robots. The UBF was demonstrated to possess capacity for rich representation of a wide range of behavior types as well as being modular and easily expandable [28]. These control structures or behavior trees could be large representing complex behaviors depending on their composing agents and structure. They were demonstrated to be both versatile in their representation and modular enough to work with genetic search.

The UBF behavior structures, since they were strictly organized as trees, were executed by calling for the recommended action of the root node. The root would in turn call for the recommendations of each of its child nodes and so forth until a leaf was reached. The leaf would resolve its recommended action based on the current state. Any historical state information existed only inside individual agent memory. The polling process propagated through the tree via a depth-first search progression. UBF’s tree behavior structures in these works could be easily adapted to more complex graph structures with some modifications.

Self-Organizing Behavior-Based Systems

A. Genetic Programming

Genetic programming is a search technique based on principles of mutation and adaptation in biological genes. It seeks an optimal solution to a problem by iteratively testing a population of solutions and modifying them to encourage the reproduction of highly-performing features [29]. Genetic programming, like all search-based AI methods, traverses a space of solutions to identify an optimal solution. The trick is to get the algorithm to traverse the space in the most efficient manner possible. To traverse the search space, a genetic program requires the solutions to be interpretable as genes which can be inherited and mutated. Credit assignment is applied in genetic programming by crediting each population member with a fitness rating [30]. The credit assignment is not made to individual components of a solution, but to the whole solution.

Genetic programming operates in the cycle of selection, duplication, and manipulation. Competitive fitness evaluations can be executed in a variety of methods spanning from an all versus all competition as used by Woolley [3], a tournament as used by Angeline and Pollack [30], a bipartite competition as used by Hillis [31], to a round-robin alignment as used by Trenaman [32]. Each method affords a different level of assessment overhead and a different type of environment for adaptation.

During the genetic manipulation phase of a genetic program, member solutions are transposed from one point of the solution space to another via several operators. The most common operators fall into two categories: mutation and crossover. Mutation is an operator that randomly alters a portion of the genome to a slightly different value or replaces it with an alternative from a pool of components. The crossover operation combines two or more solutions by taking sections from each of their genomes and splicing them together into new solutions.

Graph-based genetic programs have been researched in other studies. Mabu and Hirasawa [33] supplemented genetic programming with reinforcement learning to create a series of tile-world problem solvers. Hierarchically layered nodes were evolved by genetic programing and simultaneously trained by reinforcement learning to calibrate internal node parameters and inter-node connections. Poli [34] used genetic programming to discover non-cyclical graphs of solutions of the lawnmower problem. His technique used the tree structures representing the order of programmatic execution. The structures were pruned and grafted together to search for an optimal solution. Pereira, Machado, Costa, and Cardoso used genetic programming on a graph-based genotype by employing a crossover operation [35]. Their method copied the scheme of connections from a selected node in the first graph to its neighbors and applied that scheme to a selected node in the second graph. Spears and Anand conducted a review of crossover operations for genetic programming in which they determined that limitations of crossover effectiveness are significantly related to population size [36].

Cartesian genetic programming (CGP) represents the genome of a genetic program as a spatially arranged set of agents [37]. The arrangement of the agents in cartesian space defines their coordination. CGP allows for cyclical arrangement of agents in the graph. Harding and Miller used the method to develop and obstacle-avoiding robot controller [38]. This process was applied by Ryser-Welch to create scalable and human-understandable metaheuristics to generalize problem-solving intelligence.

"Chunking" refers to the practice of breaking down the problem and genetic solutions into sub-sections. Newell [39] used task-specific knowledge to decompose his task in the SOAR project. Once a solution was developed for a sub-goal, that sub-solution was condensed and locked. Locked components were used to compose future full solutions. Angeline and Pollack [30] implemented automated chunk discovery without prior task-specific knowledge by compressing and expanding sections of solutions to lock them as genetic components. The choice of which solutions to select for compression and expansion in their work was directed by fitness results, but the location and length of chunked segments was random.

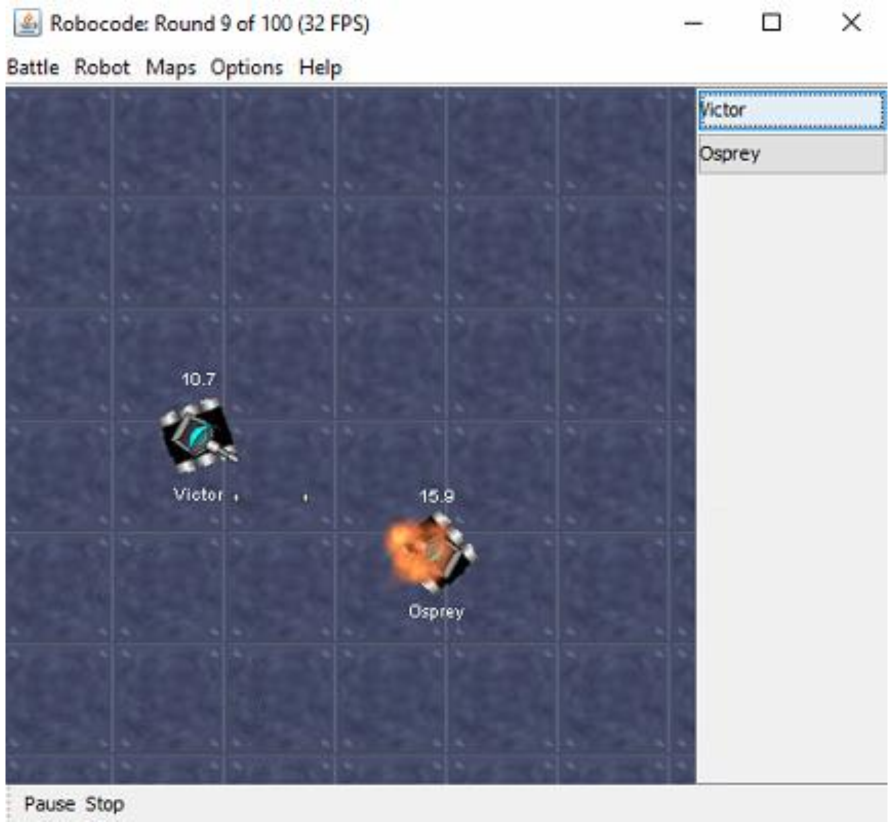
Robocode: Round 9 of 100 (32 FPS)
Battle Robot Maps Options Help
Victor
Osprey
10.7
Victor
15.9
Osprey
Pause Stop

Figure 1. A battle in Robocode fought by robots Victor and Osprey.

In previous research, Woolley and Peterson [3] developed an extension to the UBF which searched a solution space of behavior structures via a genetic program. They found that the genetic program was able to generate a robot 122% better than one designed by a human expert. Their work trained a group of behaviors structures by scoring them against each other in an all versus all battle. Crossover was completed by selecting a sub-tree of one behavior and grafting it into another at a randomly selected location. The top 10% elite members of the population were directly added to the next population without modification. Since the population of robots was locked to 10, the elite set of each generation contained a single robot. This "BestBoy" was selected from each generation to compete against a static, hand designed behavior named "Osprey" to benchmark the population's progress.

After 8 evolutionary runs of 1000 generations each, their results found that a population would reach a maximum benchmark and plateau close to the 200th generation. The resultant behaviors were all composed of 3-4 behaviors (2-3 leaf nodes and one arbiter). This was a proof of concept that genetic programming could work in the Robocode UBF environment. This research extends Woolley's by adapting his evolutionary algorithm to work with graph-behaviors and makes the significance addition of experiential simulated evaluation to evolve its behaviors. The implementation of this research's experiment was based on Woolley's design.

B. Simulation-Based Decision Making

Lessons from neuroscience have, in the past, led to advances in AI research [40]. In this vein, researchers have attempted to emulated imagination in their artificially intelligent agents. In many cases, this is realized as an agent which attempts to internally simulate the outcome of a potential action prior to choosing it. Racanière, et al. [41] developed an agent which used imaginative predictions of outcomes to decide actions. Their agent proved to be more robust than others when faced with problem misrepresentation. Briegel and De las Cuevas [42] presented an artificial intelligence which created fictitious scenarios and internally simulated its own actions and their results before making a decision. An essential concept for this internal simulation is an internal model of the world for the agent to imagine [43].

Gardelli, Viroli, and Omicini [44] discuss the utility and challenges of using simulation for decision-making in the development of self-organizing Multi-Agent Systems (MAS). These challenges include the influence of unknown variables on the agent's actions, the non-deterministic elements of the simulated environment, and the non-deterministic elements of other agents' behaviors.

Episodic memory was used by Nordin, Banzhaf, and Brameier [45] to generate a world model for a genetically programmed hierarchical behavior-based robot. They used a genetic program to learn the relationship from a set of past observations and actions to their effect. They retained memories of the environment by setting a 'childhood' phase of learning from which all memories were retained throughout the remainder of the robot's 'lifetime'. Random noise in the robot's behavior was introduced during this childhood phase to encourage a variety of foundational memories. In adulthood, additional memories were gathered and added to the experience bank in a rolling buffer. Through this method, they achieved basic behaviors such as obstacle avoidance in a static environment and wall following.

This research took a different approach. Rather than simulating a decision inside a behavior agent or using memories to construct an environment, this research evaluated each agent's performance in simulations of critical points from previous population experience. Simulation evaluation results were then used to inform the direction of evolution. The desired effect was envisioned to be like the human experience of dreaming. During dreams, the memory paths created through the day are re-activated and strengthened [46]. Associations are established between scenarios, responses, and their positive or negative consequences. The establishment of these associations help humans to learn the ways they should have acted differently or discourage them from attempting the same action again. A study conducted by Malinowski and Horton [47] found that dreams recalled by participants were largely autobiographical rather than episodic. The study concluded that dreams experienced by its subjects were conglomerations of highlights that were significant to the individual’s identity rather than rehearsals of whole events. The factors influencing the brain’s selection and assembly of these highlights was not addressed in the study. In this research, snips of very positive or very negative events are extracted and used for experiential simulated evaluation.

C. Reinforcement Learning

Reinforcement learning has been used in some studies to automatically organize the coordination of multi-agent systems. Kuyer, Whiteson, Bakker, and Vlassis used the max-plus algorithm to optimize the local actions and communications of neighboring traffic signals to optimize traffic flow [48]. In their method, the algorithm trained homogenous traffic light agents to optimize their conditional behavior and communication. The communication channels were predetermined; the timing and the content of the messages was learned. The algorithm achieved successful results even in cyclical coordination schemes.

Guestrin, Lagoudakis, and Parr developed a reinforcement learning technique for coordination graph structure learning [49]. Unlike other multi-agent reinforcement learning methods, their algorithm explicitly updates the graph structure to optimize coordination.

D. Experience Replay and Retention

Experience replay has been widely used to solve the problem of catastrophic forgetting in reinforcement learning. Catastrophic forgetting occurs when a learning agent or policy becomes too focused on the nuances of a narrow solution space and forgets features it may have previously learned during initial exploration [50]. Although even an especially forgetful agent can successfully find and hone a specific solution, catastrophic forgetfulness obstructs generalizability. Zhang et al. [5] describe an example in which a grid-world agent can reach a goal via two optimal paths. While the agent initially learned of both paths’ existence, in later iterations of training it forgot the second path and directed all movement to only one.

Feng et al. implemented a system for genetic algorithms to more efficiently solve a series of related problems by storing previously found solutions and using those solutions to bias the otherwise random generation of initial populations in subsequent problems [4]. In this way, the memories of previous problems provide a shortcut for similar problems. However, their method did not use experience replay during inter-generation population propagation.

To avoid forgetting, some reinforcement learning methods retain memories of past experiences which are used for periodic retraining of the policy. Beyond unlimited buffer, First In First Out (FIFO) retention, and reservoir sampling (RS), researchers have used many variations of experience retention [6]. These methods include Prioritized Experience Replay (PER) sampling [51], PER with Importance Sampling (PER-IS) [51], Dual Replay Buffer (DRB) [5], Multi-Timescale Replay (MTR) [52], and modifying FIFO to prefer surprise, reward, training distribution, or state space coverage [7]. These methods attempt to make intelligent decisions about which experiences are retained in the memory buffer or selected for retraining. PER retains experiences based on the element of surprise: the experiences that make the most difference to the RL policy are retained. DRB and MTR retain experiences at differing probabilities depending on how long ago they occurred. Both try to balance old and new experiences. This is akin to balancing exploration and exploitation. For many training environments, the research has found that FIFO and reservoir sampling are the most susceptible to catastrophic forgetfulness [6] [52] [7]. In practice, this forgetfulness appears as an initial spike in general fitness followed by a regression of the success measure in later episodes.

Summary

This work finds its foundations in the artificial intelligence studies of emergent behaviors in multi-agent systems. The subsumption architectures and its descendants form the background for what this thesis attempts to construct. The method of construction is influenced by works in genetic programming on tree and graph-structured genomes and the used of internal simulation for decision making. The next chapter details the aspects of these works which were directly applied as well as the specifics of their implementation.

III. Implementation & Methodology

This chapter presents details of design and implementation for the coordination graph structure, genetic program, and experiential simulated evaluation. It is clear from the trajectory of artificial intelligence design that distinct components or agents can be combined to create agents with wider capabilities. However, design of their combination is difficult. An automated method for construction of coordination graphs, could unlock accelerated progress in artificial intelligence systems design. An evolutionary system that can guide itself to construct such coordination graphs based on ancestral memory may step toward this goal more rapidly. By learning from current fitness assessment and past experiences, it may find valuable solutions more quickly by avoiding revisiting especially unfit regions of the genotypical space and by avoiding wandering away from promising lines of improvement.

To enable this graph-based coordination framework, the Unified Behavior Framework (UBF) was adapted to allow for unrestricted coordination graphs between behavior agents. The coordination graphs are represented genotypically—describing the unique interaction structure and strength between each component agent. The genetic representation and its manipulation operations are described. Simulated evaluations of candidate offspring in historically significant scenarios guide the direction of evolutionary development. The processes of experience extraction, analysis, and simulation are discussed.

Imagination of hypothetical scenarios is thought to be a special technique used by natural intelligences like humans to train more efficiently on a limited amount of execution data. In humans, the data may originate from previous experience of either the same individual or a different individual who relays their experiences. Traditional genetic algorithms rely heavily on random chance to navigate the genotypic solution space. A heuristic may be used to guide the random navigation to better options based on previously learned information. In addition to the standard fitness assessment and genetic manipulation phases, this research adds a simulated evaluation phase to the genetic program. Figure 2 describes the flow of graphs through the genetic program and simulated evaluations. Experiences are captured and stored during fitness assessment. Candidates for future generations are run through simulations of these scenarios to inform a heuristic of their fitness. To compare candidate offspring with their ancestors, the experiences containing the most drastic successes or failures are used.

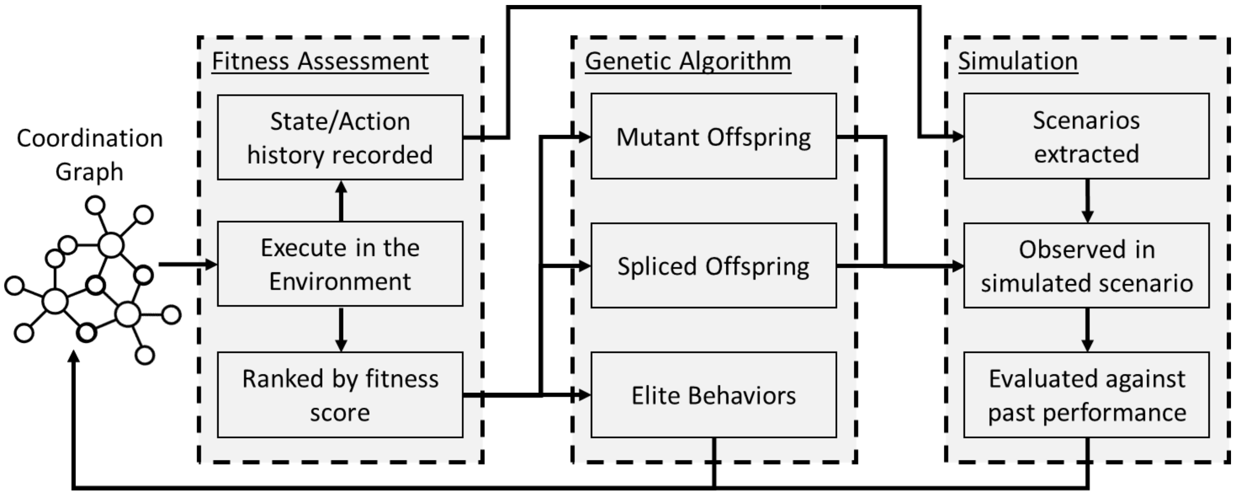

Figure 2. Process flow of genetic program with experiential simulated evaluation

If an offspring behaves differently than a previous poor performer, they score higher in the evaluation. Similarly, the offspring scores higher if it behaves similarly to a previous successful graphs.

The simulated evaluations are supported by an adaptation of the UBF and an associated development of a genetic program for the described behavior structures. The behavior framework was modified to allow for unrestricted coordination graphs. The developed genetic program adjusts the weighted, directed graph structure which describe the multi-agent coordination graph. Changes in the network result directly in changes to the arrangements' behaviors. Therefore, much of the genetic programming mutations and crossover involve graph manipulation techniques. Fitness for the genetic program is provided by the arrangement's accumulated relative scores provided by the environment.

Graph Communication Structure

This research adapted the Unified Behavior Framework (UBF) to create a flexible genotype. A behavior is a defined as any agent or collection of agents which intakes a state object and outputs an action. In this research all coordination graphs function as behaviors. The genotype is represented with an unrestricted coordination graph of various types of agents. The genetic manipulation process operates on the coordination graphs either through edits to the graph (mutation) or splicing graphs together (crossover). A central hub coordinates the communication of the coordination graph's agents (or nodes) working together according to a timing scheme. Figure 3 illustrates the interactions of a sample coordination graph.

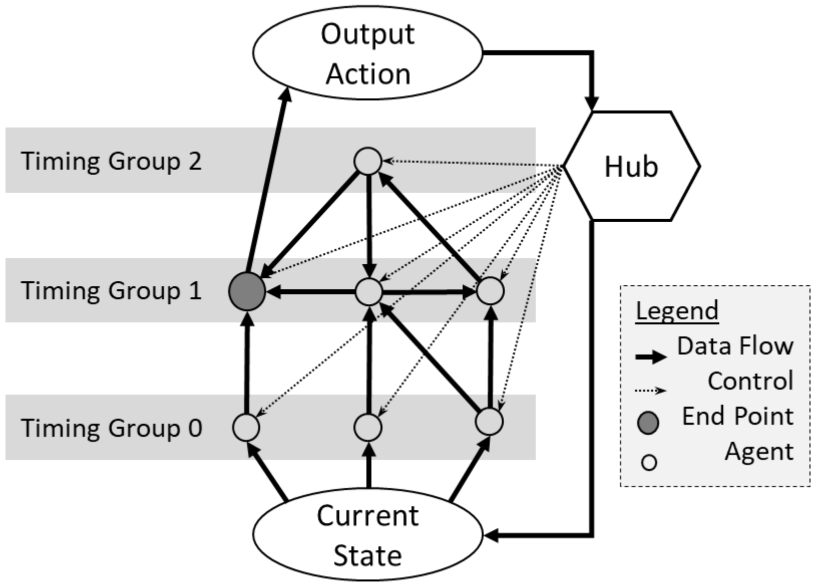

Figure 3. An example coordination structure denoting data channels, control channels, and timing groups.

The agents of the behavior graph are arranged into timing groups based on their shortest distance from a leaf node (i.e. a node with no inputs other than the current state). The hub controls the agents by prompting them to 1) generate an output and then 2) post that output for other agents to access. The hub guides each agent in a timing group through these two steps before moving on to the next timing group. If two agents are connected and in the same timing group, the receiving agent will get the sending agent's output from the previous tick. Once all timing groups have completed processing, the hub captures the output action from the end point agent. Specifics of the agents, the hub, and the timing scheme are presented in the following sub-sections.

A. Agents

Each node in the graph is a self-contained agent. Each agent maintains connections to and from other agents in the coordination graph with compatible data types. Most agents subscribe to multiple input messages; all agents generate a single output message. Agents request a list of their source agents from the behavior hub during the initial round of message generation.

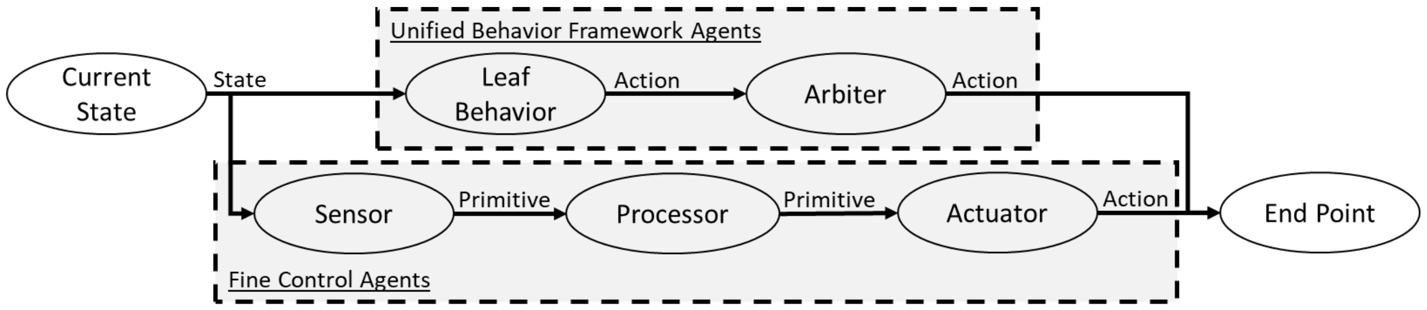

Figure 4. Generic flow of messages through behavior graph depicting the 5 types of agents: Behavior, Arbiter, Sensor, Processor, and Actuator.

In Robocode, five primary agent types were implemented. The arbiter and leaf agents were adapted most directly from the UBF behaviors. The arbiter and behaviors represent the two most complex agent types since they aimed to output complete action recommendations. In some environments, the action class is an array of component actions (e.g. turn radar/set velocity/fire gun). The actuator agents output an action with only one component defined.

Arbiters receive action recommendations and either decide between them or combine them to forward a single action depending on the implementation.

Behaviors examine only the current state and output an action recommendation.

Sensors are automatically generated based on get methods in the state class. Sensors examine a single aspect of the state and output a primitive value associated with that aspect. Examples include a getVelocity sensor which returns a double, a getIndex sensor which gets the current position in a string sequence, or a hasScannedRobot sensor which returns a boolean.

Processors take in one or more primitive types and output a single result. These agents may perform an arithmetic calculation, compare, or simply counted their inputs.

Actuators, like the sensors, are automatically generated based on the set methods in the action class. Each receives primitive inputs which are averaged; the action aspect is set to the resultant value. In contrast to the behavior and arbiter agents, actuators output an action with only a specific element of the action adjusted.

The sensor, processor, and actuator agents were inspired by a desire for fine control and novel evolution of behaviors similar to the work of Rosenblatt and Payton [24]. They are collectively termed the fine control agents (FCAs). The sequence matching environment used only behavior and arbiter agents with no FCAs.

All communications between agents are encapsulated in messages. The message class stores the value of the message as well as the message data type, the weight of the link over which the message is received, and a reference to the agent that sends the message. Agents use the weight information to influence which inputs achieve a greater effect on the output.

B. Hubs

Each coordination graph is centrally managed by a hub. The hub maintains the weighted coordination matrix for all agents in the graph. It is responsible for creation and population of the coordination graph, evolutionary manipulations, consolidation, and self-validation. Central management of each coordination graph is desirable since it allows for quick access to the entire network. Had a decentralized structure been implemented, every to the coordination graph would require graph traversal algorithms. Since much of this research involves manipulating and examining the structure of coordination graphs, graph traversal would have significantly impaired performance times.

The hub operates by calling each agent in its graph to update their outgoing message, but not publish it. During this call, if they have not already done so, the agents retrieve a list of their subscriptions from the hub. The agent then polls the posted message of each agent they are subscribed to and processes the input. After every agent readies their outgoing message, the hub directs them to make the message available. The hub maintains and updates a special state agent for serving the current state to leaf and sensor agents. Before returning the recommended action to the executive actor, the hub reports the current state and the resultant action to the recorder. At the end of execution, the hub directs the recorder to save its full memory to a file.

One agent from each coordination graph is selected as the end point. The end point is the extraction point for the entire network. The hub requires the end point to return an action, so it is always either an arbiter, leaf behavior, or actuator agent. If the graph contains arbiters, the first arbiter in the list of agents was selected. If no arbiters are available, the hub selects the first actuator to be the end point. Arbiters are preferable since they output a confluence of actions. Their outputs represent a richer composition of component actions. Leaf behaviors are preferred second snice they may or may not modify multiple fields in the output action, however, they will only receive input form the current state and never from the rest of the graph. Finally, actuators are accepted as the end point since they only modify a single feature of their output action. Following the updates of each node in the graph, the hub polls the end point for its recommended action and returns that recommendation to the over-arching agent as the selected course of action.

The hub checks the quality of its coordination graph through self-validation. A hub's coordination graph is valid if it 1) has an end point which can return an action and has at least one inbound link if it requires input, 2) is not empty, and 3) every link in the graph connects agents with compatible message types.

Hubs are initialized and their coordination graphs populated in the behavior factory. The factory can generate the hub populating it with agents randomly or generate the hub by reading in its coordination matrix from a .csv file. During random generation, the behavior factory randomly selects several agents (5-10 for Robocode and 25-50 for sequence matching) including at least one arbiter agent. For every possible pairing of agents, if the agents have compatible communication types, the factory then determines if a link should exist between the two agents with a 90% chance. If the link is added, the weight is set to a random value between 0 and 1.

C. Timing

Timing of network communication was a key consideration of the coordination graph design. Previous work with UBF enforced a strict parent-child relationship between agents with the tree-structured graphs. In that work, the root node was called on to generate an action. Before providing its answer, the root requested action recommendations from all its children nodes. The chain of requests propagated through the tree via depth first search. Action recommendations were then returned up through the tree: one computation cycle on all agents for each request at the root node. Cyclical connections can be beneficial to a coordination graph. An action that is recommended within a cycle in the graph may continue to loop in that cycle for an arbitrary amount of time. For example, in Robocode, a robot could take advantage of cycles by forwarding the recommended action to fire the gun even if it had momentarily lost active track of a target. In sequence matching, this behavior allowed graphs to repeat a character if no alternative was available.

To prevent the cycles introducing an infinite loop, a height approximation scheme was implemented. This timing scheme allows for synchronized message flow through the graph. In an earlier version of the graph-based coordination framework, the height bins were not delineated. All agents were treated as a single bin. During this time, it was found that messages took too long to propagate through the graph and consequently set the graph robot at a competitive disadvantage. For instance, a message to fire at a detected robot, if distant from the end node, may have required several ticks to reach the end node and be executed.

In the updated version, as agents are added, the hub updates their relative height in the graph and stores them each in bins. Nodes which obtain input directly from the current state node are binned to height 0. Starting with a queue of height-0 agents, all other agents are labeled with height through a breadth first search. To avoid infinite calculations of height caused by cycles, this calculation is only completed once. If two agents rely on each other for input, they each receive the other’s output from the previous tick as described in the following paragraph.

The hub manages each tick and directs execution in order of the binned height from lowest to highest. During a tick t the hub first updates its state agent which is responsible to serve up the current state to any agents that require it. Next, the hub directs each agent in the given bin to process their inputs and store their output privately. Each agent polls its list of publishers to retrieve their publicly posted outputs. Once an entire height bin processes their output message, the hub calls each agent in the bin to post their output for other agents to fetch either in the following height bin or during the next tick $t + 1$. A potential benefit of the system (in concert with the cyclic graph structure) is the persistence of information in the network.

Genetic Program

A. Program Description

The genotype of the genetic program is a weighted coordination graph of agents. A coordination graph is a composition of elements such that $(G: (A, F))$ where A is a list of component agents and F is a mapping from a source agent ($s \in A$) and a destination agent ($d \in A$) to the weight ($w \in \mathbb{Q}^{+} \cup \{0\}$) of their shared, directed connection ($F: (s, d) \rightarrow w$). If a connection does not exist between agents s and d, $F(s, d) = 0$. A valid graph must contain at least one agent which outputs an action and all edges must connect agents with compatible input and output types. The graphs are genetically modified by editing the nodes and edges of the graph or by splicing two graphs together. The phenotypic behavior emerges from the coordination graph by feeding in environment states and observing the resultant actions in the environment. Since it is possible for a genotypic description of a coordination graph to be invalid, validity checking is enforced by the hub. If a graph is invalid, the operation which created it is reversed and reattempted. The random nature of many genetic operations offers a chance for success on subsequent attempts. If the operation results in an invalid coordination graph 10 consecutive times, the genetic manipulation is aborted, and the original coordination graph is returned.

The genetic program is composed of three phases: In the *fitness assessment phase*, the population of behaviors execute in the environment and their scores are stored for later evaluation. In Robocode, each coordination graph of the generation fights the human-designed robot named Osprey for 15 rounds. In sequence matching, each graph executes only once because their execution is significantly less random. The best performer is identified and termed the "Victor".

In the *measurement phase*, the generation's Victor competes against Osprey for 100 more benchmarking rounds. The average of these 100 rounds is recorded to benchmark the progress. The benchmark score from each generation marks the progress of evolution over time. Scores are stored in the statistics file along with other measurements including approximate population diversity, coordination graph sizes, and the evolutionary method used to create each graph.

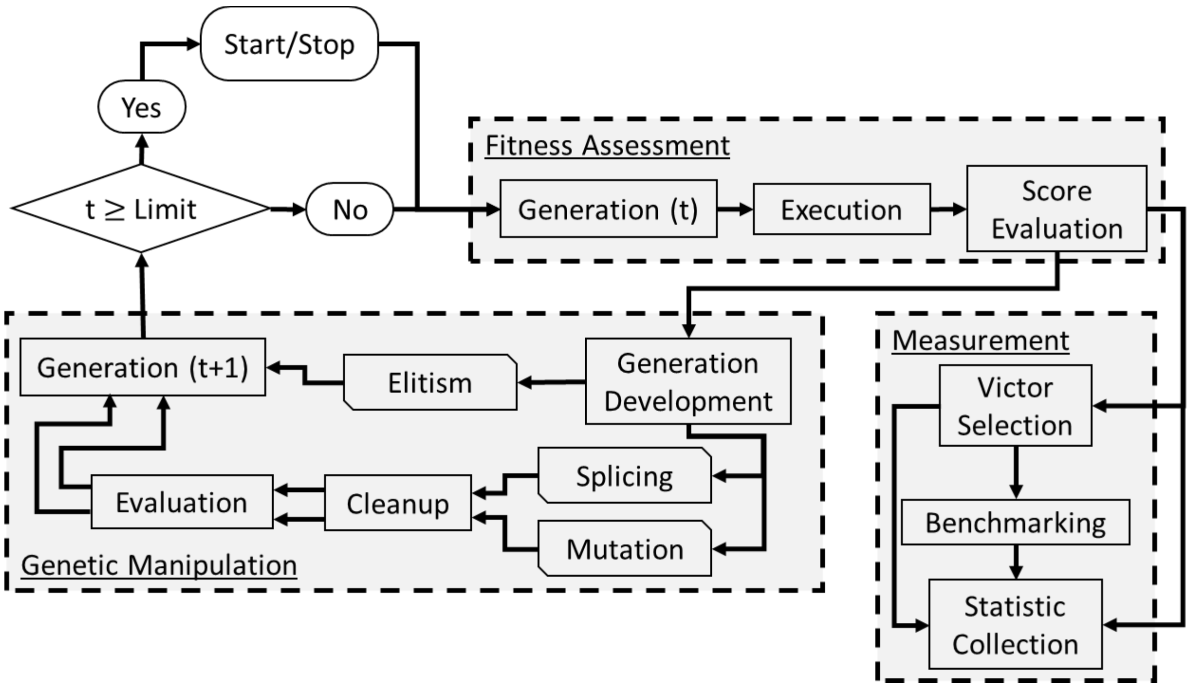

Figure 5. Flow of genetic program.

During the *genetic manipulation phase*, fitness scores are used to select arrangements for the next generation. Coordination graphs in the population are either directly transferred via elitism (10%), crossed over with each other with genetic splicing (0% or 40%), or randomly altered by mutation (50% or 90%). Parents of spliced and mutated graphs are selected probabilistically using stochastic uniform sampling.

Child graphs are randomly selected for clean-up. Clean-up consolidates duplicate agents and randomly removes agents if the behavior exceeds the maximum size. This is intended to reduce unnecessary complexity in graph structures and bound the search space.

The remainder of this section details the methods used in the genetic manipulation phase: elitism, crossover, and mutation. Behavior evaluation is described in the section on Experiential Simulated Evaluation.

B. Elitism

Elitism is employed to maintain some of the best graphs between generations. The elitism rate is set to 10% for each generation; the top graphs out of every generation are transferred directly to the next generation with no genetic manipulation. This elitism rate is high compared to other studies due to the stochastic nature of the fitness assessment rounds. In Robocode, a robot's initial placement on the board and orientation of the gun and radar can make a significant difference in performance for that round. The number of rounds is increased as a countermeasure against this random noise. A high elitism rate is used additionally to avoid accidentally discarding an effective behavior which can slow evolutionary progression. A collection of elite graphs represents a valuable pool of genetic material. If the elitism rate had been lower, the evolutionary progress of the population would have been more erratic. Between every generation, the population would have a higher probability of losing its best performer and populating the entire generation with genetically diverse graphs. With more diversity, there is an increased chance of developing a beneficial mutation in each generation, but since the majority of the possible mutations are not guaranteed to be beneficial, it is more stable to retain high performers between generations.

C. Crossover

Crossover genetic manipulation is the technique of combining genetic material from multiple parent behaviors to create the child. In the literature of genetic programs with graph-based agents, crossover is not common. The implementation of a crossover operation is not obvious since the genotypes are non-linear.

Genetic crossover is used to manipulate the genetic population. When enabled, the crossover rate for each population is 40% ($C = .4$). After all graphs in generation g are scored, $2 \times C$ of the population is selected probabilistically by relative score to be genetic donors. These donors are then randomly picked from the donor list in pairs, labeled graph A and graph B, and grafted together to form new progeny which are added to generation $g + 1$.

Graft point selection is the first step of the grafting process. The graft points are the nodes in each coordination graph at which the two are spliced together. The graft point of graph A is replaced by the graft point of graph B. The primary requirement for these points is that they have compatible input and output message data types. The graft points are selected randomly until a successful match is found or until the number of random attempts equals the number of nodes in the respective graph. Graft point selection fails more commonly if the coordination graph sizes are small. If graft selection fails, then the splice method returns graph A unaltered.

Once compatible graft points a and b are identified, the genetic program replaces a with b inside of graph A's coordination graph. If a is a bottleneck between two portions of A, then this action eliminates the portion of A not containing the end point which is denote A'. The replacement stitches all nodes previously linked to a to the newcomer b. Next, the subgraph B' of B is identified containing only nodes which contribute to b's inputs. Depth first search rooted at b is used to identify B'. With equal chances, links of B' are added or not added to A. The resultant graph C consists of $(A - a) \cup (B' + b)$. The size of C can range between 1 and $(|A| - 1) + |B|$ nodes depending on the location of the graft points. In most cases, the C is composed of two highly connected graphs joined by a single bottleneck node. Following crossover, if a coordination graph contains more than a maximum number of nodes, it is run through a clean-up process until it is under the limit.

The splicing method allows the genetic program to rapidly navigate the solution space using entire sections of successful coordination graphs in various configurations. Though it cannot generate novel genetic material, it combines the existing material faster than mutation can. Though not an explicit application, the splicing method imitates chunking. The intention of its implementation is to allow the program a path to take blocks of the graph which might represent a beneficial behavior and combine it with another graph with alternatively successful behaviors to achieve a graph better than either of its genetic donors. While splicing allows the genetic algorithm to make large steps across the genotypic space, the steps have the potential to be erratic. In some experiments, crossover is excluded from the genetic manipulation options.

D. Mutation

Mutation is the second type of genetic manipulation employed in this work. It encompasses four types of manipulation: link mutation, node mutation, node replacement, and weight mutation. In each generation g, the mutation rate is 50% (90% if crossover is disabled). Graphs are selected probabilistically based on their relative score, copied, mutated, and added to generation $g + 1$. A graph chosen for mutation had a 10% chance of undergoing mutation a number of times equal to the number of agents in the graph. For example, a graph with 20 agents would have a have a 10% chance of mutation 20 times resulting in an expected value of 2 mutations. During each mutation event, each graph has an equal chance (33% each) to be processed through either link mutation, node mutation, or node replacement.

If selected for link mutation, a random connection in the graph can either be added or removed. If the resultant mutant is still a valid graph, it is returned. Otherwise, a new link is randomly chosen. If the graph is selected for an additional link, the program randomly chooses two unconnected nodes in the graph until it finds two which are compatible, then it adds a link. Again, if the mutant graph is invalid, the program resets the graph and tries again. Attempts repeat until they the program fails more times than twice the number of nodes in the graph.

Graphs mutated with node mutation have a 50:50 chance to have a node either added or removed. Node removal randomly selects a node, removes it, and checks for behavior validity. Node addition fetches a random node from the behavior factory and adds it to the existing graph with a 90% chance of connection with compatible nodes. If the process fails to pass the validity check, mutation is reattempted up to 10 times. After the 10th failure, the program returns the original unmutated graph.

Node replacement mutation operates similarly to node addition/removal. A node is generated randomly by the behavior factory and a node with the same input and output types is selected from the graph. The selected node is replaced by the newly generated node and all previously existing links are stitched back together.

Finally, after undergoing one of the three structural mutation types, all weights of all the edges of a mutants are mutated. For each edge e in the coordination graph with original weight w, the weight of e is randomly adjusted to between $w \times 0.75$ and $w \times 1.25$. The expected value of each weight mutation is the original weight w leaving long-term growth or reduction of the weights to be controlled by the fitness evaluations.

The mutation methods in this work were designed to allow small changes in the behavior graph. Ideally, the original graph would be moderately successful and one or more of the mutant graphs would either remove some negative characteristic or add a beneficial one to the behavior. Mutation is the primary mode of traversing the genetic search space since the mutation rate is higher than any other method.

E. Cleanup

After crossover, if the behavior exceeds the maximum number of agents, it undergoes a cleanup process. Cleanup traverses the behavior graph starting at the end point via a depth first search to identify and remove any disconnected portions of the graph. It then identifies and consolidates any duplicate leaf or sensor nodes. Cleanup runs only when the maximum node limit is exceeded to allow mutations with disconnected graphs or duplicate leaf agents to compound for a while before being removed. This technique is employed based on the principle that often two or more components must evolve before either of them can be beneficial.

Experiential Simulated Evaluation

A. Storing and Evaluating Experiences

To enable experiential simulated evaluation, each hub records the entirety of its graph's execution. Each graph's observed state and the response action are sent to a recorder after each tick. The recorder packages the state and action pairs (termed moments) into a sequence of moments (termed a memory). Upon the end of execution, the hub signals to the recorder to save its accumulated memory as a file and store it in a folder for unprocessed memories. This process is illustrated in Figure 6. 1) The graph's behavior hub sends the observed state and response action to a record, 2) the recorder concatenates the state/action pair into a moment and appends the moment to the full memory, finally 3) when the graph completes execution, the recorder saves the memory to a file.

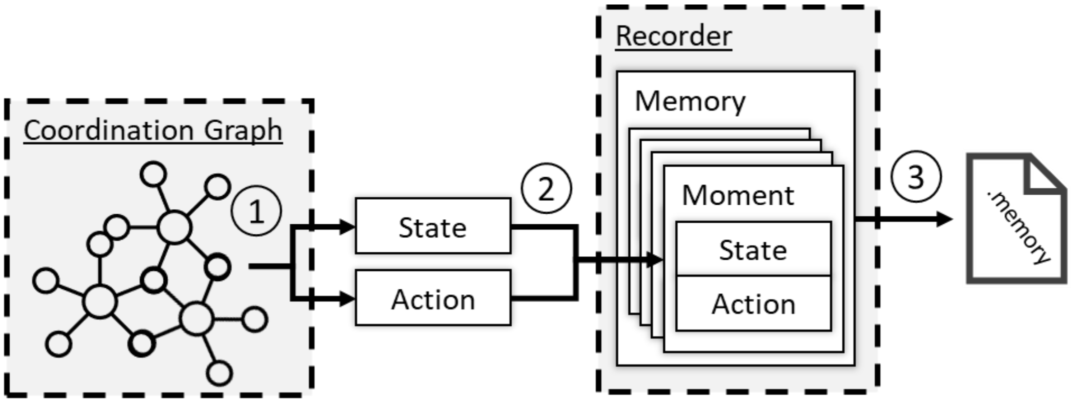

Figure 6. The process and structure of memory storage.

After all the executions are completed and before the genetic manipulation begins, a memory analyzer reviews all unprocessed memories from the past generation. The analyzer's primary role is to identify the most significant memories or snippets of memories. Viewed collectively, these experiences form a biographical picture of what a behavior should and should not do.

The first method saves the memory of the execution in its entirety. The second method splits the memory into highlights based on the magnitude of rises or falls in the actor's score. Highlights are identified using a greedy algorithm which lists every possible interval within a memory and sorts them by greatest significance, then by greatest duration, and finally by latest occurrence. The algorithm then evaluates each interval in order and returns the first two non-overlapping intervals with durations in a given range. These intervals are used to extract the highlights from the original memory along with a short preamble. The preamble is the sequence of states before a selected highlight. The preamble is used to prime a graph before simulation. The length of the preamble was selected separately for each environment based on features of the environment and the expected size of the graphs. For each execution period, each graph's two most significant highlights are extracted and saved in the collective experiential memory.

B. Experience Analysis

After each execution in the fitness assessment, graphs send their complete memory of state and action pairs to a file. These experience files are analyzed after fitness assessment and before experiential simulated evaluation. The analyzer calculates the score at during each tick to identify the moments of success or failure.

At that point, if the highlight extraction is disabled, the full memory is saved to the experience pool for significance sorting. If, however, highlight extraction is enabled, with the RS calculated for every moment in the memory, the analyzer identifies the significance (S) of any memory or sub-sequence of memory. Significance is calculated for a memory starting at a moment at index a and ending at a moment at index b by examining the relative score at those moments (e.g. RS_a).

With heuristic values set for each moment, the analyzer calculates the significance (S) of any segment of the experience from moment a to moment b with the following equation.

$$S = \frac{RS_b - RS_a}{b - a} \tag{1}$$

The significance equates to the slope of relative score over time. If relative score increases, the significance is positive and if relative score decreases, then negative. The significance indicates if a segment of a memory represents a good experience or a bad experience and to what degree. Figure 7 illustrates the qualities of a memory in relation to is significance rating on a nominal number line.

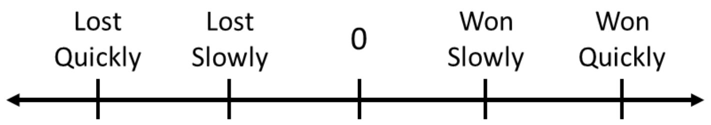

Figure 7. Memory significance rating illustrated on a nominal number line.

Early testing found that graphs can be more reliably compared against a memory segment if its actions are rerecorded in the context of only that scenario and its preamble. Therefore, once the highlights are identified and extracted, all the actions associated with that highlight are rerecorded through a simulator and a clone of the graph which originally generates the memory.

If highlight extraction is enabled, the analyzer scans over all segments of the experience to find only those segments with the most positive and most negative significances. These highlights are extracted and added to the experience buffer. If highlight extraction is disabled, the full experience is added to the experience buffer.

C. Experience Retention Methods

The large number of highlights in collective memory, if left unchecked, eventually grows to be unmanageable and impractical for future graph evaluations. At any time, no more than 100 scenarios are retained for evaluation. Two methods of experience retention are implemented and compared in the experimentation: retaining the most significant only (by absolute value, by most positive, or by most negative) or retaining the most recent only.

Significance retention sorts all available experiences by significance and kept experiences based on their absolute value of the significance, so equal numbers of good and bad experiences are kept. This is termed balanced retention. Two other methods of balancing alter the bias to keep more negative or positive experiences. Keeping more negative experiences is termed “exploration” and keeping more positive “exploitation”. Two scopes of experience retention were tested: *all-generation retention* keeps experiences from the whole buffer while *single-generation retention* never retains any experiences older than a single generation. Experience pool size under single-generation retention depends on the number of repeated fitness assessments and the size of the population. For example, if the population size is less than 100 and each graph only executes once, there will be less than 100 experiences in the experience pool.

D. Experiential Evaluation

The retained experiences inform the direction of the genetic program through simulated experiential evaluation. By intuition, if a previous graph performs poorly, future generations should be encouraged to perform different actions from those which lead to the unfavorable outcome. Conversely, if a previous graph had a good experience, future generations should be encouraged to take similar actions in the same state.

During genetic manipulation, if experiential evaluation is enabled, the genetic program generates 3 times as many graphs as are required for populating the next generation. The coordination graphs that score in the top third of evaluation scores continue to the next generation; all others are discarded. The simulator object facilitates these evaluations. The simulator takes as parameters a memory (the scenario) and a behavior (the participant). The simulator first feeds the participant the series of states from the scenario's preamble but does not record the participant's output actions. The purpose of the preamble is to prime the behavior with the context of the scenario. During initial testing, it was determined that graph behaviors could be compared more reliably if a preamble is used to propagate some of the prior state information through the deep or cyclic portions of the coordination graph.

The simulator next sends the state information from each experience in the buffer to the graph being evaluated and records the actions it returns. The actions are compared with the original graph's action stream using the average "distance" between each action in the sequence. All action distances range from 0 to 1. A pilot study determined that the action distance evaluation does not always return a 0 distance for the same graph executing the same scenario since many graphs contain stochastic behaviors.

All graphs are evaluated on the set of available experiences by an evaluator. The evaluator assigns evaluation score (S) to each graph (g) based on each experience (e) from the set of all stored experiences

(E), its distance from the original action sequence for in that scenario ($d(g, e)$), and the significance of each experience ($s(e)$).

$$\mathrm{S} = \sum_{\mathrm{e} \in \mathrm{E}} -(2d(g,e) - 1) \times s(e) \tag{2}$$

With this formula, graphs which perform similarly ($d(g, e) < .5$) to the scenario accrue negative scores if the scenario represents a negative experience but accrue positive scores if the experience is positive. Conversely, a negative score is accrued if the graph performs differently ($d(g, e) > .5$) from a positive experience and positive score is accrued for a performance differing from a negative experience.

Summary

This research integrates recorded experiences of past iterations of an agent coordination graph to guide future coordination graphs in a more constructive direction. Simulated experience evaluation can cut down on costly real-world assessments and create better behaviors more quickly. The program described in this chapter constructs graph-based multi-agent systems from a pre-existing set of agents. The components are assembled in a graph to generate more complex behaviors. The constructed behaviors are evaluated for fitness and genetically altered and combined. In some versions of the experiment, candidate behaviors are first evaluated on simulations of past experiences. The results of these retention methods and of the control group are presented and analyzed in the next chapter.

IV. Analysis and Results

This chapter presents results of the experiential simulated evaluation algorithm. Experimentation sought to determine if progress toward a more fit solution could be accomplished more quickly with experiential simulated evaluation enabled as compared to the baseline genetic program. The experimental design including the execution environments, the process overview, tested parameters, and key result measures are described. Parameter configurations for the baseline were determined through experimentation and comparison. The effect of experiential simulated evaluation with variations were tested. Finally, the experimentation results are detailed. Several populations of graphs were evolved with various parameters to evaluate the effectiveness of the techniques described.

The primary experiment of this research focused on the comparison of progress between populations evolved with and without experiential simulated evaluation. Could internal experiential simulation be used to guide a population to a target level of fitness in fewer rounds of fitness assessment? In much of artificial intelligence training and development, the costliest portion of training is gathering data from actual execution. The cost of execution may be measured in time, money, safety, or others. Identifying methods to get more value out of the execution data can help AI projects train more efficiently. Secondarily, this research sought to compare methods of experience retention for simulation to determine benefits and consequences of each. Third, comparisons are made between the genotypic representations of behaviors and their associated phenotypic expression, which demonstrates consistent functionalism regardless of the structure

Environments

The graph-based coordination framework of this research was applied to two execution environments: simple sequence matching and the Robocode game.

Sequence matching is a simple, originally implemented environment which stores a secret string and rewards agents for correctly guessing each character of the string. The environment is like the Master Mind game, but instead of giving one score after the full guess has been made, this sequence matching environment provides an update to the score after each guess. The state object of this environment is the index of the string that is to be guessed. The action object is the character which the behavior guesses. In this environment, the genotype remained a coordination graph of sub-agents, but the phenotype was the string of characters the graph produced. The sequence matching environment was not as complex as Robocode. All graphs in this environment were rewarded for producing 26-letter strings close to the English alphabet.

The Robocode project was started in 2000 and has since grown as a tool to teach and encourage programming skills [53]. Robocode is a simulation world in which miniature tank robots compete against one another to determine which is the most fit. These robots are often programmed by hand and are pitted against each other to find the most successful one. Robocode has proven to be a helpful environment for other AI researchers. The research questions of AI can be applied to these robots in many ways. Planning, observation, coordination, decision-making can all be modeled in the simulation. In the environment, robotic tanks navigate within a two-dimensional world and may move forward or backward, rotate their chassis, radar, or gun, scan with the radar, or fire with the gun. Obstacles may be placed in the environment. Walls, obstacles, other robots, and bullets can be detected with the radar. Robots are damaged (a.k.a. lose energy) when they are hit by a bullet or another robot, run into a wall or obstacle, or fire a bullet. They can regain some energy upon successfully hitting an enemy with a bullet. The round ends when the energy of one of the two competing robots drops to 0.

Table 1. Sequence Matching Agents.

Arbiters	**Behaviors**
Fowarder	Random
HighestActivation	Single
HighestPriority	
MonteCarlo	
RandomRepeater	
SimplePriority	

A. Environment-Specific Agents

Fewer agents were used for in sequence matching compared to Robocode. Fine Control Agents (FCAs) were not used in the sequence matching environment. Most of the arbiters were adapted to work with sequence matching. The fusion arbiters were not adapted since sequence matching only had one aspect to its actions, therefore no fusion was required.

Table 2. Robocode Agents by Type.

Arbiters	Behaviors	Sensor	Processors	Actuators
Activation Fusion	Aim	getBearingToObject	Adder	setAllStop
Command Fusion	Charge	getBoardHeight	Divider	setFireGun
Highest Activation	Dodge	getBoardWidth	Enabler	setGunRotation
Highest Priority	Fire1	getDistanceToObject	EqualChecker	setRadarRotation
Monte Carlo	Fire2	getEnergy	GratherThanChecker	setScan
Priority Fusion	Ram	getGunHeadingDegrees	InputCount	setTurnRate
Random Repeater	RandomTurn	getGunHeat	LessThanChecker	setVelocity
Simple Priority	RandomVelocity	getGunRotation	Multiplier	setVote
Utility Fusion	ReturnFire	getHeadingDegrees	Subtractor	
	ScanLeft	getHeight		
	ScanRight	getNumberOfOthers		
	ShortRngFire	getNumberOfRounds		
	SittingDuck	getRadarHeadingDegrees		
	SniperFire	getRadarRotation		
	TrackFire1	getRobotHasDied		
	TrackFire2	getRoundNumber		
	TrackFire3	getTimeUpdated		
	Wander1	getTurnRate		
	Wander2	getVelocity		
	Wander3	getWidth		
		getX		
		getY		
		hasBulletHit		
		hasBulletHitBullet		
		hasBulletMissed		
		hasHitByBullet		
		hasHitRobotEvent		
		hasScannedObject		
		hasScannedTarget		
		isDead		
		isHitByBullet		
		isHitByRobot		
		isHittingObject		
		isHittingRobot		

The behaviors for sequence matching were Random and Single. Random agents would produce a random character regardless of their input. Single agents would output a single, pre-determined character if they observed a pre-determined state. For example, a Single agent named Single_2_e would output the character 'e' if given a state 2. The factory created a version of the Single agent for every possible character for every possible index resulting in 26^2 possible Single agents. The agent types for sequence matching are listed in Table 1.

The sub-agents for the graphs were unique in each environment. The agents used in Robocode are listed in Table 2.

Most of the Robocode behavior agents and all the arbiter behaviors were adapted directly from Dr. Woolley's work with the UBF. An original behavior agent for aiming was added for this work. All the processors were written originally for this research. The abstract sensor and abstract actuator agents were written for this work. The sensor and actuator agents were generated automatically by the factory based reflectively on methods in the state and action classes.

B. Action Distance

The difference or distance between two actions was calculated to determine who similarly or dissimilarly two graphs behaved. In sequence matching, the raw distance was the number of steps in the alphabet between the two characters. The number of steps was the raw score. The normalized action distance was calculated as the square root of the raw score divided by the maximum possible raw score. Taking the root of the normalized score allowed a letter near the correct one to be awarded points, though significantly less than would have been if the letter were exactly right. As the guessed letter moves farther from the correct character, the decrease in reward lessens. Figure 8 illustrates, as an example, the action distance between each character and the character 'g'.

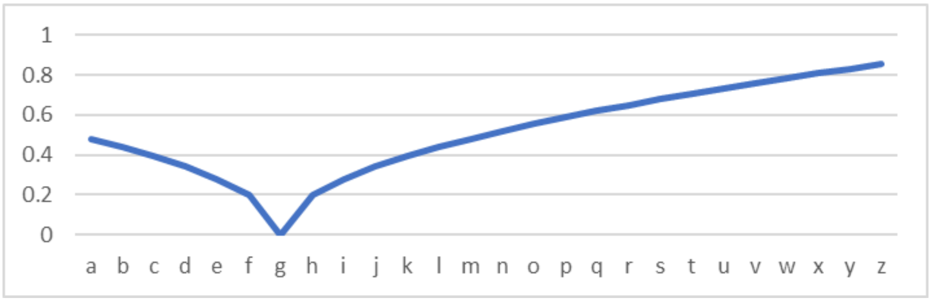

Figure 8. Action distance between each character and the character 'g' in sequence matching

In Robocode, the action object is an array of eight action components. Each component controls an function of the robot: vote, velocity, turn rate, radar turn rate, gun turn rate, scan enabled, all stop enabled, and fire power. The action distance calculation determines the distance between each of the eight aspects of the action. Action aspects are considered more distant if one is set and the other is not. Additionally, aspects are considered more distant if one value is positive and the other is negative. Otherwise, distances are scaled based on the difference between the two values. The overall action distance is the average distance between each component action.

C. Score Calculation

For the purposes of assigning a significance to each experience, the analyzer must calculate the score of a graph after each tick. In the sequence matching environment, the graph is informed of its score after every action. A maximum of one point was awarded for every correct character. If the character was incorrect, the graph was awarded one minus the action distance between the given answer and the correct answer.

In Robocode, the behaviors are aware of their own score, but their opponents score is unknown unless they are actively scanning the opponent, so the score is calculated heuristically. To measure the relative score (RS) during a moment for graph (g), the analyzer considers the energy level of g ($e(g)$), the number of observed opponents (n), and the energy levels of each opponent ($e(o_i)$).

$$RS = e(\mathrm{g}) - \frac{\sum_{i=0}^{n} e(o_i)}{n} \tag{3}$$

A robot may win a battle with any relative score greater than 0 or lose a battle with any value less than 0. Relative scores are not limited to a range since robots can achieve energy levels greater than their starting value (100.0) if they shoot their opponents efficiently. The number of opponents is reported as a feature of the given state. Opponent scores are assumed to be 100.0 unless an opponent has been scanned. If a scanned event is reported in the memory, the analyzer documents the last known score for that opponent and uses that value in the relative score calculation. This method generally means that robots lacking a scanning behavior have a lower relative score than they would otherwise thereby biasing the heuristic toward scanning robots. This bias is considered acceptable since scanning is already an advantageous behavior.

D. Environmental Parameters

Several other parameters were selected to be different between the two environments. Table 3 lists the values used for these parameters. Minimum and maximum initial agent limits were set for the factory when developing random graphs for each initial population. A separate maximum agent limit was used to determine prevent the graphs from becoming too large. Highlight duration ranged dictated the minimum and maximum lengths for highlights which could be extracted from experience recordings. Highlight lengths are measured in ticks. Each environment was run for a different number of generations based on how quickly each converged to a solution and computation time.

Table 3. Execution Environment Paramters

Parameter	**Sequence Matching**	**Robocode**
Minimum Initial Agents	25	5
Maximum Initial Agents	50	10
Maximum Agents	200	30
Highlight Duration Range	1-20	2-200
Generations	750	500
Generation Size	20	20
Preamble Length	0	10

Experiment Design

A. Parameters

The experiments studied the effect of changes over three parameters:

Retention Method defines how experiences in the buffer are either discarded or retained to ensure the maximum buffer size is not exceeded. Experience retention is run immediately following fitness assessment. The two methods tested were single generation and all generation retention. All-generation experience retention searched the experience buffer to find the most significant experiences of all time. Alternatively, single-generation retention discards all experiences that are older than a single generation. The most significant experiences from that single generation are retained.

Highlight Extraction is a technique of identifying and isolating only the most drastic portions of an experience and adding only that experience segment the buffer. Highlights were extracted from the experiences based on the segment of maximum change in the graph's score. The most positive and the most negative highlight were extracted from the full experience and saved to the experience buffer for future evaluations. In the alternative, full experiences were saved from fitness assessment and replayed during simulated evaluation.

Exploitation or Exploration are generally known tendencies of a solution-seeking algorithm to focus attention on a known good solution or to search instead for novel solutions, respectively. The differentiating factor was the percentage of good or bad experiences kept during retention. In this research, the experiential simulated evaluation receives a set of randomly genetically altered graphs and returns those with the highest evaluation score. The evaluation process is a modifier to the genetic program which biases the next generation in a direction. The method to derive the direction of the bias can be altered by choosing to keep more negative or more positive experiences in the experience pool. There are two components that positively influenced a graph's evaluation score: its similarity to a good experience and its dissimilarity to a bad experience. Similarity to positive experiences are equivalent to exploitation of a single solution while exploration is represented by preferring performances dissimilar to bad experiences. The exploratory method favors graphs which perform differently from those in the previous generation which lost by the greatest score differential or those which lost the most quickly. The explorative component biases the algorithm toward any randomly altered graph to on the condition that it exhibits behaviors dissimilar to previous failed strategies. The exploitative method prefers graphs which perform similarly to experiences from the previous generation where the graph won by the greatest margin or those which won the most quickly. By default, the exploitation and exploration foci were weighted equally. When tested, the balance was changed from 50/50 to 95/5.

B. Result Measures

The primary measurement of each generation success is the relative benchmark score. In Robocode, the benchmark score is the score earned by the victor robot divided by the total score earned by the victor and Osprey. In sequence matching, the benchmark score is simply the score of the victor since execution in that environment is deterministic. From this point, the relative benchmark score is simply referred to as the score.

Several configurations of parameters were tested and compared. Each test consisted of 10 population evolutions. To reduce noise in the benchmark scores, a centered rolling 9-generation average was used to represent each evolution. The 10 populations' smoothed results were then averaged to approximate the success of each configuration. Raw benchmark scores, population diversity, connectivity, graph size, and evaluation scores were also recorded.

Since the goal of the research was to compare the efficiency of including experiential simulated evaluation to guide a genetic program, the primary measurement method was a graphic representation of the evolutionary progress of the compared populations over the number of rounds required to reach that level of progress. Efficiency was defined as the amount of progress made between two generations. The graphic representation of evolutionary progress can be distilled into an average rate or progress measurement by finding the average slope over the course of execution from the population starting point to its maximum.

By the given measure average rate of progress, a more efficient population will have a steeper slope between its initial and peak scores when graphed over generation number. This measure can be helpful but may be misleading since a population may achieve a low peak score early in the learning process while a competitor may achieve a significantly higher peak score over a longer number of generations. To account for this, a progress inflection point is identified. The slope between the population's initial score and the progress inflection point is the initial rate of progress.

The following features were measured to compare the quality of each solution:

Terminal Score (TS) is the average score of the population over every time step after the population's maximum score.

Average Rate of Progress (ARP) is the change in a population's score over the course of the population's evolution.

Progress Inflection Point (PIP) is the point which has the greatest difference between a population's score and a line with a slope equal to the population's average rate of progress.

Initial Rate of Progress (IRP) is the slope from a population's initial score to its progress inflection point (PIP).

Progress Instability (PI) is the difference between a population's maximum score and the minimum of all subsequent scores.

C. Experimental Process

To answer the research questions, several populations of graph behaviors were evolved using experiential simulated evaluation techniques. In each environment, a baseline collection of populations was evolved with no simulation evaluations. No experiences or memories were collected from the graphs during execution. The direction of evolution in this category was driven solely by the fitness assessment scores and random genetic manipulation. In Robocode, the crossover operation and the use of FCAs were tested to determine the best combination. The results with the best combination of these two techniques was used as the baseline for all following experimentation. In sequence matching, FCAs and crossover were not used.

To determine the best retention method, four configurations were then run implementing a combination of highlight extraction or full memory storage and of all-generation retention or single-generation retention. The best configuration of these parameters was then used to test the utility of exploitation or exploration techniques. Finally, several of the final, most successful coordination graphs were examined along with their expressed behavior to determine if observations could be made on the relationship between the genotype and its phenotype.

Baseline

To usefully compare the effects of experiential evaluation, a baseline was first established. Based on pilot sampling, the utilities of Fine Control Agents (FCAs) and crossover genetic manipulation were questionable. In Robocode, four tests were run comparing the evolutionary progress of these techniques. No population evaluated for the baseline used experiential simulated evaluation; they relied only on fitness assessments to guide genetic selection. The key measurements for each of the four methods are displayed in Table 4.

Table 4. Measurements of the Robocode Baseline Configurations

	Crossover		No Crossover	
Fine Control Agents	Terminal Score:	0.6726	Terminal Score:	0.7102*
	Initial Rate of Progress:	0.0049*	Initial Rate of Progress:	0.0019
	Progress Instability:	0.0033*	Progress Instability:	0.0075
No Fine Control Agents	Terminal Score:	0.5817	Terminal Score:	0.6342
	Initial Rate of Progress:	0.0029	Initial Rate of Progress:	0.0022
	Progress Instability:	0.1409	Progress Instability:	0.0615

* best result for the given measurement

The configuration with the highest terminal score included the use of FCAs and excluded the use of crossover operations. Not only does it finish with the highest score, this configuration also out-competes every other configuration from about the 375th generation onward. This configuration, however had an exceptionally low initial rate of progress. The progress achieved by this solution was both slow and steady. The configuration with both FCAs and the crossover operation is also worth examining for its high initial rate of progress and its stability, however the terminal score is the most important factor when determining the best method.

Both trials using FCAs exemplified more gradual growth over time. This growth tended to progress rather than plateau or degrade. This may be explained that, with FCAs, there are more ways to incrementally improve a strategy as opposed to using only pre-designed behaviors which have limited effective combinations and represent a courser solution-space.

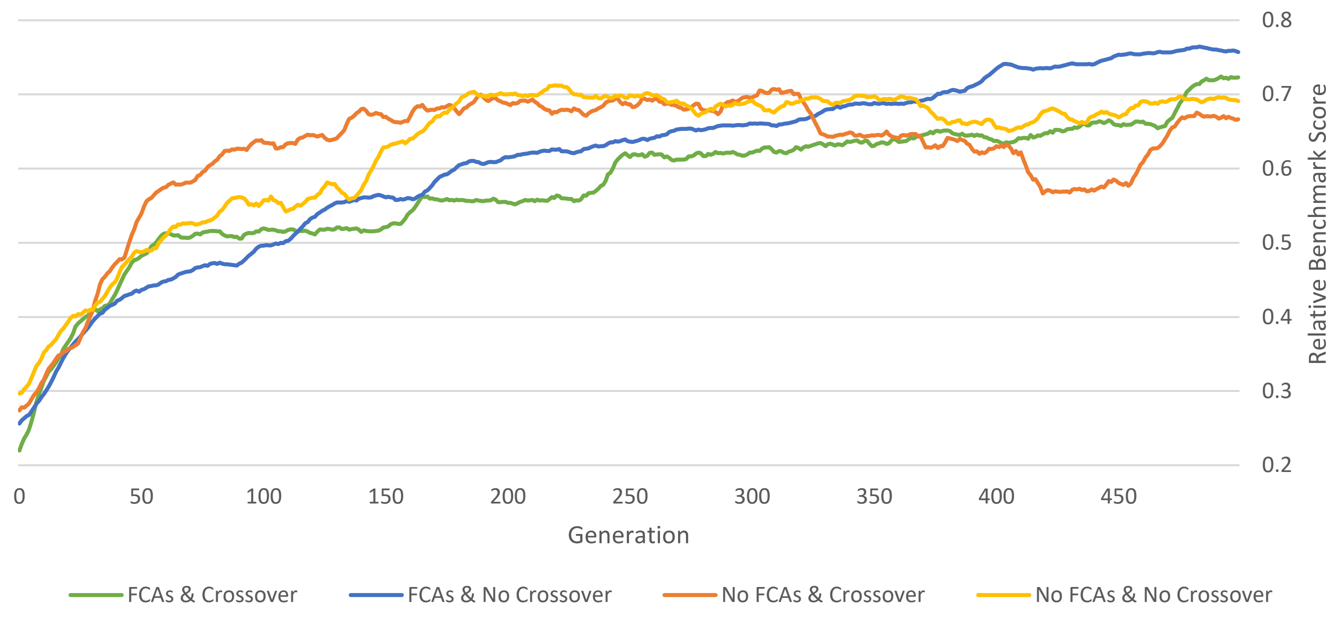

Figure 9. Relative benchmark scores from Robocode without experiential simulated evaluation.

Crossover proved to be ineffective for solution traversal. Both set of trials without crossover maintained higher scores for longer than their counterparts. This is possibly explained by the fact that the solution space is smoother when examined via steps at the level of individual mutations rather than jumps of genetic combination. Because the behaviors were graph-based, the combination of two behaviors could lead to radically different offspring. Previous research by Spears and Anand indicates that the benefit of the crossover is also reduce when used on smaller populations [36]. Based on these results, all tests were compared to the baseline with FCAs and no crossover. Furthermore, all subsequent tests were run with FCAs enabled and crossover disabled.

Analysis

A. Experience Capture

Comparisons of experiential simulated evaluation and highlight extraction tests against the baseline are presented in Figure 11 and Table 5. Comparing full experience usage with highlight extraction reveals little difference in early growth-rates or terminal scores. In the case where experiences from all previous generations were used, experiential simulated evaluation using highlights progressed more gradually and changes in progress rates were smoother.

Table 5. Measurements of Robocode Experiential Evaluation Techniques

	Single Generation		All Generation	
Full Exp	Terminal Score:	0.6207	Terminal Score:	0.4898
	Initial Rate of Progress:	0.0025	Initial Rate of Progress:	0.0031
	Progress Instability:	0.0570	Progress Instability:	0.0845
Highlights	Terminal Score:	0.6187	Terminal Score:	0.4819
	Initial Rate of Progress:	0.0032*	Initial Rate of Progress:	0.0031
	Progress Instability:	0.0529	Progress Instability:	0.0419
Baseline	Terminal Score:	0.7102*		
	Initial Rate of Progress:	0.0019		
	Progress Instability:	0.0075*		

* best result for the given measurement

Conversely, the full experience populations accelerated more quickly and suddenly leveled off into a terminal score. Despite the more gradual progress of the highlight method, progress plateaued at the same level as their counterparts. When only the single most recent generation’s experiences were used, the results appear nearly identical.

B. Retention Method

The measures for the sequence matching baseline are compared with the experiential simulated review in Table 6. The sequence matching baseline and experiential simulated review results are presented in Figure 10. The sequence matching baseline was run without FCAs or crossover.

Table 6. Measurements for Sequence Matching vs. Baseline

Single Generation	Terminal Score:	0.8416
	Initial Rate of Progress:	0.0010
	Progress Instability:	0.0002
Baseline	Terminal Score:	0.8519*
	Initial Rate of Progress:	0.0014*
	Progress Instability:	0.0*

* best result for the given measurement

Single generation retention was used to compare the experiential simulated evaluation method in the sequence matching environment. Experiential simulated evaluation was unable to outperform the baseline in this environment. Observation of the population during execution revealed that the population of sequence matching behaviors grew very homogenous quite early in the evolutionary process. The average score results confirm this observation. The evaluation review results progress rapidly at the beginning matching the baseline, however, near the 25th generation, they begin to diverge. This phenomenon is likely caused by over-learning the reasonably good solutions that the population had already found. If a relatively positive experience were stored in the buffer, graphs which performed closely to it would be pushed to the top of the evaluation pool simply because they acted in a familiar way. Those familiar behaviors became increasingly common and the diversity decreased. To escape this local maximum, the increased genetic diversity might have been introduced through novel randomly generated graphs in each generation or decreasing the influence of the evaluation score on selection of the next generation. As evolution progressed, the rate of progress approximately approached that of the baseline.

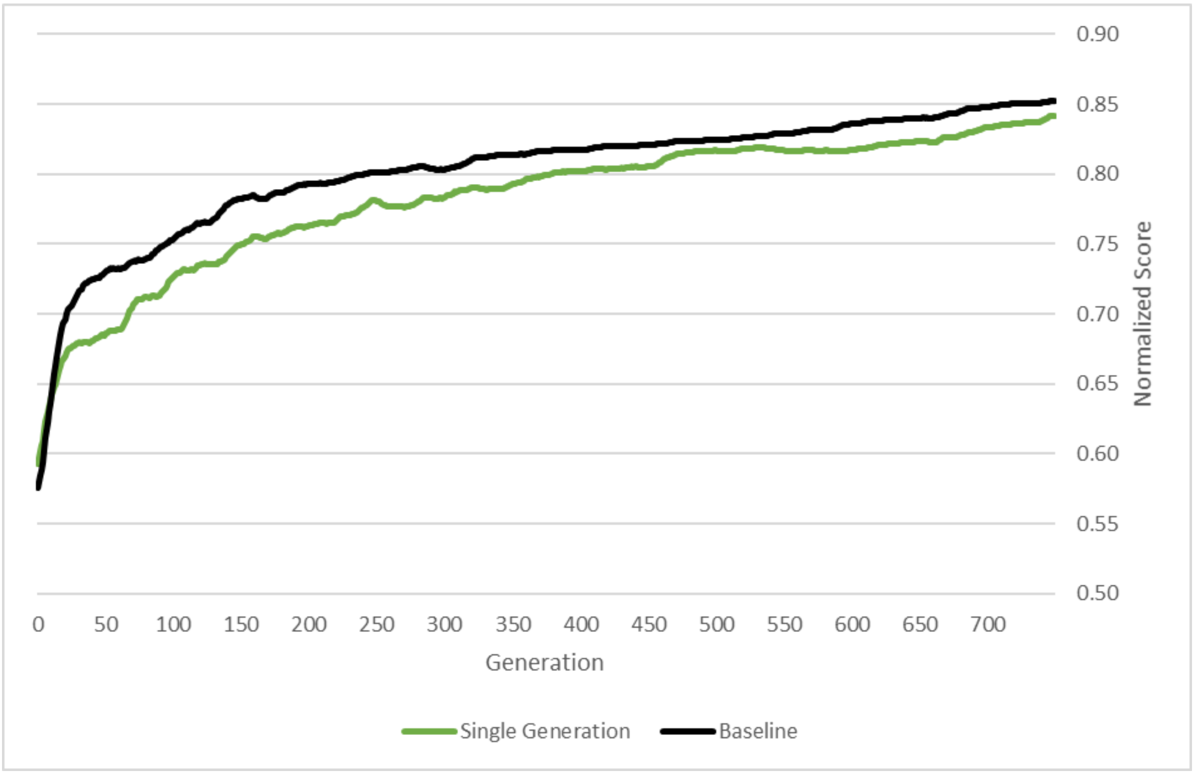

Figure 10. Sequence matching scores with recency retention compared to the baseline.

In the Robocode tests, none of the experiential simulated evaluation methods outperformed the baseline in terminal score, initial rate of progress, or progress instability. However, both runs with single-generation retention maintained higher benchmark scores than the baseline for the first 250 generations. It appears then, that the experiential simulated evaluation offered some initial benefit to the evolution process.

This benefit was overtaken by the baseline method as the experiential simulated evaluation techniques began to plateau.

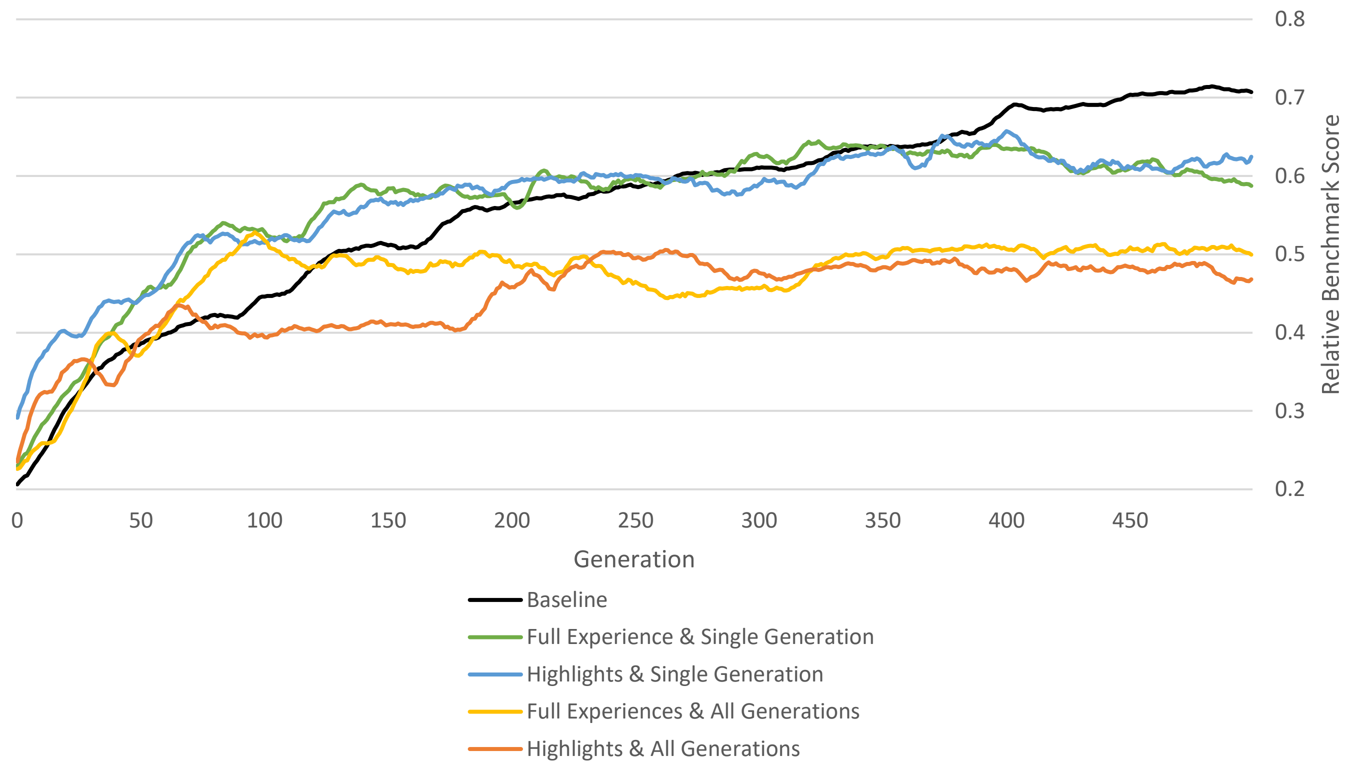

Figure 11. Robocode relative benchmark scores comparing experience creation and retention methods.

The results indicate that the retention method was the far more significant parameter. Progress with all-generation retention is capped at approximately 50% effectiveness. This retention technique was only able to produce robots which inflicted equal amounts of damage as their competitor. Single-generation retention, however, reached a 60% benchmark rating on average. This result implies that some significant experiences from the early generations of the population may have stunted progress later. It is possible that after the population equaled the opponent ability, experiences of wins and loses became more nuanced and, therefore, less significant. These later experiences, then would always be trimmed before the earlier, less relevant, but more significant experiences. Single-generation retention seems to have avoided this obstacle to a degree.

The preferred retention method was obviously single-generation retention. The choice of highlight extraction usage was much more difficult. The differences between the two sets of results are not statistically significant. However, because of the slightly higher final score, the single-generation retention method with full experiences was selected as the best technique.

C. Exploitation vs. Exploration

Comparisons of exploitation/exploration balancing tests are presented in Figure 12 and Table 7.

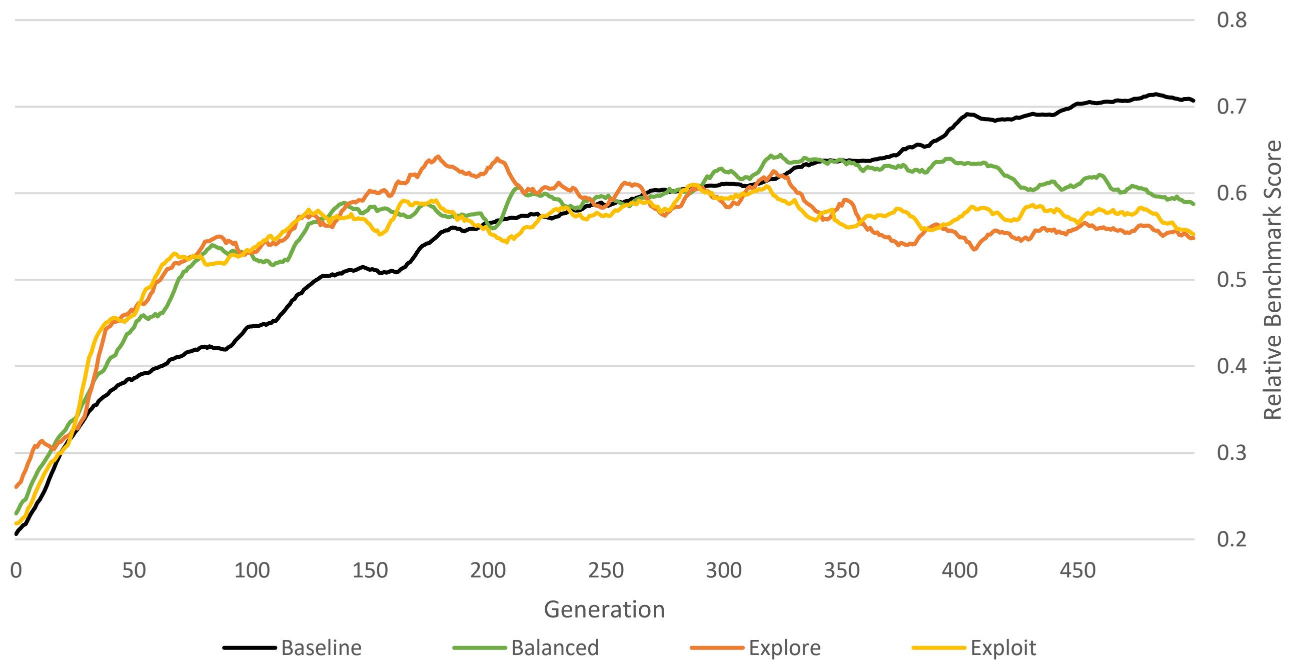

Figure 12. Comparison of Robocode relative benchmark scores for balanced, exploitation-focused, and exploration-focused experiential simulated evaluation.

Although experiential simulated evaluation reached its peak in fewer generations than the baseline, the baseline method surpassed our method between around the 250th generation and continued improving for the remainder of the experiment finishing nearly 33% better than the experiential simulated evaluation score.

Differences between the three runs are slight, however the balanced method achieved the best results. Focusing on either exploitation or exploration appear to have a degrading effect on the population progress such that the success that is achieved is more likely to be lost over that of the balanced method.

The balanced method more successfully retained earned progress. Additionally, the balanced method reached its maximum score just as fast or faster than the other variants and achieved a higher final score. The balanced method was determined to be the best technique for the setting used. From this line of experiments, it is not known if the balanced approach is best for all retention and experience extraction methods.

Table 7. Measurements of Robocode exploitation vs. exploration test

Balanced	Terminal Score:	0.6207
	Initial Rate of Progress:	0.0025
	Progress Instability:	0.0570
Exploit	Terminal Score:	0.5783
	Initial Rate of Progress:	0.0029*
	Progress Instability:	0.0574
Explore	Terminal Score:	0.5811
	Initial Rate of Progress:	0.0021
	Progress Instability:	0.1075
Baseline	Terminal Score:	0.7102*
	Initial Rate of Progress:	0.0019
	Progress Instability:	0.0075*

* best result for the given measurement

Genotypic and Phenotypic Correlation

The relationship between the genotypes and their expressions was examined. In sequence matching, as expected because of limited types of available agents, most graphs converged to a similar structure. Each of the successful graphs contained a single HighestPriority arbiter and several Single agents. The single agents were of various types. In the most successful graphs, the Single agents matched the appropriate character of the alphabet to its location. However, many of these did not include a Single agent for each character. Instead they approximated the alphabet by providing an ‘a’ at index 0 and a ‘d’ at index 2. The gaps were filled by a self-referential subscription at the arbiter. With this mechanism, the graph would repeat the last character until another Single agent had another recommendation. As an example, one graph produced the string ‘aaccfffiiillmoooqqstwwwyh’ and scored 22.53 out of a maximum of 26. Some of the early graphs in this environment survived by filling their entire string with a single letter from somewhere near the middle of the alphabet to gain the maximum score for only one Single agent.

In Robocode, to observe the relationship between the genotypic representation of a coordination graph and the phenotypic expression of that graph, three graph robots are analyzed. The three robots, named Alpha, Bravo, and Charlie, were developed through the genetic program with the baseline configuration. After the 500th generation, the genotype of the best robot was saved for evaluation. Illustrations of the coordination graphs are presented in Figure 13, Figure 14, and Figure 15.

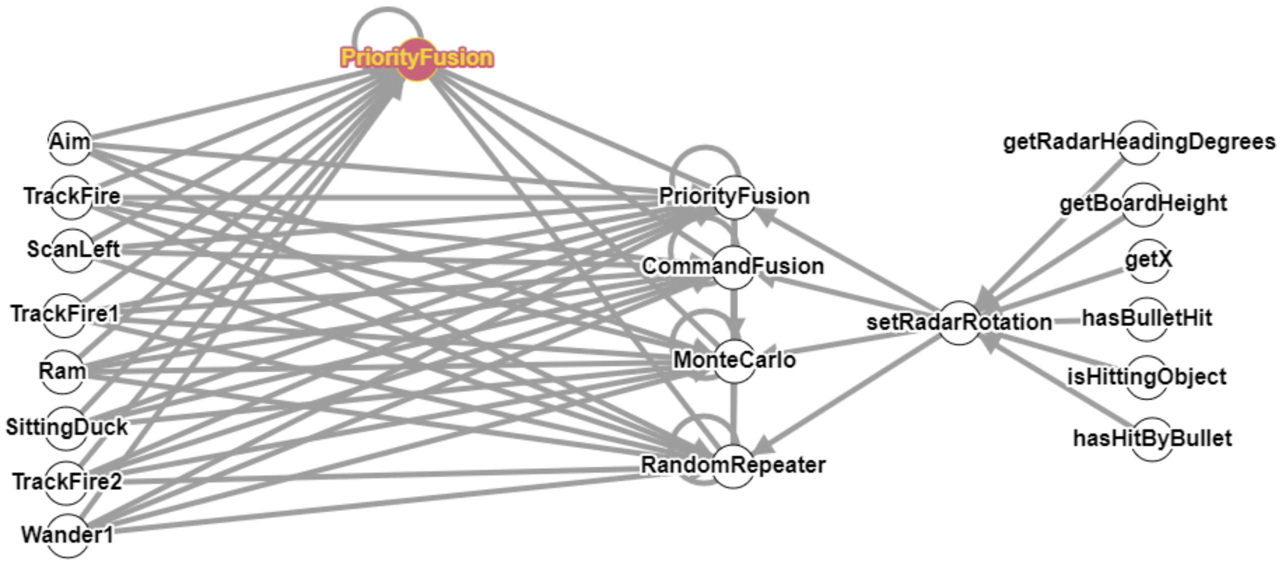

Figure 13. Genotype of Robot Alpha.

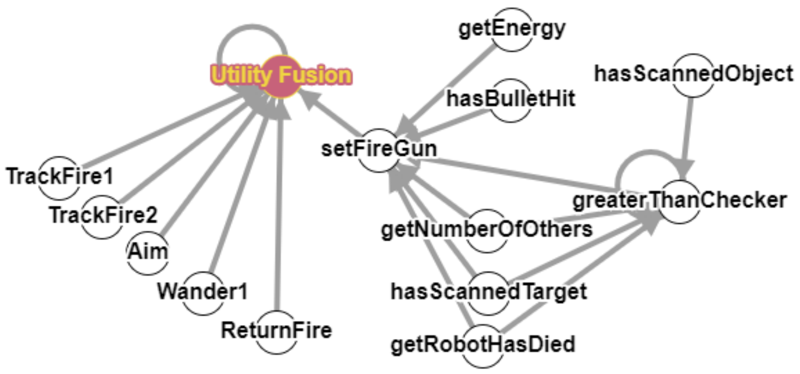

Figure 14. Genotype of Robot Beta.

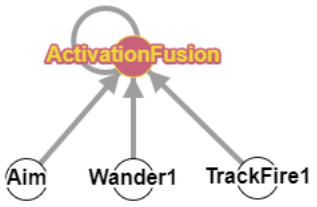

Figure 15. Genotype of Robot Charlie

The end point agent of each graph has been highlighted in red. For clarity of illustration, the edge weights have been removed. In each case, the graphs are visually distinct. Alpha's is characterized by a highly connected structure with multiple arbiters receiving input from many behavior agents and an FCA component setting the radar rotation. Beta contains only one arbiter but has a contingent of behavior agents and an FCA component which either activates or deactivates the firing of the gun. Charlie is by far the simplest of the three graphs containing only a single arbiter and three behavior nodes.

Though the differences of structure are obvious, all three graphs contain the Aim, Wander1, and TrackFire1 behaviors. It appears these components are beneficial to survival against Osprey. However, the other aspects of their coordination graph appear to also play a role. The weights from the other behaviors which are not shared are significant. An issue experienced by many robots observed in the Robocode battles was a failure to productively maneuver the radar for scanning. Occasionally, the robot would move the radar away from the opponent and it would take several seconds to reacquire the target. Alpha appears to have developed an FCA mechanism which motivates a more frequent rotation of the radar dependent on both static and situational sensors. Similarly, many robots failed to fire their gun fast enough in the face of uncertainty. Bravo appears to have developed a trigger-happy finger whereby, based on FCA sensors, it is more or less likely to fire the gun in the absence of a better option.

The differences of structure, however, do not lead to a significant difference in action. All three of these robots execute a single basic strategy: go in circles while shooting Osprey. Osprey appears to be especially vulnerable to this strategy since, if the turn radius of the circle is the right size, Osprey will expend all its bullets on missed shots. The genetically developed robots in all observed successful runs developed some sort of dodging strategy. Not all ran in circles. Some randomly moved forward or backward along a straight line, but this was not as effective. The similarity of phenotype with the dissimilarity of genotype demonstrates the versatility of graph-based genetic representations.

Discussion

Could internal experiential simulation be used to guide a population to a target level of fitness in fewer rounds of fitness assessment? In much of artificial intelligence training and development, the costliest portion of training is gathering data from actual execution. The cost of execution may be measured in time, money, safety, or others. Identifying methods to get more value out of the execution data can help AI projects train more efficiently. The environments and methods used in this research failed to guide a population to fitness in fewer rounds of execution. In all cases, the experiential simulated evaluation inhibited overall growth. This phenomenon is likely caused by overlearning of solutions discovered early in learning. As relative benefits between the status quo and alternative paths diminish, overlearning prevents the population from diversifying to other greater maxima in the solution-space.

Secondarily, this research sought to compare methods of experience retention for simulation to determine benefits and consequences of each. This research indicates a clear benefit from single-generation retention over all-generation retention. Single-generation retention mitigates the overlearning problem. At each step of evolution, the experiential evaluation on had one generation of experiences available to it. In all-generation retention, a particularly significant experience could directly impact hundreds of generations.

Third, comparisons are made between the genotypic representations of behaviors and their associated phenotypic expression, which demonstrates consistent functionalism regardless of the structure. A diversity of intrinsic structures manifested in similar extrinsic behaviors that were well suited to the environment. Given a flexible variety of sub-agent components, the coordination graphs developed the same strategies through different internal structures. This feature of the graph-based coordination framework demonstrates its ability to be flexible genotype for modeling evolutionary development.

Summary

The experiment executed in this study compared the effectiveness of using fine control agents (FCA), a crossover operation, experiential simulated evaluation with single-generation retention, all-generation retention, full memories or experience highlight extraction, and balanced, exploitative, or explorative methods. The results indicate that the baseline with FCAs and without the crossover operation developed the most effective solutions in the time observed. Among experiential simulated evaluation methods, an exploit-explore balanced technique with full experience extraction and single-generation retention performs marginally better than other method. Experiential simulated evaluation outperforms the baseline in the earliest generations but later falls far behind. Experiential simulated evaluation appears to offer an initial benefit, with diminished returns as the generations continue.

V. Conclusion

The work of self-organizing multi-agent systems is highly applicable in areas of AI research. Many problems involve finding the optimal coordination scheme between multiple simpler agents to perform more complicated tasks. If such a coordination graph could be designed automatically, much human design time would be saved. The graph-based coordination framework agents can be applied to other projects with or without the associated genetic program. Similarly, the genetic program can be used to evolve effective coordination graphs without the use of experiential simulated evaluation.

This paper presents an approach for experiential simulated evaluation applied to genetic programs a string matching and the Robocode environments. The behaviors were controlled by graph-based coordination graphs of agents. The coordination graphs were constructed and manipulated through a genetic program to find fitter configurations.

Experimentation tested over several parameters including the buffer retention method, experience highlight extraction, and leaning toward exploitation or exploration. Results showed that, experiential evaluation affects the rate of progress differently depending on the environment. In sequence matching, experiential evaluation succumbs to overlearning and fails to outperform the control. In the Robocode environment, the experiential evaluation method initially performed better than the control but suffers from stunted growth and instability later in the evolutionary process.

Future Work

Future work should apply experiential simulated evaluation to other environments to determine the causal factors which lead to initial success such as in Robocode or consistent stunting as in sequence matching. Features of the environment such as the stochasticity, reward delay time, competitive performance, duration, or score calculation may affect the algorithm. Learning which environments work better with experiential simulated evaluation will allow for better applications and improvements to the algorithm. One such alternative environment might modify Robocode to compete all the robots against one another in the fitness assessment instead of using a 1v1 battle against a static opponent. This environment will likely result in more dynamic strategies and co-evolutionary phenomenon.

Researchers continuing this work may consider a hybridized approach that includes experiential evaluation for the first portion of evolution and then control reverts to the default genetic program. A mixed technique may be able to take advantage of the initially accelerated progress seen with experiential simulated evaluation as well as the stability of the baseline method. A hybrid could either use experiential review for the initial portion of the evolution and switch to another evolutionary method after a benchmark is met or it could blend the two methods and switch between them gradually. Similarly, the exploitation/exploration measure could be adjusted throughout the populational development. Adjustments to the ratio could be made dynamically to either accelerate progress or escape from a local maximum.

Bibliography

[1] I. Georgiadis, J. Magee and J. Kramer, "Self-Organising Software Architectures for Distributed Systems," *Proceedings of the first ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02),* pp. 33-38, 2002.

[2] M.-Y. Liu and O. Tuzel, "Coupled generative adversarial networks," *Advances in neural information processing systems,* pp. 469-477, 2016.

[3] B. G. Woolley and G. L. Peterson, "Genetic evolution of hierarchical behavior structures," p. 1731, 2007.

[4] L. Feng, Y. S. Ong, I. W. H. Tsang and A. H. Tan, "An evolutionary search paradigm that learns with past experiences," *2012 IEEE Congress on Evolutionary Computation, CEC 2012,* p. 10–15, 2012.

[5] L. Zhang, Z. Zhang, Z. Pan, Y. Chen, J. Zhu, Z. Wang, M. Wang and C. Fan, "A framework of dual replay buffer : balancing forgetting and generalization in reinforcement learning ∗," *Surl Ijcai 2019,* 2019.

[6] T. De Bruin, J. Kober, K. Tuyls and R. Babuška, "Experience selection in deep reinforcement learning for control," *Journal of Machine Learning Research,* vol. 19, p. 1–56, 2018.

[7] D. Isele and A. Cosgun, "Selective experience replay for lifelong learning," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018,* p. 3302–3309, 2018.

[8] B. Lake, R. Salakhutdinov and J. Tenenbaum, "One-shot learning by inverting a compositional causal process," *Advances in neural information processing systems,* pp. 2526-2534, 2013.

[9] M. A. Bedau, "Weak Emergence," *Noûs,* vol. 31, p. 375–399, 6 1997.

[10] T. O'Connor, *Emergent Properties,* vol. 31, University of Illinois PressNorth American Philosophical Publications, p. 91–104.

[11] S. Brueckner, "Return from the Ant," *Humboldt-Universität Berlin,* 2000.

[12] R. Pfeifer, M. Lungarella and F. Iida, *Self-organization, embodiment, and biologically inspired robotics,* vol. 318, 2007, p. 1088–1093.

[13] P. F. M. J. Verschure, B. J. A. Kr, /. Sse and R. Pfeifer, "Distributed adaptive control: The self-organization of structured behavior," 1992.

[14] Y. Yamashita and J. Tani, "Emergence of Functional Hierarchy in a Multiple Timescale Neural Network Model: A Humanoid Robot Experiment," *PLoS Computational Biology,* vol. 4, p. e1000220, 11 2008.

[15] J. R. Kok and N. Vlassis, "Using the max-plus algorithm for multiagent decision making in coordination graphs," in *Belgian/Netherlands Artificial Intelligence Conference*, 2005.

[16] E. B. Baum, "Toward a model of intelligence as an economy of agents," *Machine Learning,* vol. 35, p. 155–185, 1999.

[17] P. Gu, S. Balasubramanian and D. H. Norrie, "Bidding-based process planning and scheduling in a multi-agent system," *Computers & Industrial Engineering,* vol. 32, p. 477–496, 4 1997.

[18] T. Logenthiran, D. Srinivasan, A. M. Khambadkone and H. N. Aung, "Multi-Agent System (MAS) for short-term generation scheduling of a microgrid," in *2010 IEEE International Conference on Sustainable Energy Technologies (ICSET)*, 2010.

[19] G. Sartoretti, "Leader-based versus soft control of multi-agent swarms," *Artificial Life and Robotics,* vol. 21, p. 302–307, 2016.

[20] V. Kononen, Asymmetric multiagent reinforcement learning, vol. 1, IOS Press, 2004, p. 105–121.

[21] C. Guestrin, D. Koller and R. Parr, "Multiagent Planning with Factored MDPs," in *Advances in Neural Information Processing Systems (NIPS)*, 2001.

[22] D. Yagan and C. K. Tham, "Coordinated reinforcement learning for decentralized optimal control," in *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007*, 2007.

[23] R. A. Brooks, "A Robust Layered Control System For A Mobile Robot," *IEEE Journal on Robotics and Automation,* vol. 2, p. 14–23, 1986.

[24] J. K. Rosenblatt and D. W. Payton, "Fine-grained alternative to the subsumption architecture for mobile robot control," *IJCNN Int Jt Conf Neural Network,* p. 317–323, 1989.

[25] E. Gat, "Three-layer architectures," *Artificial intelligence and mobile robots,* p. 195–210, 1998.

[26] D. Weyns, Architecture-based design of multi-agent systems, 2010, p. 1–224.

[27] R. B. Rusu, "ROS - Robot Operating System," *Rss,* vol. 1, p. 1–15, 2010.

[28] B. G. Woolley and G. L. Peterson, "Unified behavior framework for reactive robot control," *Journal of Intelligent and Robotic Systems: Theory and Applications,* vol. 55, p. 155–176, 2009.

[29] J. R. Koza and R. Poli, "Chapter 5 GENETIC PROGRAMMING," *Computer,* p. 1–40, 1983.

[30] P. J. Angeline, "Genetic programming and emergent intelligence," *Advances in genetic programming,* vol. 1, p. 75–98, 1994.

[31] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D: Nonlinear Phenomena,* vol. 42, p. 228–234, 1990.

[32] A. Trenaman, "Concurrent Genetic Programming, Tartarus and Dancing Agents," in *Proceedings of the Second European Workshop on Genetic Programming*, London, 1999.

[33] S. Mabu, K. Hirasawa and J. Hu, "A graph-based evolutionary algorithm: Genetic Network Programming (GNP) and its extension using reinforcement learning," in *Evolutionary Computation*, 2007.

[34] R. Poli, "Evolution of Graph-like Programs with Parallel Distributed Genetic Programming," *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA'1997),* p. 346–353, 1997.

[35] F. Pereira, P. Machado, E. Costa and A. Cardoso, "Graph based crossover-a cse study with the busy beaver problem," *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation,* vol. 2, pp. 1149-1155, 1999.

[36] W. Spears and V. Anand, "A study of crossover operators in genetic programming," *International Symposium on Methodologies for Intelligent Systems,* pp. 409-418, 1991.

[37] J. Miller, "Cartesian Genetic Programming," in *Cartesian Genetic Programming*, Berlin, Heidelberg, Springer, 2011, pp. 17-34.

[38] S. Harding and J. Miller, "Evolution of robot controller using cartesian genetic programming," *European Conference on Genetic Programming,* pp. 62-73, 2005.

[39] A. Newell, "SOAR as a unified theory of cognition: Issues and explanations," *Behavioral and Brain Sciences,* vol. 15, p. 464–492, 9 1992.

[40] D. Hassabis, D. Kumaran, C. Summerfield and M. Botvinick, "Neuroscience-Inspired Artificial Intelligence," *Neuron,* vol. 95, p. 245–258, 7 2017.

[41] S. Racanière, T. Weber, D. P. Reichert, L. Buesing, A. Guez, D. Rezende, A. Puigdomènech Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. S. Daan and W. Deepmind, "Imagination-Augmented Agents for Deep Reinforcement Learning".

[42] H. J. Briegel and G. De las Cuevas, "Projective simulation for artificial intelligence," *Scientific Reports,* vol. 2, p. 400, 12 2012.

[43] T. Ziemke, D.-A. Jirenhed and G. Hesslow, "Internal simulation of perception: a minimal neuro-robotic model $," *Neurocomputing,* vol. 68, p. 85–104, 2005.

[44] S. John Turner, W. Cai, M. Yoke Hean Low and H. Aydt, An Agent-Based Generic Framework for Symbiotic Simulation Systems, 2009, p. 357–387.

[45] P. Nordin, W. Banzhaf and M. Brameier, "Evolution of a world model for a miniature robot using genetic programming," *Robotics and Autonomous Systems,* vol. 25, p. 105–116, 1998.

[46] E. J. Wamsley and R. Stickgold, "Dreaming and offline memory processing," *Current Biology,* vol. 20, p. R1010–R1013, 2010.

[47] J. E. Malinowski and C. L. Horton, "Memory sources of dreams: The incorporation of autobiographical rather than episodic experiences," *Journal of Sleep Research,* vol. 23, p. 441–447, 2014.

[48] L. Kuyer, S. Whiteson, B. Bakker and N. Vlassis, "Multiagent reinforcement learning for urban traffic control using coordination graphs," *Joint European Conference on Machine Learning and Knowlege Discovery in Databases,* pp. 656-671, 2008.

[49] C. Guestrin, M. Lagoudakis and R. Parr, "Coordinated reinforcement learning," *ICML,* vol. 2, pp. 227-234, 2002.

[50] R. M. French, "Catastrophic Forgetting in Connectionist Networks," *Encyclopedia of Cognitive Science,* 2006.

[51] T. Schaul, J. Quan, I. Antonoglou and D. Silver, "Prioritized experience replay," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings,* p. 1–21,

2016.

[52] C. Kaplanis, C. Clopath and M. Shanahan, "Continual Reinforcement Learning with Multi-Timescale Replay," 2020.

[53] F. Larsen, *ReadMe for Robocode*, 2013.

[54] T. Kohonen, K. Lagus, J. Salojärvi, J. Honkela, V. Paatero and A. Saarela, "Self organization of a massive text document collection," *Kohonen Maps,* p. 171–182, 1 1999.

[55] X. Mao, Q. Li, H. Xie, R. Lau and Z. Wang, "Least Squard Generative Adversarial Networks," *Proceedings of the IEEE International Conference on Computer Vision,* pp. 2794-2802, 2017.

[56] O. Vinyals, C. Blundell, T. Lillicrap and D. Wierstra, "Matching networks for one shot learning," *Advances in neural information processing systems,* pp. 3630-3638, 2016.

[57] P. Ryser-Welch, *Evolving comprehensible and scalable solvers using CGP for solving some real-world inspired problems,* University of York, 2017.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-06-2020	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* June 2018 – June 2020

TITLE AND SUBTITLE SIMULATED EXPERIENCE EVALUATION IN DEVELOPING MULTI-AGENT COORDINATION GRAPHS	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Watson, Andrew G., Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENY) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-20-J-015
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) 1st Lt Aashae Eberle 711 HPW/RH 5135 Pearson Rd Wright-Patterson AFB, OH 45433 (937) 255-6027 aashae.eberle@us.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) JON: 20G539

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Cognitive science has proposed that a way people learn is through self-critiquing by generating 'what-if' strategies for events (simulation). It is theorized that people use this method to learn something new as well as to learn more quickly. This research adds this concept to a graph-based genetic program. Memories are recorded during fitness assessment and retained in a global memory bank based on the magnitude of change in the agent's energy and age of the memory. Between generations, candidate agents perform in simulations of the stored memories. Candidates that perform similarly to good memories and differently from bad memories are more likely to be included in the next generation. The simulation-informed genetic program is evaluated in two domains: sequence matching and Robocode. Results indicate the algorithm does not perform equally in all environments. In sequence matching, experiential evaluation fails to perform better than the control. However, in Robocode, the experiential evaluation method initially outperforms the control then stagnates and often regresses. This is likely an indication that the algorithm is over-learning a single solution rather than adapting to the environment and that learning through simulation includes a satisficing component.

15. SUBJECT TERMS
genetic programming; mutli-agent system; experiential simulated evaluation; behavior-based control

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Gilbert Peterson, AFIT/ENG
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*
U	U	U	U	75	(937) 255-6565, ext 4281 (gilbert.peterson@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18