Swarm Behaviors: Simulation and Generation

Michael A. Kovacina

August 10, 2005

Contents

List of Figures

List of Tables

Abstract

In this work, a general purpose swarm experimentation platform, SWEEP, has been constructed and interfaced with an evolutionary computing system to demonstrate the feasibility of autogenerating swarm algorithms given a set of high-level objectives. The functionality of SWEEP has been validated through the implementation of several scenarios including object manipulation and UAV chemical cloud detection. The utility of fusing SWEEP with an evolutionary computing system has been demonstrated through the successful evolution of swarm algorithms for dispersion, object collection, and object destruction. The evolved algorithms exhibited performance that approximated the runtime performance of hand-coded solutions.

Chapter 1

Introduction

This thesis implements and demonstrates a set of tools to automatically generate swarm algorithms. SWEEP (**SW**arm **E**xperimentation and **E**valuation **P**latform), a swarm algorithm development and simulation platform is implemented. To demonstrate the utility of SWEEP, several swarm-based algorithms are constructed for free-moving and dynamics-constrained agents, culminating in the development of basic searching and mapping algorithms for UAVs (unmanned air vehicles) tracking chemical clouds. ECS, an evolutionary computing system, which uses finite state machine to evolve swarm algorithms and emergent behavior. SWEEP is used to evaluate the fitness of a solution and a lexicographic ranking system is utilized. Algorithms for agent dispersion and object manipulation are evolved.

1.1 Motivation

Flocking birds, foraging ants, aggregating slime molds: these are just a few of the striking examples of self-organizing behaviors that are observed in nature. Through a solely decentralized and reactive system, where "the whole is greater than the sum of the parts," swarms exhibit behaviors that coordinate on a global scale. Deriving inspiration from nature, swarm theory is a powerful methodology that leverages the self-organization that emerges from the interactions of multiple agents to evoke a group-level behavior that is beyond the capability of any single member of the swarm. Swarm algorithms are most useful for problems that are amenable to an agent-based decomposition, have a dynamic nature, and do not require time-limited or optimal solutions.

Perhaps the most advantageous aspect of swarm intelligence is the large body of work from which to draw inspiration. Originally, swarm theory looked to societal insects such as ants for inspiration. For example, the method in which an ant colony will determine the closest of multiple food sources is also applicable to

addressing graph-based problems such as the traveling salesman problem and the minimum spanning tree problem [9]. But as more attention is focused on swarm intelligence, other sources of inspiration are identified. Recent research is focused on extracting the implicit swarm algorithms that arise from the interactions of more intelligent organisms, i.e. humans [23].

As the size of the swarm and the complexity of the tasks increase, the opportunities for emergence increase dramatically. In order to fully exploit the power of swarm algorithms, human programmers will need tools that extend beyond those currently available for traditional programming. These tools will enable the programmer to work and think on a higher-level than the single agent, specifying goals and constraints for the entire swarm as opposed to loops and conditionals. This is analogous to the development of high-level languages like C and FORTRAN that allowed programmers to focus more on concepts rather implementation. Additionally, tools will be required for the simulation of the developed algorithm for the purposes of evaluation and testing. Finally, tools to identify and classify emergence in a swarm will be required to optimize and verify the performance of a swarm algorithm.

1.2 Contributions

This thesis demonstrates the use of evolutionary computing for the successful autogeneration of swarm algorithms. In order to construct swarm algorithms, a development environment is required to facilitate experimentation and evaluation, hence the construction of SWEEP. SWEEP is designed to be open and extandable. The core SWEEP architecture is constructed with minimal assumptions regarding agent modeling, controller design, or environmental parameters. The utility of SWEEP is demonstrated through the simulation of several swarm-based scenarios including agent dispersion, task assignment, chemical cloud detection, and object manipulation.

There is a small body of work related to the evolution of swarm algorithms, mostly related to the development of behaviors for swarm robotics systems. Zaera *et al.* [29] used an evolutionary method to develop behaviors for aggregation, dispersal, and schooling in fish. Though successful in developing aggregation and dispersal algorithms, disappointing results were realized with regards to the schooling behavior, mainly due to the difficulty involved in crafting a fitness evaluation function that adequately captured the salient features of a schooling behavior. Gaudiano, Bonabeau, and Shargel [11] used a traditional genetic algorithm to tune probabilities for handmade probabilistic state machines used to coordinate the behaviors of a swarm of unmanned air vehicles. Perhaps the best known example of using evolution to generate swarm behaviors is the work performed by Koza. In Koza's work [18], he used stateless LISP S-expressions to evolve an ant-like food foraging algorithm for a small swarm of simulated ants.

This thesis extends the body of work related to the generation of swarm behaviors through evolutionary

programming. In contrast to Koza's work, this thesis uses a finite state machine solution representation that provides for the evolution of state-based algorithms as opposed to the state-less algorithms generated through LISP S-expressions. The evolutionary computation system developed uses the SWEEP platform to simulate candidate solutions and evaluate their fitness with respect to the multiple goals of the target scenario. Additionally, a unique data fusion method is employed along with lexicographic scoring to overcome the challenges presented by the multi-objective fitness functions required to evolve correct swarm algorithms.

1.3 Summary and Outline

This thesis focuses on the identification and implementation of tools required for the autogeneration of swarm algorithms. The work done in this thesis is organized into seven chapters.

Chapter 1 lays out the focus of this thesis, the motivation for researching the autogeneration of swarm algorithms, and the contributions made by this work.

Chapter 2 introduces the concept of swarm intelligence. In addition, the term swarm intelligence is formally defined for this work.

Chapter 3 presents the SWEEP platform and its design.

Chapter 4 describes several example swarm algorithms developed with SWEEP including dispersion, task assignment, and chemical cloud detection using UAVs.

Chapter 5 presents ECS, the evolutionary computing system used to evolve the swarm algorithms in Chapter 6. Here we also discuss using SWEEP as a fitness function.

Chapter 6 explores the evolution of swarm algorithms for agent dispersion, and object manipulation, a combination of object collection and object destruction. The fitness measures used for each scenario are discussed. In addition, evolved solutions for object manipulation are presented.

Chapter 7 reviews the conclusions and contributions of this work, and suggests avenues for future research efforts.

Chapter 2

Swarm Intelligence

2.1 Swarm Intelligence Overview

Birds flock, fish school, slime molds aggregate. Why is it that evolution has produced swarming in so many different contexts? The formation of a swarm simultaneously provides the individual and the group a number of benefits arising from the synergy of interaction. As an agent in the swarm, the potential benefits available to the individual is maximized through the efforts of the others in the swarm. From the perspective of the swarm as an organism, the survivability of the swarm is increased through the higher-level coordination that emerges from the low-level interactions of the individuals.

There are several examples in nature that exhibit the individual and group benefits of a swarm. First, swarming provides the ability to forage more effectively. Perhaps the best known example is the foraging behavior of ants. Each ant searches for food, and upon finding a food source returns to the colony, emitting a pheromone trail along the way. Other ants detect the pheromone trail and follow the trail to the food source. In turn, they also emit the pheromone on the return trip to the colony, thus reinforcing the trail. As a result, the trail to the closest food source is the strongest, thus without any central supervision the ant colony has located the closest food source, thus optimizing their search.

Also, there is the concept of safety in numbers. A fish traveling in a school is much safer from predators that it would be traveling alone. When attacked, the school can split to evade the predator. Also, many predators will attack a solitary fish, but will be wary of large school of fish that may retaliate *en masse*.

Finally, traveling as a group maximizes the distance able to be traveled. For example, the V-shape formation observed in flocking birds is a functional formation where the flight of the lead bird reduces the drag on the rest of the flock, thus enabling the flock to travel farther as a whole. Table 2.1 provides other examples of swarming, including some of the other benefits of swarming for both the individual and the

collective.

As seen through these examples of swarming in nature, evolution and natural selection have tuned the behaviors of these swarms, enabling seemingly globally directed behaviors to emerge solely through the interactions of the individuals in the collective. Thus, these swarms are self-organizing systems where "the whole is greater than the sum of the parts." In order to further study the application of swarm intelligence as a problem solving technique, a formal definition of what is meant by swarm intelligence must first be established.

Swarm	Behavior	Individual Benefit	Collective Benefit
Birds	Flocking	Reduces drag	Maximize flock travel
Fish	Schooling	Enhances foraging	Defense from predators
Slime Mold	Aggregating	Survive famines	Find food quicker
Ants	Foraging	Enhances foraging	Finding food faster
Termites	Nest building	Shelter	Climate-controlled incubator

Table 2.1: Examples of swarm intelligence that can be found throughout nature.

2.2 Definition of Swarm Intelligence

In order to establish a concrete definition for swarm intelligence, a working definition of *agent* must be established. For this work, an *agent* is defined as an entity that is able to sense and affect its environment directly or indirectly. A *swarm* is then defined as a collection of interacting agents. Finally, an *environment* is a substrate that facilitates the functionalities of an agent through both observable and unobservable properties. Thus, an environment provides a context for the agent and its abilities. For example, an ocean is an environment to a fish in that it enables the fish to swim, but factors such as viscosity affect the ability of the fish to swim.

With these definitions in place, a definition for swarm intelligence can be constructed. In order to construct this definition, one must consider the definitions already existing in the field. As noted by Bonabeau, the term "swarm intelligence" was first used by Beni, Hackwood, and Wang in regards to the self-organizing behaviors of cellular robotic systems [4]. This interpretation of swarm intelligence focuses on unintelligent agents with limited processing capabilities, but possessing behaviors that collectively are intelligent. Bonabeau [9] extends the definition of swarm intelligence to include "any attempt to design algorithms or distributed problem-solving devices inspired by the collective behavior of social insect colonies and other animal societies".

Clough sees swarm intelligence as being related solely to a collection of simple autonomous agents that depend on local sensing and reactive behaviors to emerge global behaviors [8]. Quite similarly to Clough, Payman [1] defines swarm intelligence as the property of a system whereby the collective behaviors of

unsophisticated agents interacting locally with their environment cause coherent functional global patterns to emerge.

The definitions of swarm intelligence outlined here are only a few of the many varied definitions available, but from these definitions several key similarities can be extracted. First, a swarm relies on a large number of agents. Secondly, the power of a swarm is derived from large numbers of direct and/or indirect agent interactions, thus the need for a relatively large number of agents. It is through the interactions of the agents that their individual behaviors are magnified, thus emerging a global level behavior for the entire swarm. Finally, the agents in a swarm tend to be thought of as simple because the complex emergent behaviors of the swarm is beyond their capabilities and comprehension.

Based on the previously discussed definitions of swarm intelligence, we will establish a new definition of swarm intelligence that will be used for the purposes of this work. Thus, given a group of agents considered simple relative to the observer, we define swarm intelligence as

> a group of agents whose collective interactions magnify the effects of individual agent behaviors, resulting in the manifestation of swarm level behaviors beyond the capability of a small subgroup of agents.

In accord with our definition, we identify several key properties required for the emergent behaviors observed in swarms. Foremost are large numbers of agent interactions. These interactions create feedback, both positive and negative, which in turn affects an agent's context. Coupled over time with changing contexts, the continued interactions among agents accumulate, creating a feedback effect larger than that realized by any single agent. This exploitation of magnified feedback pathways is the key to swarm intelligence. Additionally, the interactions between agents not only affect the agents but also the environment. The modification of the environment imbues the agents with an indirect means of communication known as stigmergy. [REF]

In addition, randomness plays a large role in the success of a swarm algorithm. When properly applied, random behavior can significantly reduce the size and complexity of an agent's ruleset by reducing the number of conditions that must be handled. For example, a random walk combined with pheromone dropping serves as a much simpler and robust foraging algorithm as compared to a more complex algorithm that would need to account for the many situations that an agent in the swarm would encounter. Of course, this reduction in choice is coupled with the loss of some precision and efficiency. Also, randomness is an excellent way to escape from local extremum, with respect to the problem at hand, without having to resort to a case-by-case enumerative method. Fortunately, the robust nature of a swarm compensates on a global scale for the possible misbehavior of a single agent, thus the adverse effects of random behavior are relatively small.

2.3 Swarm Intelligence as a Problem Solving Technique

As seen through the mentioned examples of swarm intelligence, a swarm algorithm is a balance of ability and numbers. For example, assume an arbitrary task to be completed. On one end of the spectrum, a single complex agent capable of performing the task must possess the all the intelligence and skill required to complete the task. This enables the agent to quickly and efficiently address the task, but at a cost of robustness and flexibility. If any of the required components of the agent fails, the agent cannot complete the task. Additionally, when the ability of the agent needs to be expanded, the complexity of the agent will be increased, thus possibly causing unforeseen issues and reducing the maintainability of the agent. Finally, if the task is dynamic, a single agent may have a difficult time of keeping up with the changing environment.

At the other end of the spectrum, for the same task, assume that you are given a non-homogeneous swarm of agents who collectively have the ability to accomplish the task. The swarm contains agents with redundant abilities, thus some duplicate work will be performed but with the benefit that a malfunctioning or damaged agent does not preclude the accomplishment of the task. Practically speaking, the complexity of each swarm agent is much less than that of the single centralized agent, thus maintenance and replication is much easier. Finally, though swarms may not always optimally perform small or static tasks, as the size and dynamicism of a problem grow, a swarm can scale with the problem with the addition of new agents.

Swarm algorithms are most useful for problems that are amenable to an agent-based decomposition, have a dynamic nature, and do not require time-limited or optimal solutions. Many problems today are ideal candidates for swarm algorithms, including traveling salesman problems, data fusion on distributed sensor networks, network load balancing, and the coordination of large numbers of machines, ~~i.e.~~ cars on the highway or automatons in a factory.

The concept is an enticing one: program a few simple agents to react to their limited environment, mechanically replicate the agents into a swarm, and gain the ability to realize complex goals. The resulting system is flexible, robust, and inexpensive. Flexibility comes through the ability of a swarm to adapt behaviors to operational contexts. The system is robust because it provides for the possibility that several agents can be lost or go rogue while the swarm still achieves its goals. In addition, the size and composition of the swarm can change over time in conjunction with evolving scenario parameters. Furthermore, this approach is also less expensive because many simple system components consume fewer resources to build and maintain than a single, centralized system. **Table 2.2** summarizes the main benefits of swarm intelligence approach.

Robust	Swarms have the ability to recover gracefully from a wide range of exceptional inputs and situations in a given environment
Distributed and Parallel	Swarms can simultaneously task individual agents or groups of agents to perform different behaviors in parallel
Effort Magnification	The feedback generated through the interactions of the agents with each other and the environment magnify the actions of the individual agents
Simple Agents	Each agent is programmed with a rulebase that is simple relative to the complexity of the problem
Scalability	An increase in problem size can be addressed by adding more agents

Table 2.2: Benefits of using a swarm intelligence as a problem solving technique.

2.4 Applications of Swarm Intelligence

Many real-world problems are amenable to a swarm intelligence approach. Specifically, the areas of computer graphics, scheduling, and networking have leveraged swarm methodologies to create scalable, robust, and elegant solutions to complex, challengening problems. This section presents a summary of some of these real-world applications of swarm intelligence.

Computer Animation

Boids are simulated flocking creatures created by Craig Reynolds to model the coordinated behaviors bird flocks and fish schools [25]. The basic flocking model consists of three simple steering behaviors that describe how an individual boid maneuvers based on the positions and velocities its neighboring boids: separation, alignment, and cohesion. The separation behavior steers a boid to avoid crowding its local flockmates. The alignment behavior steers a boid towards the average heading of its local flockmates. The cohesion behavior steers a boid to move toward the average position of its local flockmates. These three steering behaviors, using only information sensed from other nearby boids, allowed Reynolds to produce realistic flocking behaviors. The first commercial application of the boid model appeared in the movie *Batman Returns*, in which flocks of bats and colonies of penguins were simulated.

While boids modeled the flocking behaviors of simple flying creatures, Massive is able to generate lifelike crowd scenes with thousands of individual "actors" using agent-based aritificial life technology. Agents in Massive have vision, hearing, and touch sensing capabilities, allowing them to respond naturally to their environment. The intelligence for the agents is a combination of fuzzy logic, reactive behaviors, and rule-based programming. Currently, Massive is the premier 3D animation system for generating crowd-related

visual effects for film and television. Most notably, Massive was used to create many of the dramatic battle scenes in the *Lord of the Rings* trilogy.

Scheduling

One of the emergent behaviors that can arises through swarm intelligence is that of task assignment. One of the better know examples of task assignment within an insect society is with honey bees. For example, typically older bees forage for food, but in times of famine, younger bees will also be recruited for foraging to increase the probability of find a viable food source. Using such a biological system as a model, Michael Campos of Northwestern University devised a technique for scheduling paint booths in a truck factory that paint trucks as they roll off the assembly line [5]. Each paint booth is modeled as an artificial bee that specializes in painting one specific color. The booths have the ability to change their color, but the process is costly and time-consuming.

The basic rule used to manage the division of labor states that a specialized paint booth will continue to paint its color unless it percieves an important need to switch colors and incur the related penalties. For example, if an urgent job for white paint occurs and the queues for the white paint booths are significant, a green paint booth will switch to white to handle the current job. This has a two-fold benefit in that first the urgent job is handled in an acceptable amount of time, and second, the addition of a new white paint booth can alleviate the long queues at the other white paint booths.

The paint scheduling system has been used successfully, resulting in a lower number of color switches and a higher level of efficiency. Also, the method is responsive to changes in demand and can easily cope with paint booth breakdowns.

Network Routing

Perhaps the best know, and most widely applied, swarm intelligence example found in nature is that of the foraging capabilites of ants. Given multiple food sources around a nest, through stigmergy and random walking, the ants will locate the nearest food source (shortest path) without any global coordination. The ants use pheromone trails to indicate to other ants the location of a food source; the stronger the pheronome, the closer the source. Through a simple gradient following behavior, the shortest path to a food source emerges.

Di Caro and Dorigo [7] used the biological-inspiration derived from the ant foraging to develop AntNet, an ant colony based algorithm in which sets of artificial ant move over a graph that represents a data network. Each ant constructs a path from a source to a destination, along the way collecting information about the total time and the network load. This information is then shared with other ants in the local area. Di Caro and Dorigo have shown that AntNet was able to outperform static and adaptive vector-distance and link-state shortest path routing algorithms, especially with respect to heavy network loads.

The success of swarm-inspired network routing algorithms such as AntNet have encouraged many in the communications industry to explore ant colony approaches to network optimization. For example, British Telecom has applied concepts from ant foraging to the problem of load balancing and message routing in telecommunications networks. Their network model is populated by agents, which in this case are modeled as artificial ants. The ants deposit pheromone on each node traversed on their trip through the network, thus populating a routing table with current information. Call routing is then decided based on these routing tables.

Chapter 3

Swarm Simulation Software

This chapter describes the framework and implementation of SWEEP, the Swarm Experimentation and Evaluation Platform. The implementation of SWEEP discussed in this chapter is an extension of the original SWEEP implementation created by Palmer and Hantak [12].

SWEEP identifies four major software components for a swarm simulation: *Simulation*, *Agent*, *Environment*, and *Probe*. A *Simulation* object is composed of at least one Agent object, one or more *Environment* objects, and zero or more *Probe* objects. A *Simulation* object is responsible for constructing the various simulation components, establishing the manner in which the simulation components communicate and interact, and managing the runtime aspects of the simulation. Associated with a *Simulation* is one or more *Agents*. An *Agent* represents a logical individual within a simulation, which in implementation can in fact be a collection ("swarm") of other *Agents*. The *Environment* components associated with a *Simulation* define the space where *Agents* exist, providing a context for the sensing and actuation. Finally, *Probes* define methods of extracting internal simulation data from and inserting external information into a running simulation.

3.1 Design Overview

The overall design goal for SWEEP was the creation of a flexible, general-purpose swarm algorithm testing and development platform targeted for both users and developers alike. The high-level overview of the SWEEP design can be seen in **Figure 3.1**. In order to separate the logic of an *Agent* and the simulation of the *Agent*, the basic unit of autonomy in SWEEP is defined as an *Entity*, being composed of an *Agent* ("mind") and an *Avatar* ("body"). An *Agent* is defined as an object having both a *State* and a *Controller*. The *State* is used as input to the *Controller* to determine the next *Action* to be performed by the *Avatar* associated with the *Agent*. The *Avatar* consists of a *Model* that defines the way the *Avatar* interacts

with the *Environment*. After executing the *Actions* from an *Agent*, the *Avatar* updates the *State* of the *Agent* to reflect the effects of the executed *Actions*. Finally, all SWEEP components communicate through *Connectors*, which allow for flexible and probeable inter-component communication.

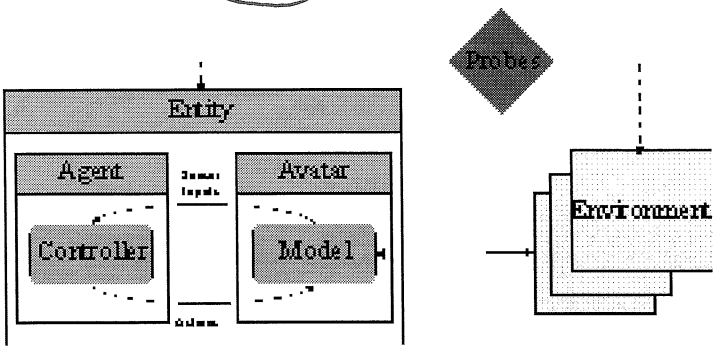


Figure 3.1: High-level overview of the SWEEP architecture showing the relationship between the major components.

3.2 Simulation

A SWEEP simulation is built from an XML simulation specification file. Within the specification file are six subsections, each defining a portion of the simulation. Using a psuedo-XML formatting, **Figure 3.2** lists and defines the six major subsections found in a SWEEP simulation specification.

Each XML subsection is passed to a *Builder* [10], which may be explicitly specified, to parse its own section. The preferred parsing method is to recursively pass well-defined subsections to *Builder* objects that know how to handle that specific data. This design guideline is used throughout the implementation of SWEEP. The *SimulationBuilder* receives the simulation section of the specification file, and then breaks that section into the six main sections. The *SimulationBuilder* then passes each subsection to the appropriate *Builder* associated with that section. This parsing strategy allows each section to define a custom syntax, which in turn provides for more flexible and usable component architectures by separating the data format from the data itself. For example, a *Controller* that uses a proprietary state machine library can use either an XML state machine encoding or a custom state machine encoding without requiring any modifications to the proprietary state machine code.

The default *Simulation* implements a sequential update strategy. First, the *Environment* is updated to ensure that any *Agent* decisions are based on current information. Then the *Entities* are updated by first updating the *Agent* that determines which *Actions* are to be performed by the *Avatar*. Then the *Avatar* is updated, executing commands from the *Agent*, resulting in an updated *State* for the next *Agent* update. Finally, any defined *Probes* are executed. It is important to note that SWEEP is not wedded to a

```
<simulation>
    <main>
<!--    Defines how the specified components should
        be connected together to create the simulation -->
    </main>
    <agent>
        Describes how the controller and the avatar interact,
        defines state variables that will be needed
    </agent>
    <controller >
        Specifies the logic for an agent
    </controller>
    <model>
        Defines the platform to be simulated, used by the avatar
    </model>
    <environment>
        Setup of the context in which the agents interact
    </environment>
    <probes>
        Specification of data extraction and insertion
        method, can be used for interactive GUIs
    </probes>
</simulation>
```

Figure 3.2: The SWEEP simulation specification defines six major subsections. Each subsection defines characteristics for different aspects of the simulation. The file uses an XML format.

sequential update strategy. Any update model, such as a fully threaded or a distributed update model, can be implemented and used with SWEEP.

3.3 Agent

An *Agent* has two main components, a *State* and a *Controller*. The *State* represents the current values of all variables that define the agent, and can be composed of both manifest and latent variables. A manifest variable is something that the *Agent* has the ability to measure, whereas the value of a latent variable cannot be measured. Many of the manifest variables in a *State* object are values linked to *Sensors* in the *Environment*, thus providing a mechanism for cleanly communicating *Sensor* information from the *Environment* to the *Controller*.

The *Controller* defines the logic by which the *Agent* is governed. Based on the current *State* of the *Agent*, the *Controller* determines what *Actions* should be executed by the *Avatar* associated with the *Agent*. The default controller implemented for SWEEP is a finite state machine. Finite state machines were chosen for their balance of power and simplicity. A state machine uses sensor information from an *Avatar* as input, and *Actions* for the *Avatar* as output. In the simulation specification file, the state machine has two input

formats: state-based and transition-based. In the state-based version, each state was defined by and XML section populated with the list of rules associated with that state, as shown in **Figure 3.3**. In the transition-based version, the state machine section was populated by a list of transition rules that indicated the source state associated with the rules (**Figure 3.4**).

```
<state name="s-1">
<transition nextState="s-2" condition="101" action="esc"/>
<transition nextState="s-1" condition="111" action="jmp"/>
...
</state>
```

Figure 3.3: A simple example of a state-based finite state machine in an XML format.

```
<transition state="s-1" nextState="s-2" condition="101" action="esc"/>
<transition state="s-1" nextState="s-1" condition="111" action="jmp"/>
```

Figure 3.4: A simple example of a transition-based finite state machine in an XML format.

3.4 Avatar

The *Avatar* is the part of the *Entity* that extends into the *Environment*. The purpose of the *Avatar* is two-fold. First, by separating the modeling of an *Agent* into an *Avatar*, it is easier to keep free the logic of the swarm algorithm from dependencies on environmental implementation issues. Secondly, the information abstraction ability of an *Avatar* allows an *Agent* to seamlessly interact with either a computer-model of the target platform, or an actual hardware platform whose sensors and actuators are networked into the simulation. This allows earlier hardware testing of swarm algorithms by mixing real and virtual platforms within the same scenario.

Every *Avatar* has an associated *Model* that dictates the characteristics and abilities of the platform being simulated. For example, a UAV model will dictate a minimum turning radius, maximum speed, and wing configuration. *Sensors* are the most important aspect defined by the *Model* because *Sensors* provide decision information to the *Agent*. The *Model* is also responsible for defining the *Actions* chosen by an *Agent*. Finally, the *Model* and the *Environment* are responsible for determining what, if any, direct methods of inter-agent communication are available.

3.5 Environment

The *Environment* defines the context in which the agents exist. The term "environment" is used in a very broad manner in order to create the most flexible definition for the system. There are three core functionalities that the *Environment* is responsible for:

1. Defining fundamental laws that *Agent* objects must respect
2. Presenting an information abstraction layer for the purposes of *Agent* interaction
3. Facilitating direct and indirect *Agent* interactions.

Perhaps the most important functionality of the *Environment* is the role it plays as an information abstraction layer, which enables agents to "sense" information about the environment without having to understand the implementation details of the environment. For example, the concept of a neighborhood is functionally different on a graph environment as opposed to a grid environment. However, to an agent whose ruleset depends on neighbor information, the meaning of the data does not need to change to accommodate different *Environment* implementations. Thus, the information abstraction component of the *Environment* object facilitates the creation of simulations where environments can be interchanged without a change to the logic of the agent.

Additionally, the *Environment* is responsible for facilitating agent interactions. One of the key components of any swarm algorithm is the large number of both direct and indirect interactions between agents. Though some of these interactions may occur externally between different agents, the majority of interactions will take place within the context of the *Environment*, and thus will need to be mediated in order to minimize the need for agents to understand the implementation of the environment. For example, when an agent uses a pheromone to stigmergically communicate with other agents, an agent should not have to be concerned with how the chemical gradient information is stored and processed, just that it exists. Thus, facilitating agent interactions is a more specific way of presenting an agent-oriented information abstraction layer.

Finally, the *Environment* defines a set of rules or laws that must be adhered to by any *Avatar* associated with that particular *Environment*. Depending on the *Environment* implementation, these laws can range from constants (like gravity or the maximum network throughput) to constraints that must be satisfied or obeyed (i.e. $F=ma$, or no more that three agents can be resident on a node at any one time). It is through the definition and manipulation of these laws that environment with varying levels of complexity and fidelity can be constructed and interchanged.

3.6 Probe

The ability to interact with a running simulation through *Probes* is one of the most useful features of SWEEP. *Probes* in SWEEP serve two purposes: to extract information from a simulation, and to insert information into a simulation. Possible uses for *Probes* include: generating animations of the simulation, storing statistical information on a simulation, or producing a user interface to allow a user to examine the effects of changing a parameter value during the execution of the simulation.

Due to the data models of modern programming languages, which restrict access to memory based on a course permission models, completely unrestricted access to data for *Probes* is not entirely possible. A solution to this problem is the introduction of *Connectors* that serve as data conduits between components in SWEEP. Any data passed through a conduit is made available to any *Probe*. In a sense, *Probes* tap the communication lines in the same way one might tap a phone line. *Probes* are also able to delete or inject data into a *Connector*. Thus, using *Connectors*, the full functionality of *Probes* is realized. No standard format was introduced for the definition of *Probes* due to the wide and disparate variety of *Probes* that are possible.

Chapter 4

SWEEP Applications

This chapter presents three progressively more complex swarm demonstrations using the SWEEP platform. The agent dispersion demonstration implements a simple algorithm that physically disperses a swarm to a predefined density. The task assignment demonstration implements a decentralized task assignment algorithm which is able to find "good solutions quickly". Finally, a more comprehensive demonstration is constructed in which a swarm of UAVs is tasked to search for and map a chemical cloud in a bounded region. This demonstration includes UAV agents that respect the flight dynamics of a simple UAV, an environment with simple wind conditions and a chemical cloud, and the use of SWEEP within Monte Carlo style data analysis framework.

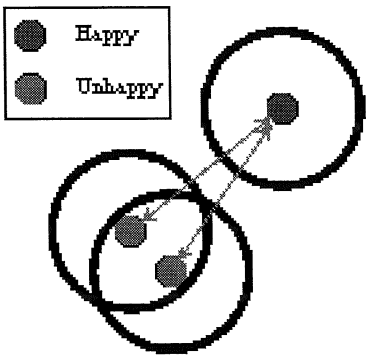
4.1 Dispersion

The goal of a swarm dispersion algorithm is to achieve positional configurations that satisfy some arbitrarily defined criteria. Such criteria include distances from neighboring agents, densities of neighboring agents, or the total area covered by the entire swarm. Practical applications of dispersion algorithms for mobile agents include: conducting a search of the Martian landscape using biologically-inspired tumbleweed robots [15], forming an ad-hoc wireless network using autonomous UAV or blimps [6], or as a preparatory action for a more complex swarm algorithm. The key to the dispersion algorithm is a measure of the quality of an agent's position. The measure used in this demonstration is a binary function, returning true if the position is good and false if the position is bad, where good and bad are defined relative to the number of neighbors for a given agent. Below is a pseudo-code description of the dispersion algorithm

Figure 4.1(b) is an example of the dispersion algorithm in progress. The goal of each agent is to achieve and maintain a constant minimum and maximum distance from all neighboring agents. The red agents are

If Evaluate(pos) == false
    Then
        Move randomly
    Else
        Do not move
End

(a) Simple dispersion algorithm



(b) An example of the dispersion algorithm in action

Figure 4.1: The simple dispersion algorithm used in this demponstration and an example of the dispersion algorithm in action.

not satisfied with their position because they are too close to each other, and thus are mutually violating the minimum distance constraints. The blue agent is satisfied with its position because the red agents are being kept at a distance that satisfies both the minimum and maximum distance requirements.

Table 4.1 is a summary of the simulation parameters used in implementing the dispersion algorithm. Assume a swarm is randomly spread throughout a region, with the goal of dispersing out to a certain density in order to provide a predetermined level of area coverage. The goal of each agent is to achieve a position where no neighboring agent is within 2 grid units, and there is at least one agent within 5 grid units. Each agent is equipped with a distance sensor that can determine the distance of neighboring agents who are within 6 grid units.

Number of Agents	100
Size of Region	20x20, 30x30, 50x50
Actions	Move-RANDOM, Move-NONE
Sensors	Neighbor-Distance
Dispersion Measure	No agents closer than 2 grid units and at least 1 agent within 5 grid units

Table 4.1: Summary of dispersion simulation parameters

The region boundaries provide an upper limit on the total area covered by the swarm, thus region size can indirectly influence execution speed. In the 50x50 case, on average the swarm disperses to a stable pattern within 300 time steps. The 30x30 case takes substantially longer to emerge a stable pattern, usually around 500-600 time steps due to the reduced amount of area with which to disperse. Some simulations in the 30x30 case took over 1000 time steps to settle into a stable pattern due to clusters of agents forming in a corner of the region, thus reducing the number of useful moves available. Finally, the 20x20 case magnifies the clustering problem seen in the 30x30 case. A stable dispersion pattern is not found by the swarm in over 2000 time steps, but non-stabilizing dispersion patterns do begin to emerge.

4.2 Task Assignment

One of the many benefits of a decentralized multi-agent algorithm is the ability to decompose a problem into parallelizable components. In order to realize the full potential of a distributed multi-agent algorithm, a co-operative approach that coordinates tasks in a non-overlapping manner becomes necessary. The Cooperative Assignment of Swarm Tasks (CAST) auction [21] is a fully decentralized task assignment algorithm based on synchronized random number generators that requires only agent-local knowledge about all possible tasks. The CAST auction exhibits several desirable features for a swarm including parallel computations, a limited number of broadcast communications, workable solutions with very short computations, and additional solution refinement with increased computation time. In addition, the CAST auction guarantees that with enough time the optimal solution will be found.

The basic assumptions of the CAST auction algorithm are minimal. First, assume that each agent has a random number generator that can be synchronized across the swarm. This can be done by broadcasting a common seed or having a common list of seeds ready for use. The CAST auction also requires that all agents can perform a broadcast communication that all other members of the swarm can receive. In addition, assume that all agents either have or can compute their cost for each task.

The basic principle of the CAST auction is a time-limited, probabilistic search of the full permutation space that yields good solutions with comparatively small amounts of computation and communication. The steps of this approach are as follows:

1. All agents compute their cost tables for the task independently.
2. Agents sort their choice lists in descending order of task cost.
3. Each agent internally computes the bid order for the current round of bidding.
4. The current bidder selects their best choice task and broadcasts the selected task and associated.
5. On receiving a broadcasted selection, the selected task is removed from the task pool and the task cost is added to the accumulated cost for this round.
6. If the end of a round is not reach, goto 4, else 7.
7. Repeat the auction with a different bid order as needed. If the cost is less, the new task mapping is adopted.

Table 4.2 is a summary of the simulation parameters used in implementing the CAST auction algorithm. Each agent is capable of two actions, *Receive-Bid* which listens for a bid broadcast, and *Bid* which selects the best task available with respect to the bidding agent's preferences and broadcasts that choice. Each

agent is able to determine if it is their turn to bid through the binary sensor *Is-Turn-To-Bid* which returns true if it is the agent's turn to bid and false otherwise.

Number of Agents	10, 50
Number of Tasks	10, 50
Number of Rounds	1, 10, 50, 100
Actions	Bid-on-Task
Task Evaluation Criteria	Cartesian distance

Table 4.2: Summary of CAST Auction simulation parameters

In order to compare results to those found in [21], simulations using [10 agents, 10 tasks] are examined. For each instance of the problem, the position of both the agents and the tasks were randomized. The results of the CAST auction implementation are consistent with the results found in [21], with near-optimal solutions being found within 10 bidding rounds. In order to further show the correctness of the CAST auction implementation, results from the simulations are compared against the assignments obtained using the Hungarian Method, a centralized linear programming solution to the assignment problem that finds optimal assignment in $O(n^3)$ time. Assignment problems for [10 agents, 10 tasks] and [50 agents, 50 tasks] are solved using both the CAST auction and the Hungarian Method, with the CAST auction successfully finding assignment solutions that are within 90% of the optimal solution.

4.3 Chemical Cloud Detection with UAV Swarms

In order to further test the the simulation capabilities of SWEEP, a real-world problem was selected for simulation and modeling within SWEEP. The scenario chosen involves unmanned air vehicles attempting to coordinate in order to detect a chemical cloud.

4.3.1 Introduction

Swarm algorithms can coordinate UAVs in situations that require a level of flexibility and robustness unattainable by a single, more complex air vehicle controller. Additionally, achieving effective use of UAVs requires more than just controlling a single vehicle; the larger problem, with a potentially greater payoff, resides in establishing coordinated behavior of many such vehicles [8]. While much work has been done applying emergent behavior and swarms to a wide variety of problems [20, 5, 19, 3, 13], there has been a deficiency in simulating realistic air vehicles within swarms. Air vehicles have been modeled as particles [27], as objects that can arbitrarily move within a hexagonal grid [24] or as detailed *individual* representations of actual aircraft [14]. This simulation considers both issues: *swarms* of air vehicles that *respect the physical realities of their flight envelopes.* The model of the flight envelope is both general enough and realistic enough

to apply to a wide variety of UAVs. Also, an equivalent of randomized motion within motion constraints is defined and is used to build simulations of UAV missions that exhibit emergent behavior.

The main simulation scenario used in the development and evaluation of UAV swarm control strategies was that of a chemical cloud threat, as shown in **Figure 4.2**. In this scenario, a UAV swarm is assigned a region to patrol, with the objective being to determine if a chemical cloud is present. The swarm is deployed from a tower-like structure, which could include the roof of a tall building. Once deployed, the UAV swarm executes various distributed search algorithms. If a cloud is detected, many actions can be taken, such as attempting to map the size and density of the cloud. Also, if a UAV does find the cloud, and subsequently becomes contaminated, it must not return to the home base, but to a predefined decontamination center.

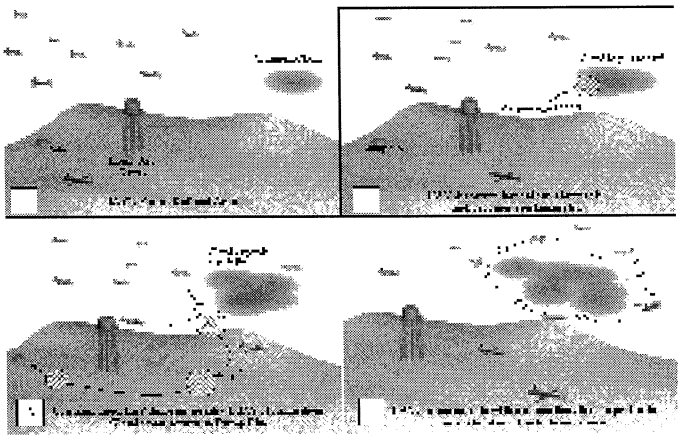


Figure 4.2: The different stages of a chemical cloud detection mission: deployment, detection, and mapping.

4.3.2 System Overview

In order to aid the development and simulation of emergent behavior strategies for UAVs, an optimal path planner that respects the UAV flight envelope is used. The optimal path planner uses simple geometric operations to construct the optimal flight path between two points for a UAV traveling at a constant speed. Given the destination and desired final heading, the optimal path planner constructs a control history for the UAV [28].

Postulated UAV Configuration Swarm intelligence algorithms focus primarily on emerging complex behaviors from simple agents. With this in mind, the postulated UAVs are as minimalist as possible, thus allowing the developed strategies to be applicable over a broad range of UAVs. The developed algorithms can be easily built upon to leverage the capabilities of more advanced UAVs.

The UAVs are assumed to have a constant velocity of 40 knots. Computing capabilities will be comparable to a 66 MHz computer. Some form of inter-agent communication is assumed to be present, with

no bias towards global communication capabilities. The UAVs will have modular sensor arrays that can accommodate specific mission requirements. The battery or fuel cell for the UAVs should provide at least one hour of operational time and be able to be recharged while detached from the craft.

In this 2D model (**Figure 4.3**), the UAV is a point mass with its state specified by (x, y, θ), where (x, y) is the instantaneous position and θ is the heading angle. In addition, ω, which is the time rate of change of heading angle, is a controllable input. It is assumed that the input ω is bounded by $[-\Omega, \Omega]$. Such bounds for the control input are due to the limits on the flight envelop of UAV turning rate and speed. As per the postulated UAV configuration, a constant velocity v_C is assumed.

$$\begin{cases} \dot{x} = v_C \cos(\theta) \\ \dot{y} = v_C \sin(\theta) \\ \dot{\theta} = \omega \end{cases}$$

Figure 4.3: UAV Flight Model

4.3.3 Search Algorithms

UAV swarms are well-suited for tasks involving reconnaissance due to their inherent parallelism and highly robust nature. A monolithic air reconnaissance platform can only observe a single area at a time, and no information can be obtained if the aircraft is disabled. These restrictions do not apply to UAV swarms. The parallel nature of a UAV swarm can allow multiple areas to be observed by a single swarm, but also allows for continued reconnaissance even if multiple members of the swarm are disabled.

Restrictions The largest obstacle encountered when designing search strategies for UAV swarms is managing the physical restrictions of the UAV. There are four main restrictions when it comes to UAV performance: flight envelope, radar and communication, on-board processing power, and fuel supply.

The flight envelope of a UAV is very restrictive. The cruising speed and turning radius of the UAV are such that even the most basic search strategies need to account for these factors. For example, if the objective is to locate or track a fast moving object, the total area being simultaneously searched may have to be reduced in order to improve the overall accuracy of the swarm due to the low cruising speed of the individual UAVs.

Typically, in reducing the size and cost of a UAV, the communication and radar capabilities of the UAV are comparably diminished. These restrictions greatly impact emergent behavior search strategies attempting to magnify the capabilities of the swarm. For example, a swarm of UAVs with low-resolution sensor footprints could locate an object smaller than their sensor footprint by merging their individual sensor readings while still only requiring minimal communication overhead [8].

Search strategies for UAV swarms must expect limited on-board computation capabilities, thus complex calculations will be either too slow or impossible to perform in a real-time environment. Therefore, highly computational search strategies will not be very effective.

Perhaps the most severe restriction is battery life. If a UAV can only be operational for one hour before its power supply is completely drained, then one knows how sufficiently efficient and reliable a search strategy must be.

Randomized Search

Randomized searching is the most basic search strategy capable of being implemented on a UAV swarm. Each UAV chooses a random heading and a random amount of time, and then proceeds to fly in that direction for that duration. The operational requirements for randomized searching are minimal. No communication capabilities need be present on-board the UAV. The randomized search is adaptable for any type of flight envelope and sensor array. As can be expected from any distributed, randomized algorithm, results can only be guaranteed in a probabilistic manner. When the number of UAVs is low, emergent behavioral algorithms suffer and perform no better than any other strategy.

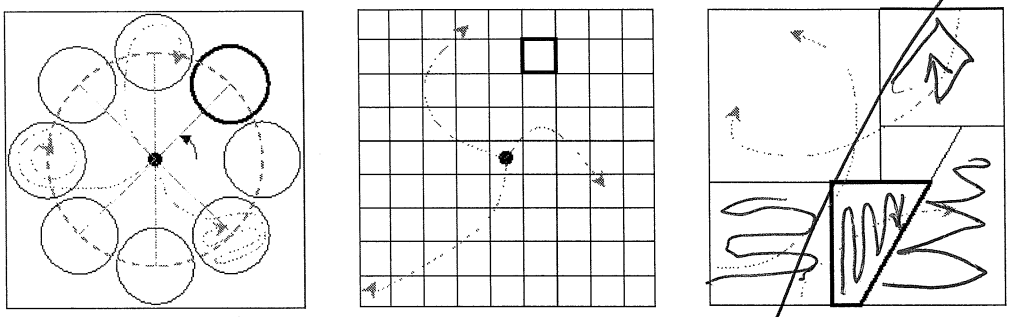
Symmetric Sub-region Search

A symmetric sub-region search divides the search area into equally shaped and sized regions, and then assigns one or more agents to each region, as shown in **Figure 4.4(a)**. Symmetric sub-region searching is useful when there is little a-priori information about the target (i.e. size, location), and when there are a relatively large number of agents as compared to the size of the region to be searched. For example, if there exists the threat of a chemical cloud, a symmetric sub-region approach could prove effective. Since the only information known is the wind heading and the fact that a threat may exist, symmetric sub-regions allow for a higher probability that a threat will be detected due to all of the simultaneous searches. This search strategy is only effective when each UAV is assigned a search area proportional to its sensor capabilities and physical characteristics, otherwise large, unexamined gaps will remain.

Asymmetric Sub-region Search

An asymmetric sub-region search divides the search area into sub-regions of varying shapes and sizes, and then proceeds to assign one or more agents to each region, as shown in **Figure 4.4(b)**. Asymmetric sub-region searching can be useful when more detailed information about the search area is available. Unlike the symmetric sub-region search, this approach can leverage the power of relatively few UAVs by defining sub-regions according to the probability of finding a target, then assigning agents to those sub-regions. If

the possibility of threats exists in any parts of the search area, then fewer agents can be assigned to high probability regions, leaving extra agents to better search low probability areas. This logic may seem counter intuitive, but since the sub-region probabilities are calculated based on a-priori knowledge that targets will be in certain areas, low probabilities really mean that less information is known about that region as compared to other regions, so more agents need to be placed there.



(a) Symmetrical Search (b) Another Symmetricsl Search (c) Asymmetrical Search

Figure 4.4: Examples of symmetrical and asymmetrical search algorithms for UAVs

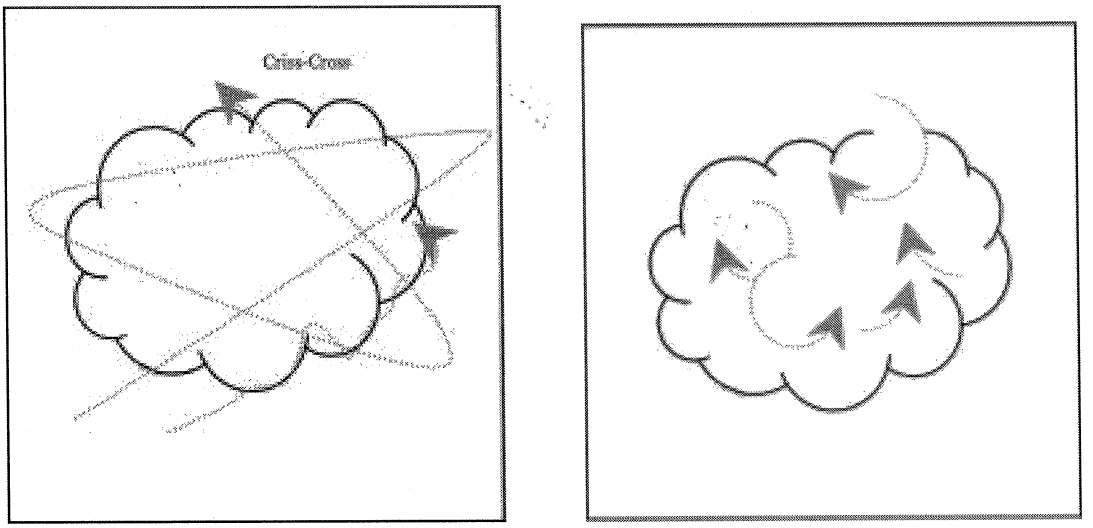
4.3.4 Cloud Detection

UAV Restrictions The chemical cloud scenario imposes many restrictions on UAV swarms. Generally, UAVs are capable of flying at speeds much greater than the average wind speed, thus causing a problem in cloud mapping because a UAV can easily out pace the cloud. The clouds involved in this scenario generally range from $1/2$ km to 3 km in length. The cloud is relatively small as compared to the 10 km^2 region that must be patrolled. Since our swarms only consisted of 5 to 15 UAVs, detecting clouds of this size reliably made this scenario a good driving problem. Battery life was the largest restriction on the UAVs' capabilities. The UAVs evaluated had $1/2$ hour of operational time, thus making it critical to manage flight time with respect to returning to base or a decontamination center.

Various enhancements to the UAV control schemes and search patterns arose from the restrictions placed upon the scenario. In order to improve the chances that a UAV swarm will successfully detect a chemical cloud, different search strategies were considered with the modification that UAVs would search in manners that cross-cut the wind, thus leveraging the wind as much as possible. For example, the symmetric sub-region search would have each UAV search their sub-region using some kind of cross-cutting pattern, shown in **Figure 4.4(c)**. The random search strategy would have each UAV fly, and then at a randomly chosen moment cross-cut the wind for a time, then continue a random flight. An added bonus of cross-cutting the wind is that the UAVs can easily adjust to changing wind conditions by simply changing their cross-cutting vector.

Once a chemical cloud is detected by the UAVs, its location and heading are known. By switching to alternative behavioral rules, the swarm can begin to ascertain the cloud's dimensions and density. With all types of mapping algorithms, the data collected must be adjusted with time to take into account for the movement and diffusion of the cloud. Either this can be done at a central location, or if capable, each UAV can manipulate the data itself. When considering algorithms for cloud mapping, the type of sensors that the UAVs are equipped with must be taken into account. Binary sensors, which deliver a *chemical present/absent* signal, were assumed without loss of generality. There are two types of mapping strategies we considered: inside-outside and dispersion.

The inside-outside method is very straightforward, as illustrated in **Figure 4.5(a)**. If an agent is inside a cloud, it chooses a direction and flies until it is outside of the cloud. Once outside of the cloud, the UAV chooses a point randomly offset from the last intersection with the cloud, and then flies back in. The points where the agent exits or enters the cloud, transition points, are then recorded. From this data, a model of the cloud size can be extrapolated.



(a) Inside-outside mapping

(b) Dispersion mapping

Figure 4.5: Examples of inside-outside and dispersion cloud mapping algorithms for UAVs

Dispersion algorithms can very effectively be used to map chemical clouds. Essentially, once a UAV detects the cloud, it broadcasts this information to all surrounding UAVs. After re-broadcasting the received message, the UAVs converge to the general area that the cloud was initially located and begin executing a dispersion algorithm. The agents execute the algorithm with only one modification; the UAVs continue to expand the dispersion area until a UAV is out of the cloud. At this time, the UAVs outside the cloud begin moving back into the cloud, as seen in **Figure 4.5(b)**. Thus, the shape of the cloud emerges.

4.3.5 Results

Chemical Cloud Mapping

In order to demonstrate the capabilities of emergent behavioral strategies, a UAV rule-base was constructed for mapping a chemical cloud. The scenario used 10 UAVs in a bounded 10 km^2 region. Initially, a single UAV was placed in close proximity to the cloud while the rest of the UAVs were randomly distributed throughout the region, thus simulating an end-game situation for cloud searching. Quickly, a UAV finds the chemical cloud and broadcasts the location of the cloud. Upon receiving the broadcasts, the UAVs swarm to the general region where the cloud was found and begin the mapping process. For simulation purposes, the inside-outside mapping method, as previously described, was implemented. For purposes of simplicity, the cloud was represented as rectangular and immobile. If implemented with a moving cloud, the data collected by the UAV swarm would have to be processed more rigorously in order to obtain the shape of the cloud. **Figure 4.6** shows a plotting of the data the UAV swarm collected in approximately 5 minutes of flight time. As can be clearly seen, the UAV swarm was able to emerge the shape of the chemical cloud relatively quickly. The mapping of the cloud is an emergent behavior. In all three mapping strategies, no agent has any knowledge about the cloud's size, shape, or rate of diffusion. They simply randomly modify their behavior (change flight direction) whenever they detect an inside/outside transition. By recording the time and location of these transitions, the UAVs create a data set that contains all information necessary to accurately depict the cloud's movement over time.

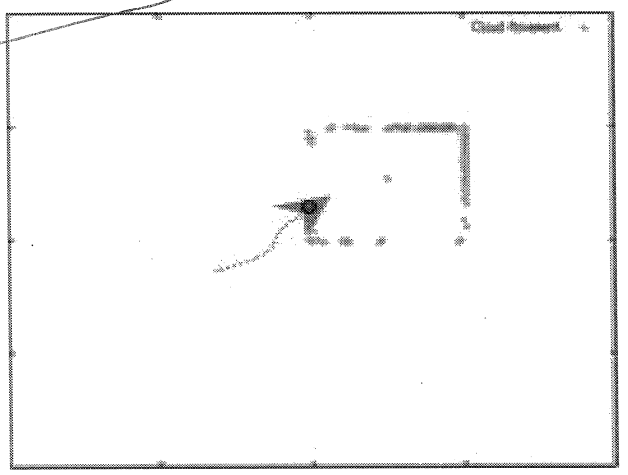


Figure 4.6: The outline of a chemical cloud as mapped by a UAV swarm

Chemical Cloud Searching Scenario

The scenario chosen for the trade-off study was the detection of a chemical cloud by a UAV swarm. The purpose of this trade-off study is to empirically determine the best size for a UAV swarm searching a bounded region for a hostile-released chemical cloud. In this scenario, a UAV swarm is deployed from a central tower in a square region. Since the main objective of the swarm is detection of chemical clouds, it is unknown to the swarm whether there previously existed a chemical cloud in the region they patrol. Once released, the UAV swarm executes a search strategy in order to detect the presence of a chemical cloud. If the cloud is not found before the UAVs' power supplies are depleted, the swarm returns to the tower. When the chemical cloud is observed, the UAV swarm can transition into multiple cloud-mapping behaviors, which report sensor-based information such as cloud size, density, and composition.

The swarm is responsible for patrolling a bounded square region. The intended target is a chemical cloud of unspecified size or type that moves with the wind. The chemical cloud is capable of diffusion, thus there is a fleeting window of time in which the cloud can be detected before the cloud has diffused completely. Also, the cloud diffusion causes the size of the cloud to increase, thus improving the chances of detection, while at the same time causing a decrease in cloud density.

The UAV swarm used a modified random search to detect the cloud. Since the UAVs are capable of detecting wind direction, a wind cross-cutting addition was made to the randomized search. Cross-cutting is defined as flying perpendicular to the wind direction. When en route to randomly chosen destination, the UAV can decide to randomly cross-cut the wind, thus increasing the chances of finding the cloud. The UAV will cross-cut the wind for a non-trivial random amount of time, then resume the randomized search.

Though there are more structured search algorithms, the nature of the scenario lends itself to the randomized search. As Clough states in [8], "random searches are optimal given no a-priori information." Consider the case where the wind speed is 0 m/s, thus the cloud does not move and has minimal dispersion. In this case, more structured search algorithms will outperform the randomized search. Since the cloud does not move, a previously searched location need not be searched again. Thus, information does not become stale, and the swarm can systematically search the region with confidence. In more reasonable cases, the wind is driving a cloud along a path, and since the UAV swarm has no *a-priori* knowledge about the cloud's location, information is only fresh for a wind-speed dependent amount of time. Since the UAV swarms being considered are relatively small (typically between 5-15 agents) and more complex algorithms provided no additional benefit, the trade-off study only examined the performance of the randomized search algorithm.

We examined cases of the chemical cloud scenario with UAV swarms of sizes 5 through 15 in a region 10,000 m^2. Cloud size varied from 1 km to 3 km in length, and 1 km to 1.5 km in width. The size of the cloud was regulated using diffusion and decay parameters. The cloud was simulated as if there existed a

single moving source of the chemical (e.g. a crop duster).

The cloud lengths, beginning at 1 km, were incremented by 0.1 km. For each cloud length, 100 simulations were executed for each discrete swarm size, starting at 5 and incrementing up to 15, for a total of 30,000 simulations. The simulations ran until either the cloud was detected or the swarm ran out of fuel. The time to cloud detection was recorded and presented in **Figures 4.7-4.10**.

As the figures indicate, there is a performance increase when the number of agents is increased for the same-sized cloud. **Figure 4.7** represents the normalized amount of time taken by a UAV swarm to locate a chemical cloud. The search times were normalized against the total amount of time with which the UAV swarm could have searched. As expected, larger swarms were able to find similarly sized chemical clouds faster than smaller sized swarms.

Figure 4.8 represents the percentage of times that the UAV swarm was able to successfully detect the chemical cloud. An increase in the number of UAVs in the swarm increases the swarm's chances of finding the chemical cloud because probabilistically speaking, more of the territory is being covered.

Figure 4.9 illustrates the average hit percentage of a UAV swarm of size n for any sized cloud. **Figure 4.10** represents the average amount of time taken by a UAV swarm of size n to find any sized cloud. As can be seen, even with the maximum number of agents, the chances of a UAV swarm finding a cloud of indeterminate size is 83%. This performance rating may be acceptable for some situations, for example, if the UAV swarm is used as an early detection system. As shown in **Figure 4.9** and **Figure 4.10**, there exists a linear improvement in the performance of the UAV swarm with the addition of ~~only one~~ each more agent.

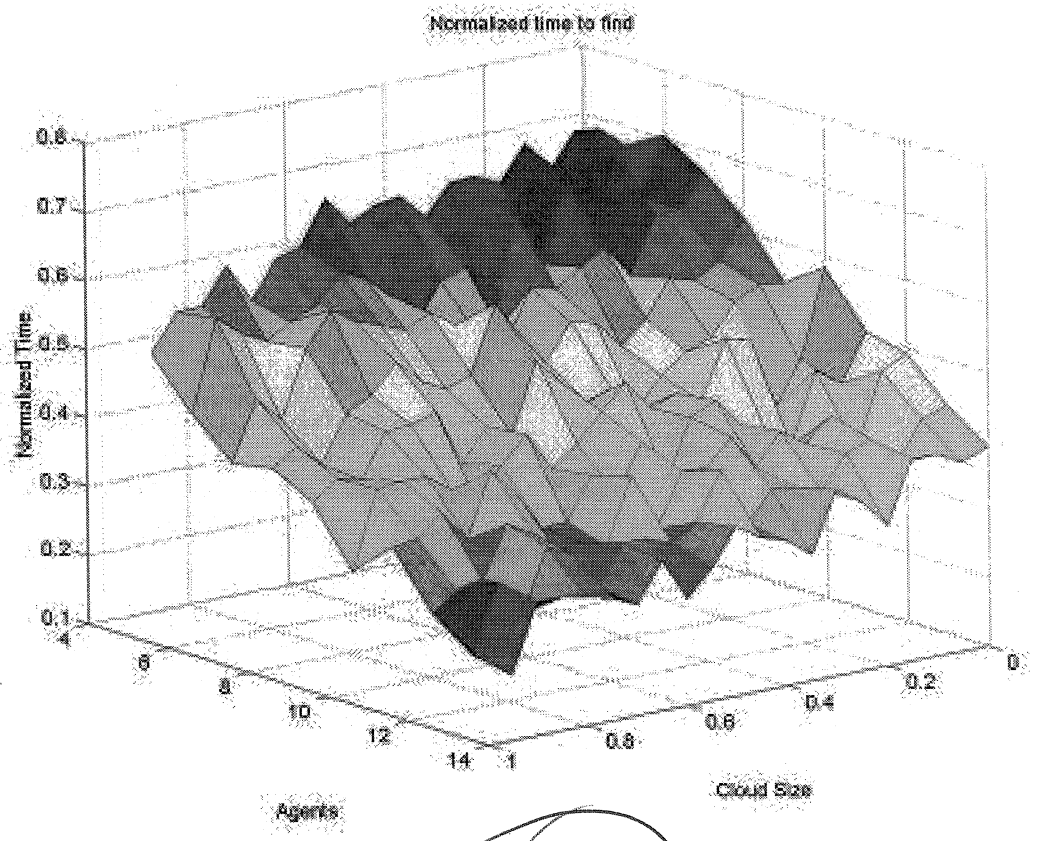


Figure 4.7: graph-1

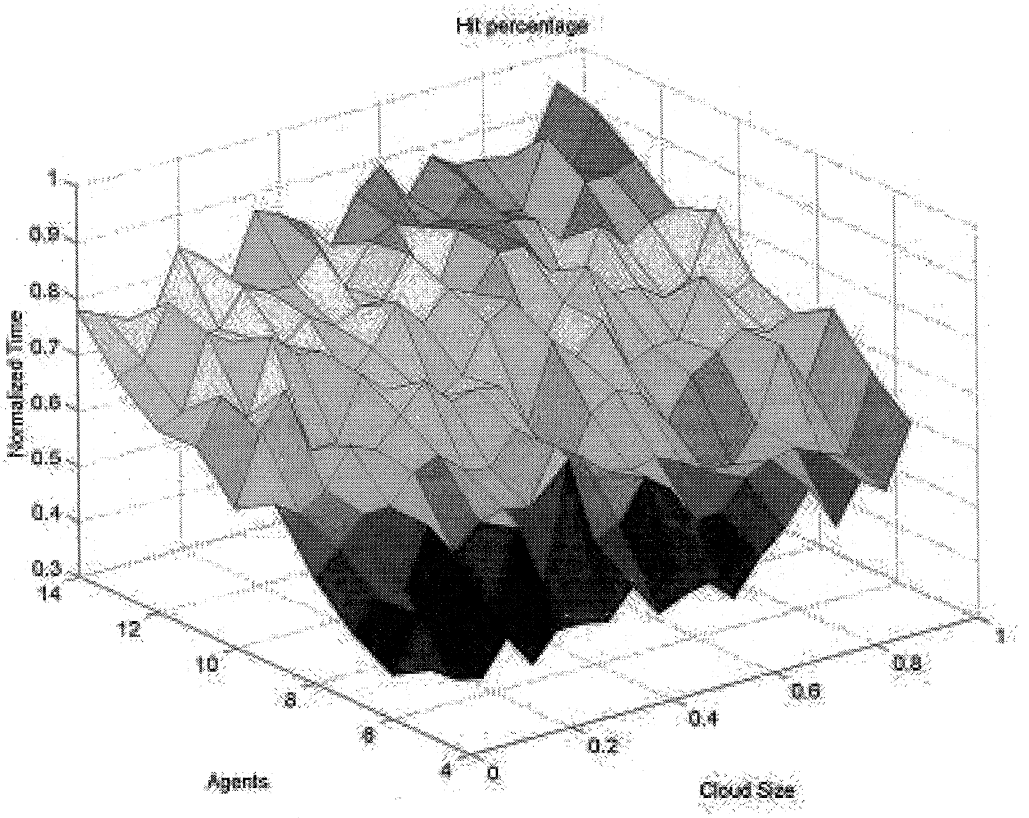


Figure 4.8: graph-2

PUT W/ 4.7 ON SAME PAGE (RESIZING IF NECESSARY)

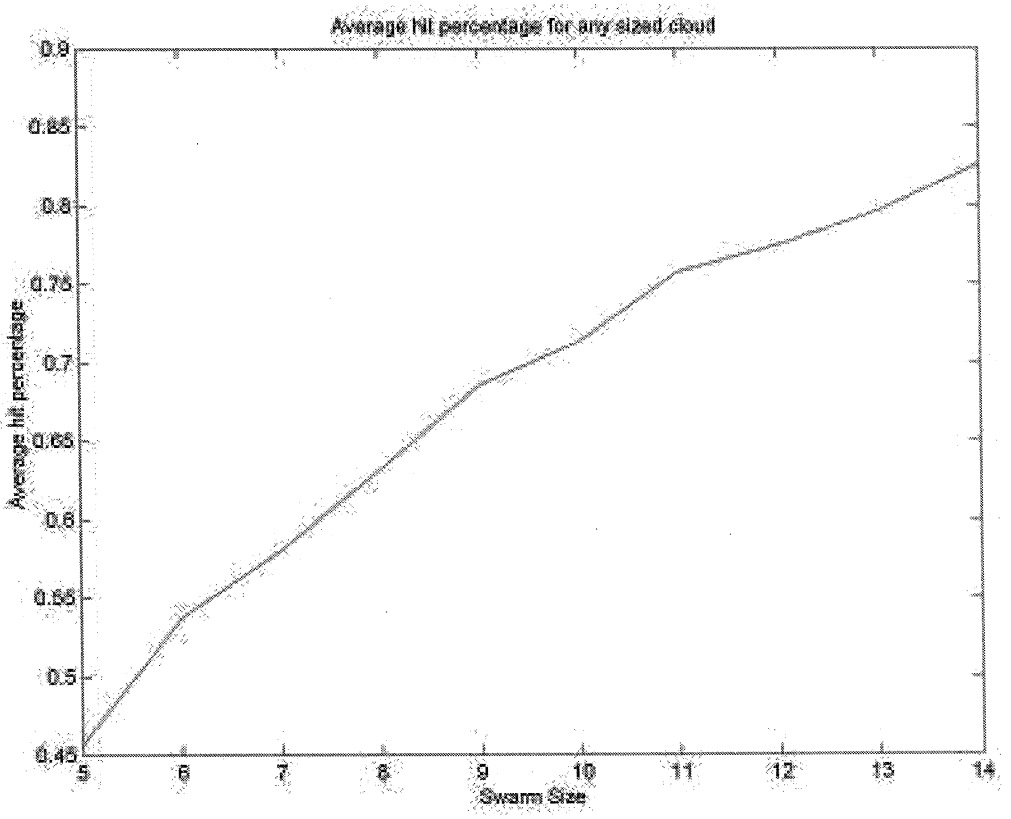


Figure 4.9: graph-3

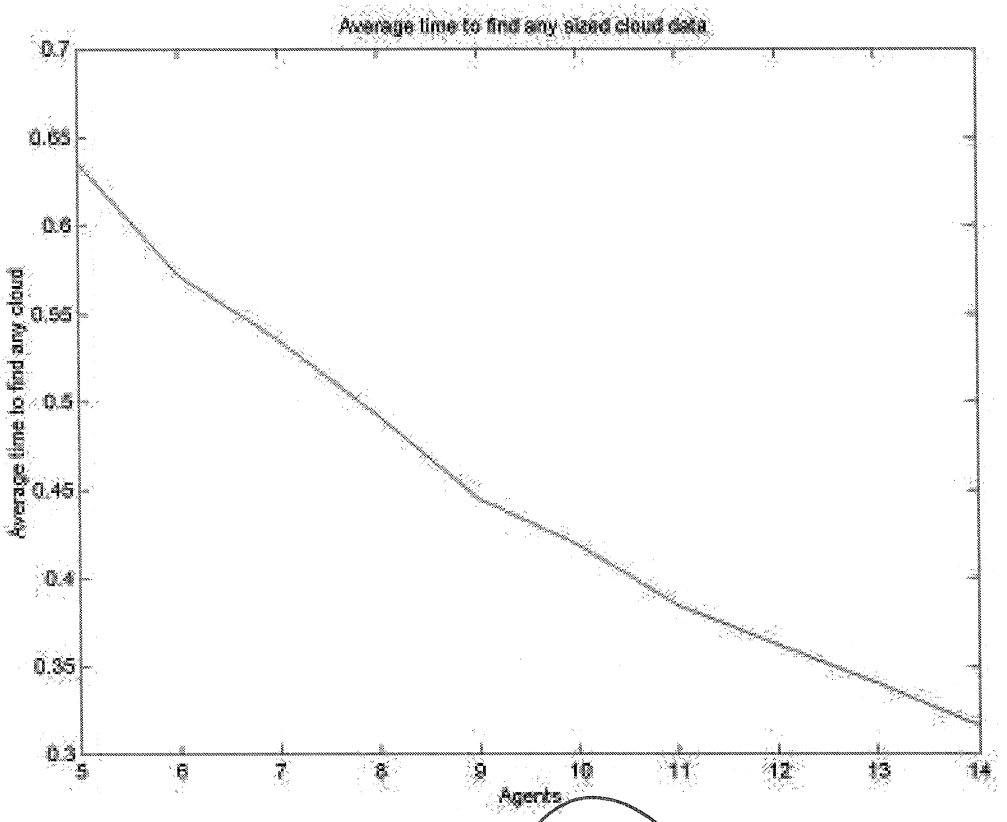


Figure 4.10: graph-4

Chapter 5

Evolutionary Generation of Swarm Behaviors

5.1 Introduction

Swarms are complex systems whose behavior is difficult to determine *a priori* regardless of the predictive measures used. The full simulation of a swarm algorithm is currently the best known method of evaluating both the expected and emergent behaviors of the algorithm. Thus, it is not surprising to find that the current state of the art in swarm algorithm development is primarily relegated to a trial-and-error methodology. The realization of desired high-level behaviors for a swarm without the need to define and program the necessary lower-level behaviors would be an invaluable tool to both programmers and non-programmers alike.

This chapter describes the development of ECS (Evolutionary Computing for Swarms), an evolutionary computation engine designed to evolve swarm behavior algorithms. The overall goal of ECS is the realization of an evolutionary computing architecture that is capable of automatically generating swarm algorithms that exploit the inherent emergent properties of swarms with minimal user interaction.

5.2 System Overview

Figure 5.1 is a conceptual overview of ECS. The system is a composition of a component-based evolutionary computing framework and the SWEEP swarm algorithm simulator. From a high-level perspective, ECS operates similarly to a traditional evolutionary computing system. An initial population of candidate solutions, represented as finite state machines, are randomly generated. Each solution in the population is then evaluated using SWEEP, resulting in a fitness score for each solution. The next generation of the pop-

ulation is produced through the application of reproduction and mutation operators on individual solutions selected by criteria such as relative fitness or random selection. Finally, the entire evaluate-reproduce cycle is repeated until a solution is found that satisfies the termination criteria.

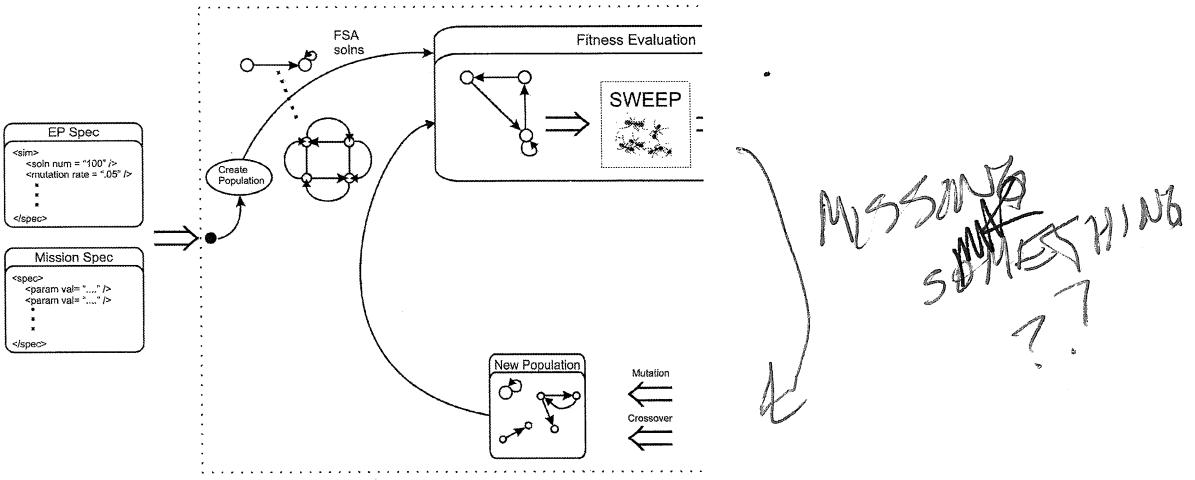


Figure 5.1: Conceptual overview of ECS.

5.3 Solution Representation

In ECS, a solution is encoded as an extended Mealy state machine, where that state machine encodes the program for a single agent in a homogeneous swarm. Instead of using input symbols to trigger the firing of transition, the firing of each transition is decided by a boolean condition composed of a sensor value and a target sensor value. The condition is evaluated to true when the actual sensor condition matches the target sensor condition, resulting in the firing of the transition. When a transition fires, an action associated with the agent is triggered, and the state machine goes to the next state as dictated by the triggered transition.

There are many reasons why the state machine representation is the native representation for candidate solutions. First, the state machine representation is simple but yet capable of expressing complex logic. Also, the state machine representation is inherently graph-based, thus previous work involving evolutionary computing using graphs can be leveraged, especially with respect to defining evolutionary operators. Secondly, the state machine encoding is particularly enticing as its structure is robust to random modifications,

Within the evolutionary computing framework, the candidate solutions are stored as XML objects representing valid SWEEP controller structures. The benefit of this representation is two-fold. First, the need for complex encoding/decoding of solutions at evaluation time is eliminating as a simple serialization of the object results in a valid XML file in the SWEEP state machine format. And second, mutation implementation is simplified as genetic operators manipulate the solution directly by using XML libraries, thus eliminating

the possibility of accidentally creating malformed XML content.

5.4 Evolutionary Operators

The mutation operators defined for this work focus on the modification of the primary components of the state machine representation: states, actions, and transitions. **Table 5.1** lists the mutations defined in this work.

Mutation	Description
AddState	Add a new state and randomly connect to the state machine
AddTransition	Add a new transition to a state
ChangeNextState	Change the next-state attribute of a transition
ChangeAction	Change the action attribute of a transition
InvertSensor	Invert the sensor condition on a transition

Table 5.1: Mutations defined for a SWEEP state machine solution encoding

The *AddState* and *AddTransition* mutations apply large changes to the structure of a candidate state machine solution, allowing for better exploration of the solution space. Without these mutations, the number of states and transitions in a solution would be fixed, thus purely relying on serendipity to randomly generate a state machine with a structure appropriate for the problem. The mutations *ChangeNextState*, *ChangeAction*, and *InvertSensor* apply finer-grained changes to state machine solution, thus allowing state machines to be debugged and tuned through many applications over several generations.

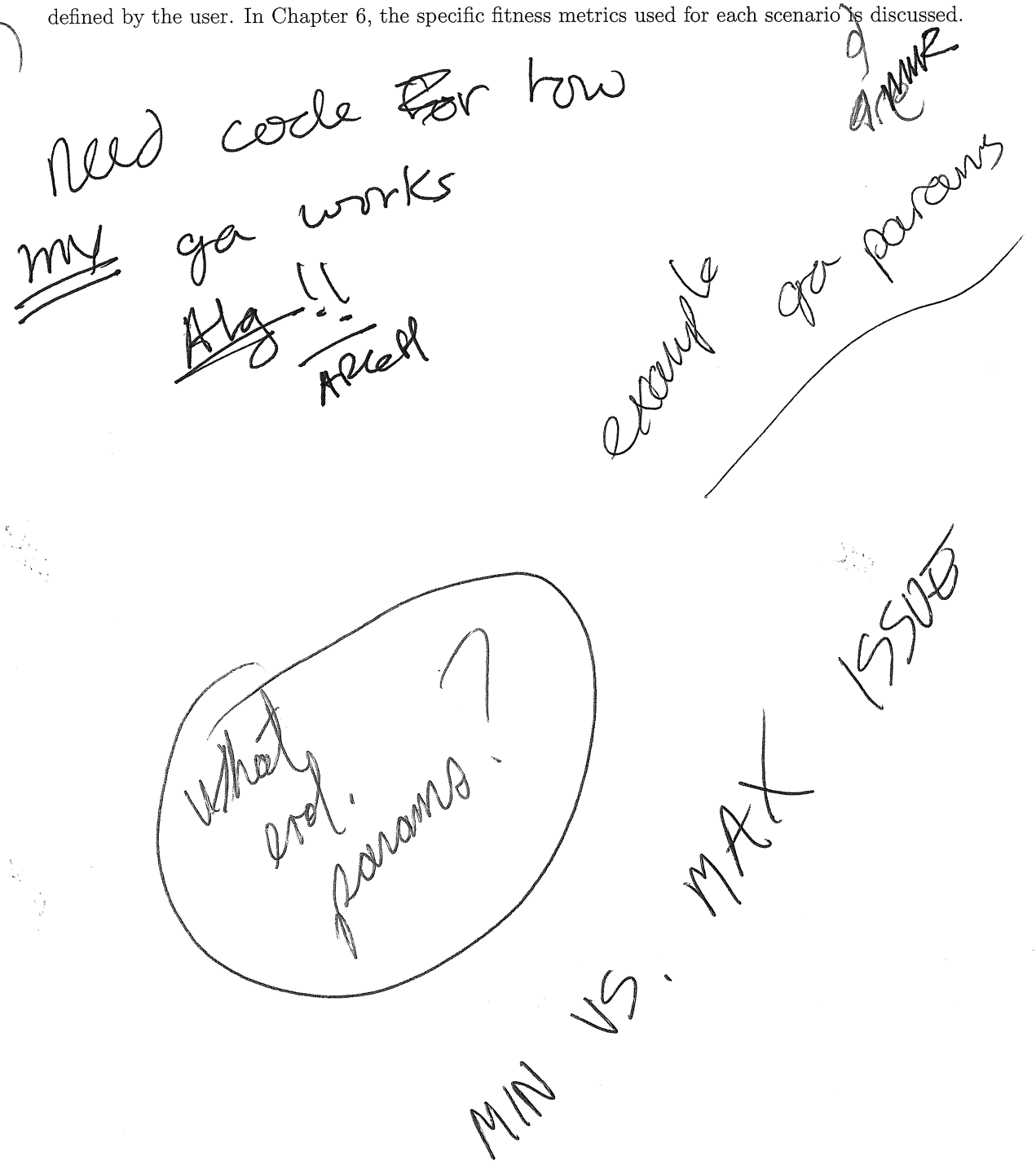
5.5 Fitness Evaluation

The driving force in an evolutionary computation system is the fitness function because it imposes an ordering on the population of solutions, with ranking determined by the aggregation of each solution's ability to satisfy some set of metrics. Thus, the fitness function acts as a differentiator, separating the better from the best.

The idea is that the fitness function guides selection and mutation through a selective pressure that manifests through a solution's score relative to the score of the entire population. Thus, a good fitness function filters a population on two levels: separating "bad" solutions from "good" solutions, and by differentiating "good" solutions from "better" solutions.

Each solution in the population is a state machine that encodes the program for a single agent in a homogeneous swarm. Thus, simulation must be used in order to evaluate the fitness of a solution. For this work, SWEEP is used to simulate a homogeneous swarm where each agent is programmed with the logic encoded in the state machine solution. The simulation is then run multiple times to remove any biases introduced by the randomness of a swarm. The fitness function takes the raw data generated by SWEEP,

which is customizable and application specific, and calculates a fitness score based on the evaluation metrics defined by the user. In Chapter 6, the specific fitness metrics used for each scenario is discussed.



Chapter 6

Results of Evolving Swarm Behaviors

The goal of this chapter is to develop several proof-of-concept scenarios showing how evolutionary computing can be leveraged to design swarm algorithms. This chapter presents the results of using the evolutionary computing system outlined in Chapter 5 to breed swarm algorithms for two basic swarm functionalities, swarm dispersion and object manipulation. The goal of swarm dispersion is for every agent in the swarm to satisfy some conditions of the positions resulting in a swarm of a certain density. The swarm dispersion problem is relatively simple, but serves as a proof-of-concept of the functionality of ECS. The second problem examined is object manipulation where agent attempt to collect one type of object and destory a different type of object. As there are many different approaches that may be taken to address the object manipulation problem, thus providing an adequate final scenario to test the abilities of the ECS.

Each section in this chapter is broken down into three subsections: description, implementation, and results. The first section gives a detailed overview of the problem and the scenario. The second section outlines the agent configuration and parameters used for the evolving the target behavior. The third section presents the results of evolving the target swarm behavior.

6.1 Swarm Dispersion

This section discusses the process of evolving a swarm behavior for agent dispersion. Physical dispersion is a fundamental swarm behavior that has many applications including establishing perimeters and performing searches. Additionally, dispersion serves as a basis for more complex swarm behaviors as it provides a flexible method of controlling the physical configurations of a swarm without the need to exert control over each individual agent.

The goal of a dispersion algorithm is for a swarm is to achieve a physical configuration that satisfies

one or more constraints, such as average neighbor distance, nearest-neighbor distances, or neighborhood densities. Since both an agent's physical dynamics and the dispersion criteria may not be static, the physical configurations achieved by a dispersion algorithm can be either static or dynamic in nature.

6.1.1 Implementation

For this work, a swarm of agents are deployed in a bounded environment. In general, the initial physical configuration of the swarm is arbitrary, so a random distribution in a bounded subregion is used (see Section 4.1). The goal of the swarm is to expand and/or contract to achieve a specific density, specified in terms of distances between individual agents. More formally, an agent has satisfied the dispersion criteria if all of their neighbors are at least d_{min} units away, but not more than d_{max}, where $d_{min} < d_{max}$ (see **Figure 6.1**). Thus in this case, a swarm achieves a successful dispersion when all agents meet the defined minimum and maximum neighbor distance criteria.

Dispersion example figure

Figure 6.1: For an agent to satisfy the dispersion criteria, all of their neighbors are at least d_{min} units away, but not more than d_{max}, where $d_{min} < d_{max}$

The environment for the simulation is a bounded grid-based environment. The agents possess limited mobility and sensing capabilities. Each agent can move in any of the four cardinal directions at a uniform speed. Additionally, an agent is also capable of moving in a random cardinal direction or not moving at all. Each agent is only capable of sensing the presence of agents in their Conway neighborhood.[1] A neighbor sensor returns a boolean value indicating when the neighbor conditions are satisfied (true) or not satisfied (false) for a given region around the agent.

The agent behavior primitives available for state machine construction are: *move-up*, *move-down*, *move-left*, *move-right*, *move-random*, and *move-none*. The agent sensor primitives available for constructing state transitions are: *neighbor-up*, *neighbor-down*, *neighbor-left*, *neighbor-right*.

The raw fitness of a solution is measured by the total number of agents violating the dispersion criteria with respect to another agent. For example, if four agents are arranged in a square, the fitness score would be 12 because each agent is too close to the other three agents, thus violating the dispersion criteria for those three agents. A worst-case scenario is where every agent is violating the dispersion criteria of every other agent. If n is the total number of agents, this scenario establishes a maximum error, $MAX\_ERROR = n * (n - 1)$. So, for the scenario examined here with 100 agents, $MAX\_ERROR = 100 * 99 = 9900$. MAX_ERROR is then used to normalize the fitness score of each solution. Since the fitness function is also

[1] A Conway Neighborhood is defined as the set of squares adjacent to a given square either vertically, horizontally or diagonally.

an error function, the goal of this scenario is to minimize the error to 0.

The evolved solutions are scored using scenarios with randomized agent positions. **Table 6.1** shows the parameters used for evolving a swarm algorithm to address the dispersion problem.

Parameters	
Objective	Dispersion
Max. Generations	500
Population Size	73
Mutations	
Change-Sensor-Value	top 6 + 2 random
Change-Action-Value	top 6 + 2 random
Change-Next-State	top 6 + 2 random
Add-State	top 6 + 2 random
Add-Transition	top 6 + 2 random
Actions	
move-up	
move-down	
move-left	
move-right	
move-random	
move-none	
Sensors	
neighbor-up	
neighbor-down	
neighbor-left	
neighbor-right	
Simulation	
Number of Agents	100
Environment	50x50 grid
Maximum time	400

Table 6.1: Parameters for evolving dispersion

6.1.2 Results

In all of the evolutionary runs, a general fitness trend is observed. Initial generations produce very poor results, with most candidate solutions scoring near the very bottom of the scale. Gradual progress is then made for a short period, then a dramatic jump in fitness occurs. The process of spiking and stabilizing continues one or more times until a solution with the target fitness value is evolved. The characteristics of the fitness over time is not typical of traditional evolutionary computing fitness trends, which have a steady improvement as generations progress. One large contributing factor is the granularity of the fitness function. Traditional fitness functions produce real-valued fitness measures, thus small improvements introduced through mutation manifest themselves as slight improvements in fitness. In this case, since the fitness function is integer based and thus discrete valued, the slight improvements introduced through mutation may be overlooked. Thus, the fitness function is coarse-grained and does not provide enough information to

effectively drive the selective pressure of evolution [2]. Fortunately though in this case, the fitness function as defined does produce enough information to drive evolution towards an acceptable solution.

A typical evolutionary run for dispersion is shown in Figure 6.2. This run shows a steady increase in fitness until generation 20, indicating that the solutions are progressively becoming better at dispersing the agents. However, after generation 20, there is a sharp decline in both the best and average scores, indicating that one of the evolutionary operators made a change in one of the solutions that resulted in a correct solution to the dispersion problem.

A particularly enticing application of evolutionary computing to swarm algorithm development is the construction of standard algorithms for cases where the agents are disabled or obstructed in some way beyond their control, such as having intermittently faulty sensors. In these cases, the inherent logic of the swarm algorithm must mitigate the deficiencies of the agent. For example, imagine a scenario where agents cannot hold their position. In this case, a static dispersion is not possible, thus a dispersion algorithm that can create a dynamic dispersion equilibrium is required. Figure 6.3 shows the fitness evolution over time of a run attempting to evolve a dynamic equilibrium dispersion algorithm. The behavior of the fitness over time demonstrates a similar asymptotic trend for the first half of the run in both the best and average fitness scores as compared to the results from Figure 6.2. After generation 25, the best fitness score undergoes several fast and sharp drops, indicating that the evolutionary operators are making useful modifications to the existing solutions. Note that in this run, the system settles far above the optimum score of zero. This is a result of the agents lacking the ability to hold their position. Since each agent is required to be in constant motion, the resulting algorithm not only has to navigate an agent towards a criteria satisfying configuration at the next step, but the solution must also simultaneously avoid bad configurations.

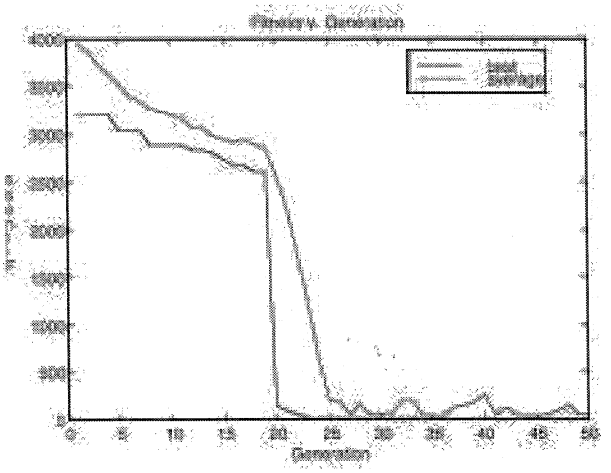


Figure 6.2: Best and average fitness scores resulting from evolving a dispersion algorithm.

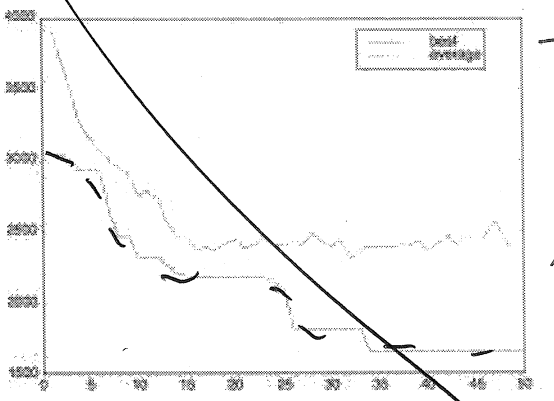


Figure 6.3: Best and average fitness scores from evolving a dynamic equilibrium dispersion algorithm.

6.2 Object Manipulation

In the object manipulation scenario, there are two types of objects scattered throughout the environment, objects for collection and object for destruction. Thus, the swarm has two simultaneous goals: (1) locate all collectible objects and return them to a defined goal location and (2) locate and destroy all objects marked for destruction.

Object manipulation presents many interesting challenges for a swarm. Different types of coordination are required to address each of the separate tasks, that being collection and destruction. Object collection requires the development of a primarily independent agent strategy, whereas object destruction requires the development of a cooperative strategy. From a theoretical perspective, object manipulation is interesting because there is not a trivial solution. In order for the situation to be addressed, the swarm must first coordinate to find the objects, then decide how to handle each object. There are many different approaches that may be taken to address the object manipulation problem, thus providing an adequate final scenario to test the abilities of the evolutionary system developed in this work.

This section discusses the evolution of the basic object manipulation swarm algorithm. A three phase approach will be taken for the evolution of an object manipulation algorithm. First, the evolution of an object collection algorithm and an object destruction algorithm will be addressed separately. Even though both are similar, there are salient features of each problem that will need to be accounted for separately. Then, leveraging the work from evolving the object collection and destruction algorithms, an algorithm for the full object manipulation problem will be undertaken.

6.2.1 Implementation

This section will separately discuss the implementation details used for both the object collection and object destruction scenarios. It is assumed that the object manipulation scenario will use the combined implementations for object collection and destruction.

Scenario Description

There are two types of objects randomly scattered throughout the environment, objects to be collected (type C) and objects to be destroyed (type D). The type C objects are to be located and return to a predisposed goal repository known to all the agents. The type D objects require two separate "attacks" to be completely destroyed. As a by-product of an agent attacking an object, that agent is disabled for the duration of the simulation. Type D objects have three distinct states: untouched, damaged, and destroyed. All type D objects originate in the untouched state, and must become damaged before they are destroyed. When an

agent first attacks an object, the object transitions from the untouched to the damaged state. When another agent attacks an object in the damaged state, the object transitions to the destroyed state.

The agents in this scenario are free-moving and have the ability to randomly wander. The initial physical configuration of the swarm is arbitrary, thus a random distribution through the environment is assumed. An agent is able to sense and differentiate between type C and D objects. Additionally, the ability to navigate directly towards an object is incorporated to eliminate the need to evolve path planning capabilities, which is beyond the scope of this work. Finally, an agent has the ability to distinctly broadcast to near-by neighbors the location of sensed objects. When an agent receives a broadcast, the information is treated similarly to that of the object proximity sensor, thus the broadcasted information is compatible with the navigation behaviors.

State Machine Primitives

Table 6.2 and **Table 6.3** provide a summary of the actions and sensors available as state machine building blocks. For both cases, the default behavior when no transition fires is *move-random*. Additionally, each agent is equipt with one sensor that indicates when the agent is on an object, and one sensor that indicates when the agent is near an object. With the implementation used in this work, when an agent is on an object, the agent is also considered near an object.

The agent behavior primitives available for the object collection scenario are: *move-up*, *move-down*, *move-left*, *move-right*, *move-random*, *move-to-object_C*, *move-to-goal*, *pick-up*, *put-down*, and *broadcast_C*. The available sensor primitives are: *near-object_C*, *on-object_C*, and *on-goal*. The agent behavior primitives available for the object destruction scenario are: *move-up*, *move-down*, *move-left*, *move-right*, *move-random*, *move-to-object_D*, *first-attack*, *second-attack*, and *broadcast_D*. The available sensor primitives are: *near-D_object* and *on-D_object*.

Fitness Metrics

Table 6.4 describes the raw metrics that evaluate the performance of a candidate solution with regards to the object collection and object destruction scenarios. Note, the metrics are formulated in terms of measuring error, thus the overall goal is to minimize the error measures.

The metric t measures the execution speed of an evolved candidate algorithm in terms of simulation timesteps. Two metrics are defined to measure the fitness (error) of a candidate object collection algorithm. As described in **Table 6.4**, metric c_2 measures the number of objects not collected, and metric c_1 measures the number of collected objects not returned to the target repository. For the individual solution fitness scores, the metrics are ordered in decreasing importance: c_1, c_2, t.

Behavior	Scenarios		
	Collection	Destruction	Manipulation
move-up	x	x	x
move-down	x	x	x
move-left	x	x	x
move-right	x	x	x
move-random	x	x	x
pick-up	x		x
put-down	x		x
move-to-goal	x		x
broadcast_C	x		x
move-to-object_C	x		x
first-attack		x	x
second-attack		x	x
broadcast_D		x	x
move-to-object_D		x	x

Table 6.2: Behavior primitives available for state machine construction as related to each scenario.

Sensor	Scenarios		
	Collection	Destruction	Manipulation
near-object_C	x		x
on-object_C	x		x
holding-object_C	x		x
on-goal	x		x
near-object_D		x	x
on-object_D(untouched)		x	x
on-object_D(damaged)		x	x

Table 6.3: Sensor primitives available for constructing state transitions as related to each scenario.

Similarly, two metrics are defined to measure the fitness (error) of a candidate object destruction algorithm. Metric d_2 measures the number of objects in the untouched state, and metric d_1 measures the number of objects in the partially destroyed state. For the individual solution fitness scores, the metrics are ordered in decreasing importance: d_1, d_2, t.

As seen in the object collection and object destruction fitness metrics, the ordering of the metrics also coincides with the sequential dependencies inherent in the scenario. In the object manipulation scenario, the sequential dependency problem becomes more pronounced as the sequential dependencies that exist for the individual object collection and destruction subtasks are independent but equally weighted. Since c_1 and d_1 are equally weighted, the ordering of the two metrics skews the evolutionary progress towards the metric given sequential preference. For example, when c_1 is given preference over d_1, the solution pool tends to first evolve experts in object collection, then the system has to modify through mutation an expert object collection algorithm to accommodate for object destruction [16].

Since c_1 and d_1, and c_2 and d_2, are independent but equally weighted, a regular ordering is not enough to adequately characterize the object manipulation scenario. Thus, to address this issue, the raw fitness

Name	Description
c_1	number of objects picked up but not put in the goal
c_2	number of objects not collected
d_1	number of objects in the partially destroyed state
d_2	number of objects in the untouched state
t	number of time steps

Table 6.4: Basic fitness metrics for an object manipulation scenario.

metrics are used to construct a set of composite metrics that do not exhibit the sequentiality conflicts seen between the raw metrics, thus a lexicographic sorting of the solutions is sufficient to establish a fitness-based ordering. **Table 6.5** describes these composite metrics.

Name/Priority	Metric	Description
m_1	$\sim c_1 \wedge \sim d_1$	flag solutions fully performing both collection and destruction
m_2	$\sim c_2 \wedge \sim d_2$	flag solutions partially performing both collection and destruction
m_3	$\sim c_1 \vee \sim d_1$	flag solutions fully performing collection or destruction
m_4	$\sim c_2 \vee \sim d_2$	flag solutions partially performing collection or destruction
m_5	$max(c_1, d_1)$	select the weakest of the two primary subgoals
m_6	$max(c_2, d_2)$	select the weakest of the two secondary subgoals
m_7	$min(c_1, d_1)$	select the strongest of the two primary subgoals
m_8	$min(c_2, d_2)$	select the strongest of the two secondary subgoals
m_9	t	number of timesteps to complete collection and destruction

Table 6.5: Composite metrics used for the object manipulation scenario. The metrics are created from raw fitness data in order to eliminate the conflicting sequential dependencies that exist in using the raw fitness metrics.

In order to create the correct ordering needed for the object manipulation scenario, first a series of boolean flags are defined. These flags ensure that the solutions that meet the flag criteria are ranked higher than those that do not. Thus, high-level logic can be incorporated into the lexicographic scoring, such as the metrics m_1 and m_2 which encode the rule "solutions that act on both subgoals are better than solutions that address only one subgoal." Metric m_3 ensures that a solution that completely addresses one subgoal is ranked higher than a solution that only partially addresses a subgoal, regardless of the raw metric values. Metric m_4 ensures that solutions that partially address any subgoal are ranked higher than solutions that do nothing. Though m_4 is not needed, it is included for completeness. Metrics m_5 and m_6 rank solutions according to their weakest subgoals, focusing on c_1/d_1 first and then c_2/d_2. These metrics are implemented in this fashion because a solution's total fitness is only as good as the weakest component. Similarly, metrics m_7 and m_8 rank solutions according to their strongest subgoals, focusing on c_1/d_1 first and then c_2/d_2.

Table 6.6 summarizes the parameters used in the evolution of solutions for object collection, destruction, and manipulation.

Parameters	
Objective	Object Manipulation
Max. Generations	1000
Population Size	73
Mutations	
Change-Sensor-Value	best 6 + 2 random
Change-Action-Value	best 6 + 2 random
Change-Next-State	best 6 + 2 random
Add-State	best 6 + 2 random
Add-Transition	best 6 + 2 random
Actions	
move-random	
move-to-object_C	
move-to-goal	
move-to-object_D	
pick-up	
put-down	
first-attack	
second-attack	
broadcast-object_C	
broadcast-object_D	
Sensors	
on-object_C	
near-object_D (range = 5 grid units)	
on-goal	
holding-object_C	
on-object_D(untouched)	
on-object_D(damaged)	
near-object_D (range = 5 grid units)	
Simulation	
Number of Agents	100
Environment	50x50 grid
Maximum time	400
Number objects type C	50
Number objects type D	30

Table 6.6: Parameters for evolving object collection, destruction, and manipulation

6.2.2 Results

Viable solutions to each of the three scenarios outlined are successfully evolved. The performance for the object destruction and object collection scenarios are very similar, with each finding a solution that completely addresses the secondary subgoal for the scenario. Then within 25 generations a complete solution is found, with a somewhat time optimized solution being evolved within 20 generations after that. The object manipulation scenario has a longer convergence time on average, taking 300 generations to evolve a full solution, and up to 40 more generations to evolve a more time efficient solution.

The object destruction and object collection scenarios each had similar average convergence times, with object destruction being the slightly easier of the two scenarios. From a practical standpoint, the object destruction scenario is the easiest of the three scenarios because the sequential subtasks are assigned to the swarm, not an individual agent. Specifically unlike the object collection scenario where a single agent is responsible for both picking up and depositing an object, an agent in the object destruction scenario is only responsible for one "attack" on an object, relying on another member of the swarm to complete the job. Thus, evolution of object destruction behaviors is easier because the sequential dependencies on a single agent are eliminated.

Separately looking at the average convergence times for the collection and destruction tasks in the object manipulation scenario (**Figure 6.8** and **Figure 6.10**), the same convergence rate is not observed as when the tasks are evolved separately (**Figure 6.4** and **Figure 6.6**). This performance decrease is expected, as the collection and destruction tasks are in direct competition for the same pool of fitness resources. So, the evolution of both behaviors simultaneously increases the competition faced by each solution at every generation. In the early implementations of the system, this competition actually inhibited convergence because the the fitness function used a more traditional approach of weighing individual metrics, which led to the development of "expert" solutions [16]. In this case, an expert solution is one that is able to completely satisfy one goal but not the other, such as collecting all type C objects but not destroying any type D objects. These expert solutions dominated the population, thus once the population was sufficiently homogeneous, the system focused on using mutations to transform an expert solution into a solution that addressed both goals. The homogenization identified a need for a method to balance the evolution between multiple objectives, hence the introduction of the lexicographical scoring method. As seen in **Figure 6.8** and **Figure 6.9**, the lexicographic scoring is able to balance the evolution towards both collection and destruction. The simultaneous progress is made on each secondary subgoal, m_6 and m_8, and each primary subgoal, m_5 and m_7, thus enabling the simultaneous evolution of a complete solution for the object manipulation scenario.

Object Destruction

The average number of generations required to evolve a solution for object destruction is approximately 50 generations. As seen in **Figure 6.4**, within 2 generation a solution is found that completely minimizes d_2, and within 5 generations a majority of the population is able to sufficiently minimize d_2, as shown by **Figure 6.5**. As the population drives towards minimizing d_2, slow progress is eventually made towards the minimization of d_1. Mutated solutions that begin to minimize d_1 rise to the top of the fitness scale, thus being selected more often for reproduction. Eventually in generation 14, a solution is produced that completely minimized d_1 and d_2. As the population converges towards a set of complete solutions, the last metric to be minimized is the time t. It is interesting to note how the mean error for the population increases once a complete solution is found. This behavior can be attributed to the mutations being applied during reproduction since at this point most of the solutions in the population minimize d_1 and d_2, thus the probability that a random mutation will be helpful is small.

A solution that completely minimizes both d_1 and d_2 is found in generation 14, but approximately 5 more generations of evolution is able to produce a solution that also minimizes the time needed to destroy all the objects. The solution evolved here, shown in **Table 6.7**, is able to completely destroy all 30 objects in approximately 30 timesteps. **Table 6.8** is a simplified version of the evolved object destruction algorithm shown in **Table 6.7**. The simplifications include the removal of unreachable states and non-firing transitions, and the marking of actions that will have no effect.

Every agent executing the algorithm in **Table 6.8** starts in state A. If the agent is on any type D object, the appropriate attack will be executed and the agent will be disabled for the duration of the simulation. If the agent is not on a type D object, the agent transitions to state E, where they will randomly wander until they are within sensing range of a type D object. Though unlikely here, if the agent stumbles upon a partially destroyed type D object, they will attack it appropriately and become disabled. What is more likely to happen that the agent will come in proximity to a type D object, and since they are not on a partially destroyed type D object, they will move towards the nearest type D object and transition from state E to C. Implicit in the transitions from state E to state C is that the agent is near a type D object, thus in state C the first transition fires, moving an agent towards a type D object and transitioning back to state A, where this cycle repeats until all the type D objects are destroyed.

The transition from state E to state C has the implied condition that the agent is near a type D object, say X, but since there are multiple agents executing this same algorithm, a race condition occurs. It is conceivable that in the transition from E to C another agent completes the destruction X, which will probably cause the agent that transitioned from E to C to execute the second transition in state C, sending the agent to state J. In state J, the agent will randomly wander until it is sitting on top of an untouched

type D object, say Y, at which time it will wander off Y and transition to state F. In the unlikely event that the agent wanders onto a partially destroyed type D object, the second transition in state F will fire, completing the destruction of a type D object and disabling the agent. More often though, the agent will execute the first transition in state F, bringing the agent back on top of Y and causing a transition to state H, where the agent will enter a do-nothing infinite loop between states H and D. Only once another agent attacks Y will the agent be able to break from the infinite loop. In the end though, even if the agent is broken free from the H/D infinite loop, the agent is relegated to transitioning between states B, D, H, and I which do not provide much help in accomplishing the task of object destruction.

Table 6.7: Evolved object destruction state machine

State	Condition	Action	Next State
A	on-object_D(untouched)	first-attack	A
	!on-object_D(damaged)	move-towards-object_D	E
	near-object_D	second-attack	F
B	!near-object_D	broadcast-object_D	H
	!on-object_D(untouched)	move-towards-object_D	B
	near-object_D	second-attack	I
C	near-object_D	move-towards-object_D	A
	on-object_D(untouched)	first-attack	H
	!on-object_D(damaged)	move-random	J
	on-object_D(damaged)	move-random	A
D	!on-object_D(untouched)	move-random	B
	!on-object_D(damaged)	second-attack	H
	!near-object_D	move-towards-object_D	I
E	!near-object_D	move-random	E
	on-object_D(damaged)	second-attack	C
	on-object_D(untouched)	move-random	E
	!on-object_D(damaged)	move-towards-object_D	C
F	!on-object_D(damaged)	move-towards-object_D	H
	!on-object_D(untouched)	second-attack	I
H	!on-object_D(untouched)	broadcast-object_D	H
	on-object_D(untouched)	second-attack	D
I	!near-object_D	second-attack	H
	near-object_D	second-attack	B
J	on-object_D(untouched)	move-random	F

Table 6.8: Simplified version of the evolved object destruction state machine

State	Condition	Action	Next State
A	on-object_D(untouched)	first-attack	-
	!on-object_D(damaged)	move-to-object_D	E
	near-object_D	second-attack	-
B	!near-object_D	nop	H
	!on-object_D(untouched)	move-to-object_D	B
	near-object_D	second-attack	I
C	near-object_D	move-to-object_D	A
	!on-object_D(damaged)	move-random	J
D	!on-object_D(untouched)	move-random	B
	!on-object_D(damaged)	nop	H
	!near-object_D	nop	I
E	!near-object_D	move-random	E
	on-object_D(damaged)	second-attack	-
	on-object_D(untouched)	move-random	E
	!on-object_D(damaged)	move-to-object_D	C
F	!on-object_D(damaged)	move-to-object_D	H
	!on-object_D(untouched)	second-attack	-
G	on-object_D(damaged)	second-attack	I
H	!on-object_D(untouched)	broadcast-object_D	H
	on-object_D(untouched)	nop	D
I	!near-object_D	nop	H
	near-object_D	second-attack	-
J	on-object_D(untouched)	move-random	F

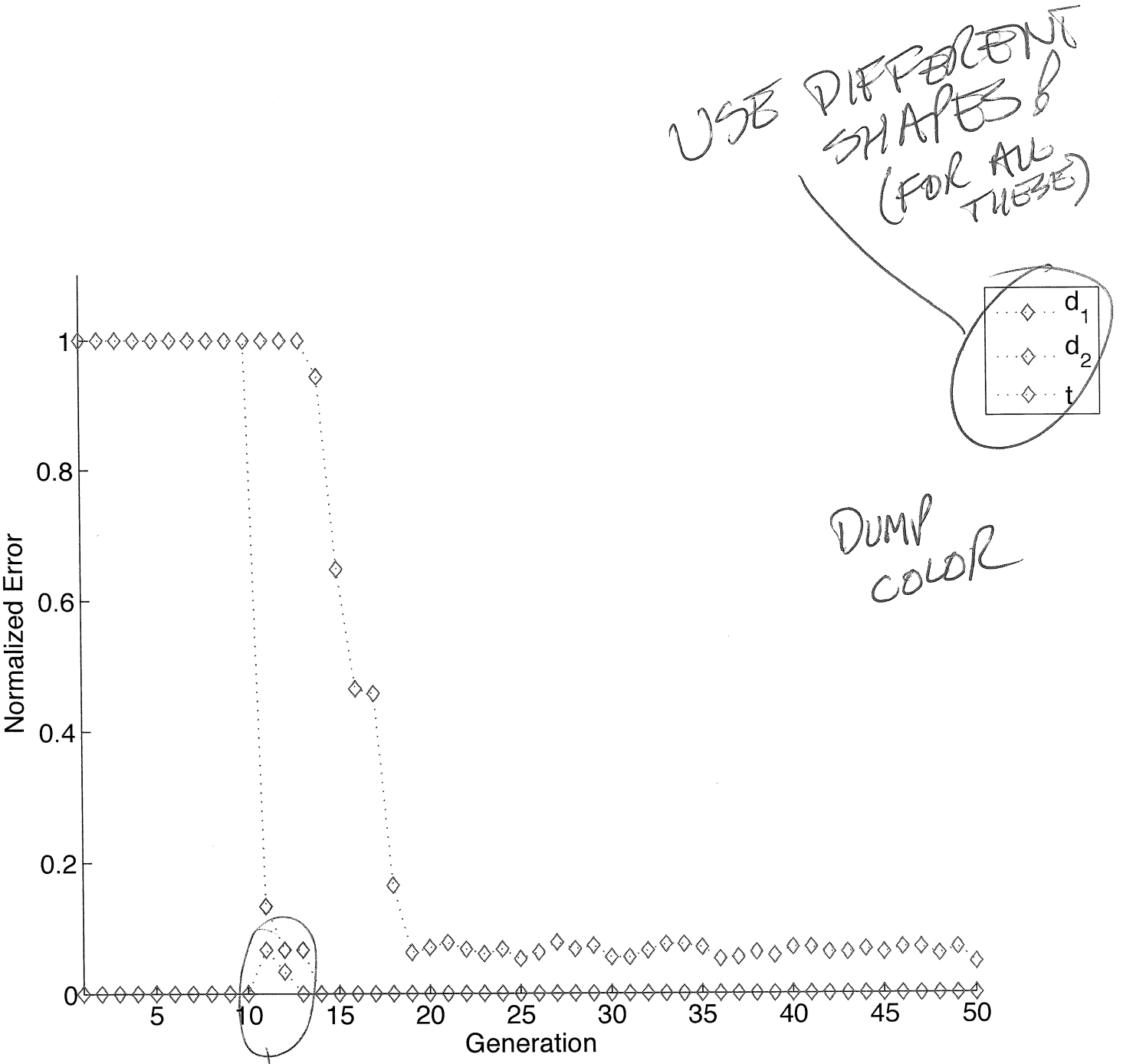


Figure 6.4: Progress of the best solution over time for object destruction is shown. In generation 14, a solution that minimizes to 0 the error measures d_1 and d_2 is found. Around generation 19, a solution that minimizes the amount of time to complete object destruction is found.

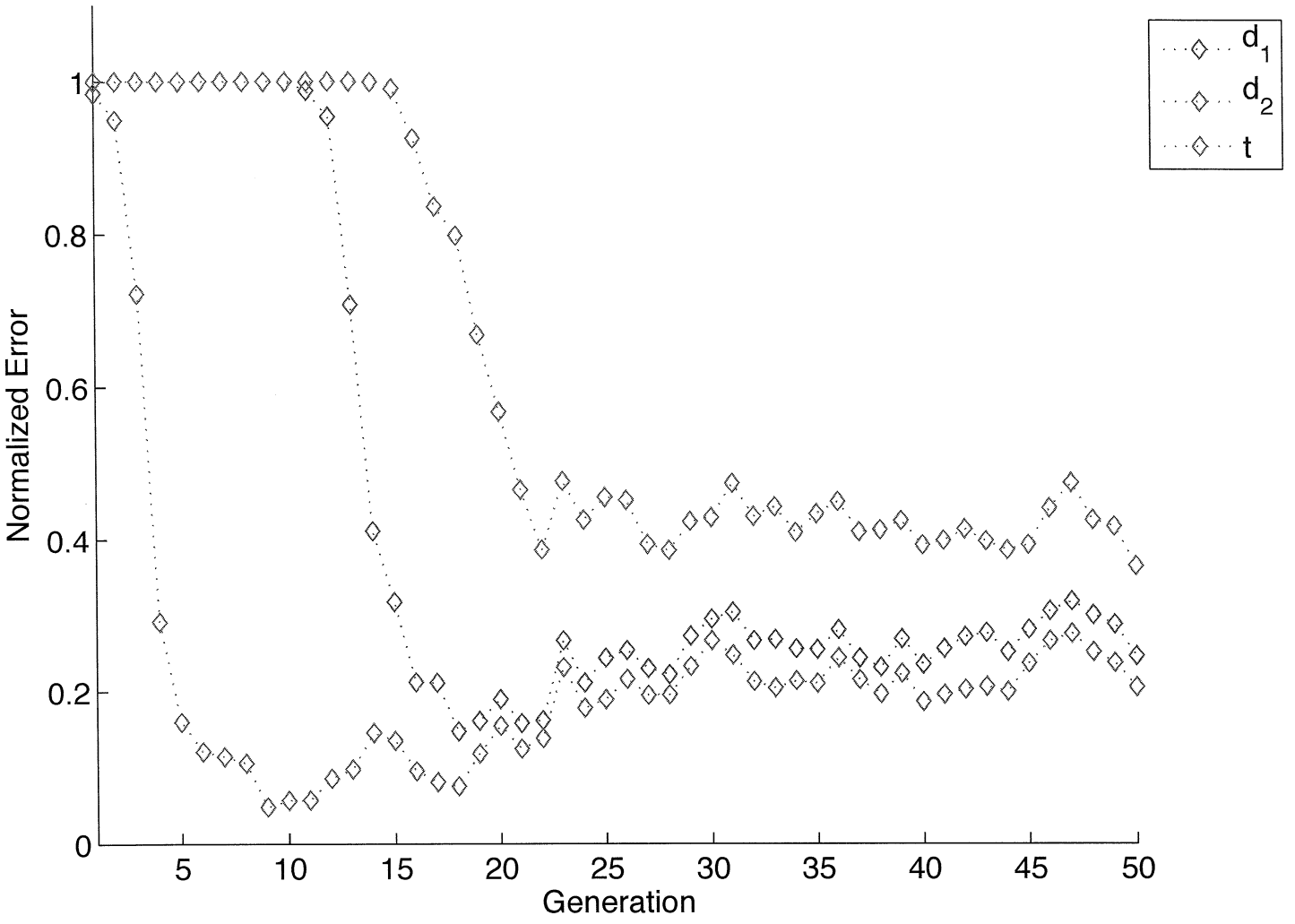


Figure 6.5: Mean error over time for object destruction is shown. Notice how the mean error begins to increase as soon a the best solution is found at generation 19. This is attributed to the mutations being applied during reproduction since at this point most of the solutions in the population minimize d_1 and d_2, thus the probability that a random mutation will be helpful is small.

Object Collection

The average number of generations required to evolve a solution for object collection is approximately 55 generations. As seen in **Figure 6.6**, within 5 generation a solution is found that completely minimizes c_2, and within 10 generations a majority of the population is able to sufficiently minimize c_2, as shown by **Figure 6.7**. As the population drives towards minimizing c_2, slow progress is eventually made towards the minimization of c_1. Mutated solutions that begin to minimize c_1 rise to the top of the fitness scale, thus being selected more often for reproduction. Eventually in generation 21 a solution is produced that completely minimized c_1 and c_2. As the population converges towards a set of complete solutions, the last metric to be minimized is the time t. The performance of the best solution with respect to t up until generation 45 is quite varied, indicating that the solutions are still relying on some form of serendipity to complete the task. As seen in **Figure 6.6**, after generation 45, four marked periods of consistent performance occur. During these periods, a single solution dominates the population, thus this one solution receives a majority of the mutations. Hence at this point, the optimization of t has essentially become a pure random search, using the mutation operators to explore the solution space around the best solution. The additional work required to optimize t is due to the fact that agents do not receive credit for navigating to the object repository unless they arrive there and successfully deposit the object they are holding. Additionally, similar to what is seen in evolving object destruction, the mean error for the population increases once a complete solution is found.

The solution evolved for object collection is shown in **Table 6.9**, and a simplified version of the algorithm is shown in **Table 6.10**. The simplifications include the removal of unreachable states and non-firing transitions, and the marking of actions that will have no effect.

When executing the solution in **Table 6.10**, each agent starts in state A. No agent is holding any type C objects at the beginning, so the second transitions in state A cannot fire. Thus the first time through state A, the agent eventually will select the first transition, attempt to pick up and object, and goto state E. When transitioning from state A, the first two transitions in state E cannot fire, but either the third or fourth will definitely be activated. These two branches will be discussed separately.

First, if the agent is not near any type C object, the third transition of state E will fire, resulting in going to state B. As a consequence of the transition to state B from E, the first two transitions in state B cannot fire. If the agent is not holding a type C object, the agent will transition to state C and eventually end up back in state A. If the agent is holding a type C object, then the third transition in state B fires, navigating the agent towards the goal area and transitioning into state E. At this point, if the agent is in the goal area (which is unlikely), the agent will transition from state E to state G, an inescapable loop that will cause the agent to hold onto the type C object forever. Now, in order for the agent to continue navigating towards the goal area, there cannot be any near by objects preventing the firing of the third transition in state E. If

there is an object near by, the agent will navigate towards it, broadcast its location, and wait to deliver its object until the other object is collected. After cycling like this through states B and E until the agent is in state B and located in the goal area, the first transition in state B fires, and assuming that there are not type C objects around, the agent will not move and will transition to state A, resulting in the completion of collecting one object. Unfortunately, from here the agent transitions from $A \rightarrow F \rightarrow E \rightarrow G$. Thus, after successfully collecting an object, the agent becomes useless as it is stuck in the infinite loop at G.

The other branch point initially reached at state E happens when the fourth transition fires because an agent is near an object. The agent will cycle through states $E \rightarrow C \rightarrow A \rightarrow E$, navigating towards the object until it is picked up in state A. At this point, the agent is holding an object and is in state E, which is the same situation previously discussed.

The evolved object collection algorithm does exhibit a form of emergent behavior. Given the algorithm in **Table 6.10** and a swarm of agents, the object collection task is performed quickly and completely. But, when this algorithm is executed with only one or a few agents, the swarm fails to complete the object collection. What happens is that an agent picks up an object in state A, transitions to state E, and if the agent is near any other objects, they navigate towards that object and broadcast its position until another agent picks up the object. But in this case with one or few agents, there is no one able to collect the object and break the loop. Thus, the agent(s) are locked in a broadcasting loop and never complete the object collection task.

Table 6.9: Evolved object collection algorithm

State	Condition	Action	Next State
A	!on-goal	pick-up	E
	holding-object_C	put-down	F
	on-object_C	move-to-goal	G
B	on-goal	move-to-object_C	A
	near-object_C	broadcast-garbage	D
	on-object_C	move-random	G
	holding-object_C	move-to-goal	E
	!on-object_C	move-random	C
C	!on-goal	move-to-object_C	A
	on-object_C	pick-up	E
	near-object_C	put-down	B
D	!on-goal	put-down	F
	!near-object_C	put-down	B
	on-object_C	move-random	G
E	on-goal	move-random	G
	on-object_C	put-down	B
	!near-object_C	move-to-goal	B
	!on-object_C	move-to-object_C	C
F	!holding-object_C	put-down	E
	holding-object_C	move-to-object_C	B
G	!holding-object_C	move-to-object_C	G

Table 6.10: Simplified version of the evolved object collection algorithm

State	Condition	Action	Next State
A	!on-goal	pick-up	E
	holding-object_C	put-down	F
B	on-goal	move-to-object_C	A
	near-object_C	broadcast-garbage	D
	holding-object_C	move-to-goal	E
	!on-object_C	move-random	C
C	!on-goal	move-to-object_C	A
	near-object_C	put-down	B
D	!on-goal	nop	F
	!near-object_C	put-down	B
E	on-goal	move-random	G
	on-object_C	nop	B
	!near-object_C	move-to-goal	B
	!on-object_C	move-to-object_C	C
F	!holding-object_C	nop	E
	holding-object_C	move-to-object_C	B
G	!holding-object_C	move-to-object_C	G

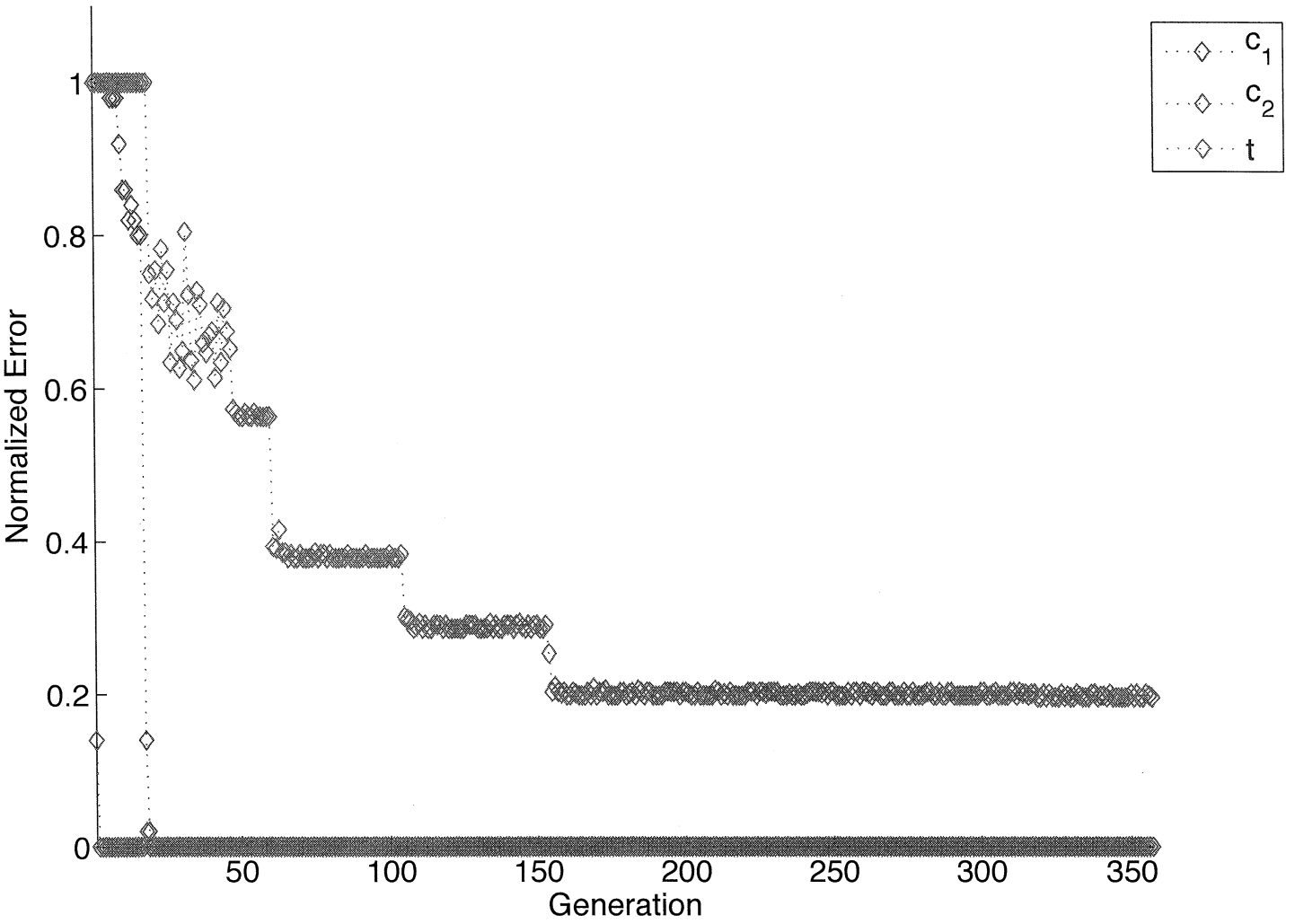


Figure 6.6: Progress of the best solution over time for object collection is shown. In generation 21, a solution that minimizes to 0 the error measures c_1 and c_2 is found. Around generation 150, a solution that minimizes the amount of time to complete object collection is found.

Object Manipulation

The final scenario presented to the system is the complete object manipulation scenario involving the simultaneous collection and destruction of all the objects in the environment. The average behavior of the evolution of an object manipulation solution is much less well defined than the behavior of either of the other scenarios. As shown by **Figure 6.8** and **Figure 6.10**, two different runs of object manipulation can have very different behavior with respect to convergence. In **Figure 6.8**, almost 325 generations are required to evolve a solution that completely collects and destroys all the objects in the environment, but only around 160 generations for the run shown in **Figure 6.10**. Additionally, the run in **Figure 6.8** never brought the the run-time of the best solution below 80% of the allowed runtime, whereas the run in **Figure 6.10** is able to find a solution in on average 50

On average, the evolution of a solution for object manipulation requires approximately 300 generations to evolve a complete solution, and at least 100 additional generations is required to begin time optimizing a solution. The convergence time for the combined object collection and destruction scenarios is much longer than that of the individual scenarios. This performance is expected as the multiple independent objectives compete with each other for survival.

The balancing effect of the lexicographical sorting can be seen in both **Figure 6.9** and **Figure 6.11**, preventing one metric from dominating and skewing the mean performance of the population towards that metric. This point is clearly illustrated through the progress of the mean population performance shown in **Figure 6.9**. Metrics m_6 and m_8, representing the secondary subtasks for object collection or destruction, and metrics m_5 and m_7, representing the completion of the object collection or destruction task, do not deviate far from each other. Also notice the effect of the lexicographic scoring and how neither of the two main tasks (collection or destruction) take a commanding lead over the other task. Additionally, unlike the object collection and destruction evolutions, due to the larger search space the mean fitness does not begin to increase after a good solution is found.

The performance of the lexicographic scoring is observed in **Figure 6.14**; note how the number of solutions that satisfy one of the four metrics m_1, m_2, m_3, or m_4 increases with respect to the progress made by the best solution. This same performance can also be observed through **Figure 6.12** and **Figure 6.13**, which depict the complete progress of the population over time for metrics m_5 and m_7 respectively. **Figure 6.12** shows the fitness of every solution, indexed by generation, with repect to metric m_5. So, for example, in generation 300, almost half the population has mimized m_5 to 0, a quarter of the population is very close to 0, and about 10% of the population are not minimizing m_5 at all. Similarly to **Figure 6.12**, **Figure 6.13** represents the fitness of all generations with respect to m_7.

Unlike the previous two scenarios, where once a complete solution is discovered that solution is further

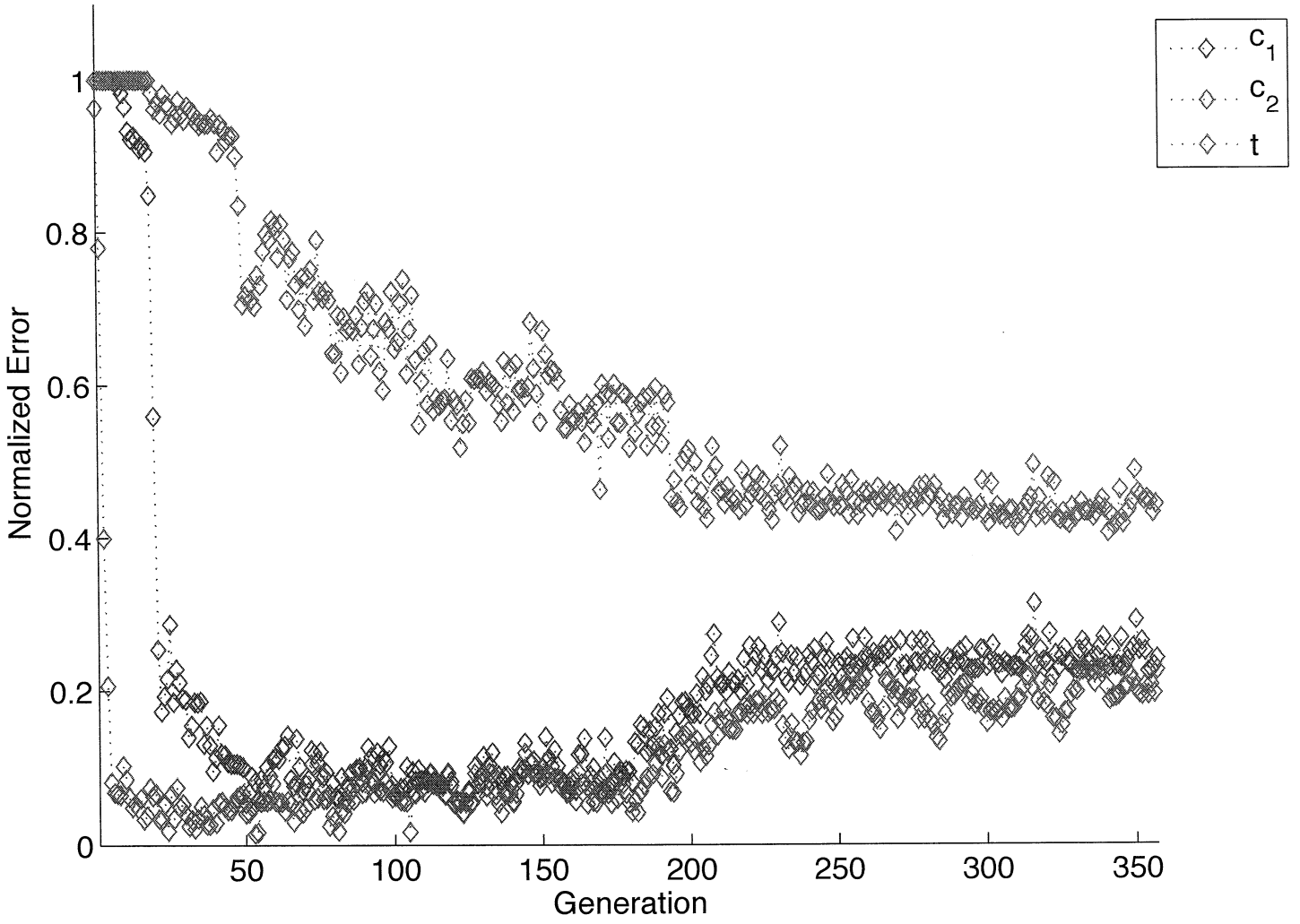


Figure 6.7: Mean error over time for object collection is shown. Notice how the mean error begins to increase as soon a the best solution is found at generation 150. This is attributed to the mutations being applied during reproduction since at this point most of the solutions in the population minimize c_1 and c_2, thus the probability that a random mutation will be helpful is small.

evolved for faster performance, no noticeable performance increases are guaranteed in this scenario once a complete solution is found, as shown in the difference between **Figure 6.8** and **Figure 6.10**. Overall, about 50% of the time a complete solution is evolved that is able to perform quickly. The inability of the system to consistently evolve a fast solution is attributed to increased number of actions and sensor options in this scenario, thus increasing the complexity of the evolved solution as compared to the previous evolved solutions, and exponentially increasing the number of candidate solutions.

The faster of the two solutions evolved for object collection is shown in **Table 6.11**, and a simplified version of the algorithm is shown in **Table 6.12**. The simplifications include the removal of unreachable states and non-firing transitions, and the marking of actions that will have no effect.

The solution evolved here is much more complex than either of the solutions evolved for object collection or object destruction. For the most part, the main functionality of the solution is spread across multiple states. Due to the complexity of the solution and distribution of functionality across many states, it is difficult to analyze the algorithm completely and extract core behaviors. With this in mind, a brief analysis of the salient portions of the algorithm will be discussed.

If an agent is in state A and is on a type C object, the agent will pick up the object and transition to state J. Given a typical run of events, the agent will navigate to the goal area and deposit the object, cycling through states $J \rightarrow G \rightarrow K \rightarrow J$.

If an agent is not on a type C object, they will loop through state A until they are near some type of object. Upon sensing a type C object, the agent transitions to state J. From here, most likely they will fire the last transition in state J, which leads to state I, and begin to navigate towards the nearest type C object. From state I the agent will most likely transition to state C where it will loop until one of the conditions on the transitions is met. At this point, the serendipity of the agent's random walk is relied upon to guide the agent towards the type C object. If the agent does in fact get to the object, the agent will broadcast the location of the object, transition to state J and then to state A where it will pick up the object. From this point, the agent will deliver the object to the goal area as previously discussed.

In the other case, if the agent starting in state A randomly wanders until it is in the proximity of a type D object, the agent transitions to state H, broadcasts the location of the object, and then transitions to state G. From state G, it is difficult to reason about what transition an agent will follow because there are many possible situations. In examining simulation runs with only a single agent, the basic result is that the agent will essentially rely on serendipity to guide the final steps towards the object, then most likely the object will be acted on in state F reached through state I.

Table 6.11: Evolved object manipulation algorithm

State	Condition	Action	Next State
A	on-object_C	pick-up	J
	near-object_C	move-to-object_D	J
	near-object_D	move-random	H
	holding-object_C	move-to-object_C	F
	on-goal	pick-up	G
B	!on-goal	move-random	J
	!on-object_D(untouched)	move-to-goal	K
C	!near-object_C	broadcast-object_D	A
	on-object_D(untouched)	broadcast-object_D	A
	holding-object_C	broadcast-object_D	D
	on-object_C	broadcast-garbage	J
D	near-object_D	broadcast-object_D	C
	!on-object_C	move-random	K
	holding-object_C	broadcast-garbage	D
	!holding-object_C	move-to-object_D	B
E	!near-object_D	pick-up	K
F	on-object_C	move-to-object_C	B
	holding-object_C	put-down	K
	on-object_D(damaged)	second-attack	B
	!near-object_D	first-attack	B
	on-object_D(untouched)	first-attack	I
	near-object_C	put-down	K
	near-object_D	move-to-object_D	D
G	on-object_D(damaged)	broadcast-object_D	C
	holding-object_C	move-to-goal	K
	!near-object_D	move-to-object_C	B
	near-object_C	broadcast-object_D	A
	!on-goal	move-to-goal	I
	near-object_D	move-to-object_C	J
H	!on-object_C	broadcast-object_D	G
	!on-object_D(untouched)	move-to-goal	H
I	on-goal	pick-up	E
	on-object_D(untouched)	move-to-object_D	F
	!near-object_C	broadcast-object_D	J
	on-object_C	put-down	A
	!on-object_D(damaged)	move-to-object_C	C
	near-object_C	second-attack	B
J	on-object_D(damaged)	pick-up	I
	holding-object_C	move-to-goal	G
	near-object_D	pick-up	I
	on-object_C	broadcast-garbage	E
	!near-object_C	put-down	C
	!on-goal	move-to-object_C	I
K	!on-object_D(damaged)	put-down	J
	on-object_D(damaged)	put-down	G

Table 6.12: Simplified version of the evolved object manipulation algorithm

State	Condition	Action	Next State
A	on-object_C	pick-up	J
	near-object_C	move-to-object_D	J
	near-object_D	move-random	H
	holding-object_C	move-to-object_C	F
	on-goal	nop	G
B	!on-goal	move-random	J
C	!near-object_C	broadcast-object_D	A
	on-object_D(untouched)	broadcast-object_D	A
	holding-object_C	broadcast-object_D	D
	on-object_C	broadcast-garbage	J
D	near-object_D	broadcast-object_D	C
	!on-object_C	move-random	K
	holding-object_C	broadcast-garbage	D
	!holding-object_C	nop	B
E	!near-object_D	pick-up	K
F	on-object_C	nop	B
	holding-object_C	put-down	K
	on-object_D(damaged)	second-attack	-
	!near-object_D	nop	B
	on-object_D(untouched)	first-attack	-
	near-object_C	put-down	K
	near-object_D	move-to-object_D	D
G	on-object_D(damaged)	broadcast-object_D	C
	holding-object_C	move-to-goal	K
	!near-object_D	move-to-object_C	B
	near-object_C	broadcast-object_D	A
	!on-goal	move-to-goal	I
	near-object_D	move-to-object_C	J
H	!on-object_C	broadcast-object_D	G
I	on-goal	nop	E
	on-object_D(untouched)	nop	F
	!near-object_C	broadcast-object_D	J
	on-object_C	nop	A
	!on-object_D(damaged)	move-to-object_C	C
	near-object_C	second-attack	-
J	on-object_D(damaged)	nop	I
	holding-object_C	move-to-goal	G
	near-object_D	pick-up	I
	on-object_C	broadcast-garbage	E
	!near-object_C	put-down	C
	!on-goal	move-to-object_C	I
K	!on-object_D(damaged)	put-down	J
	on-object_D(damaged)	nop	G

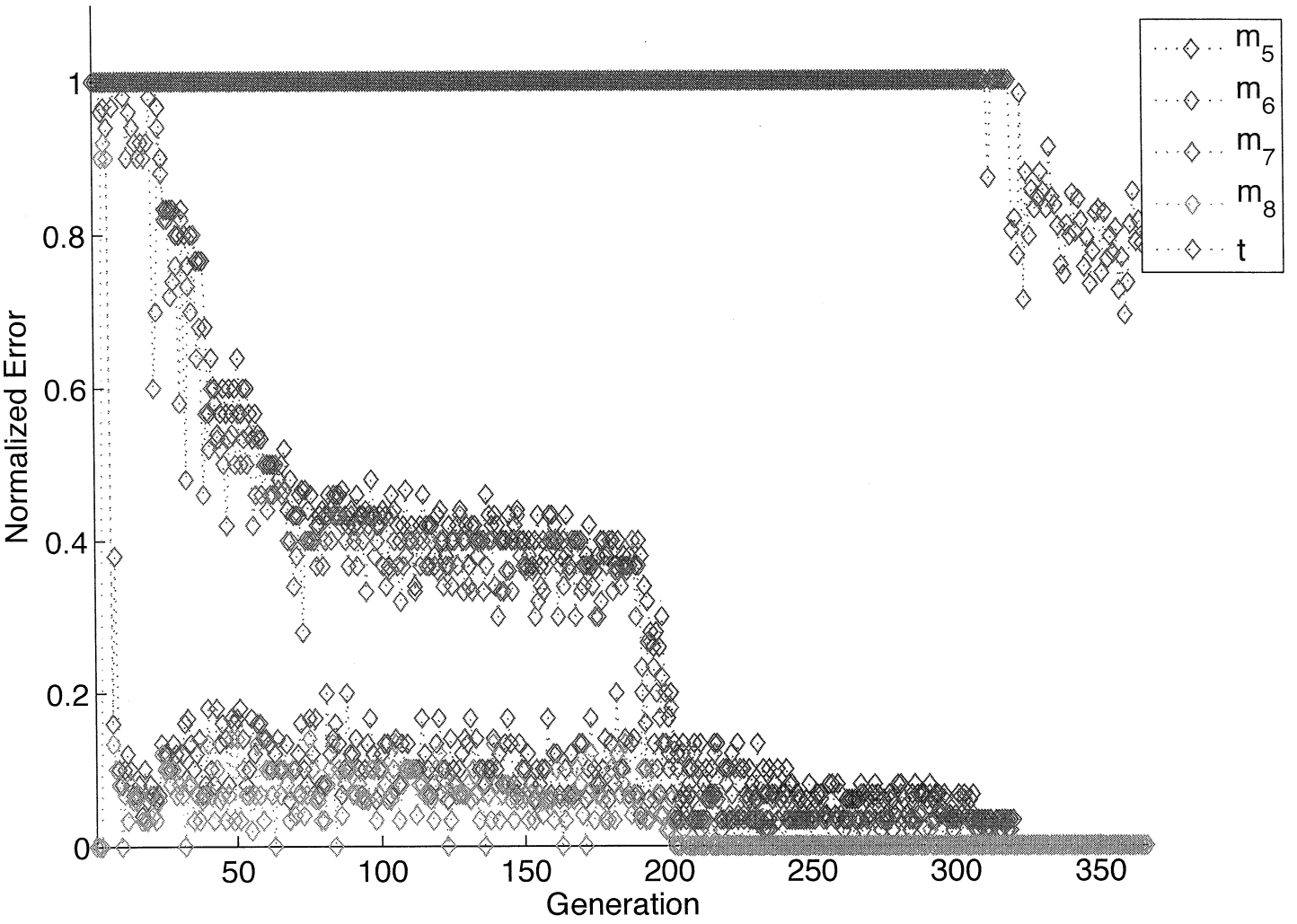


Figure 6.8: Progress of the best solution over time for object manipulation is shown. In generation 325, a solution that minimizes to 0 the error measures m_5-m_8 is found. After this, some progress is made on optimizing the speed of the algorithm, but no real progress is made (hence truncation after generation 370).

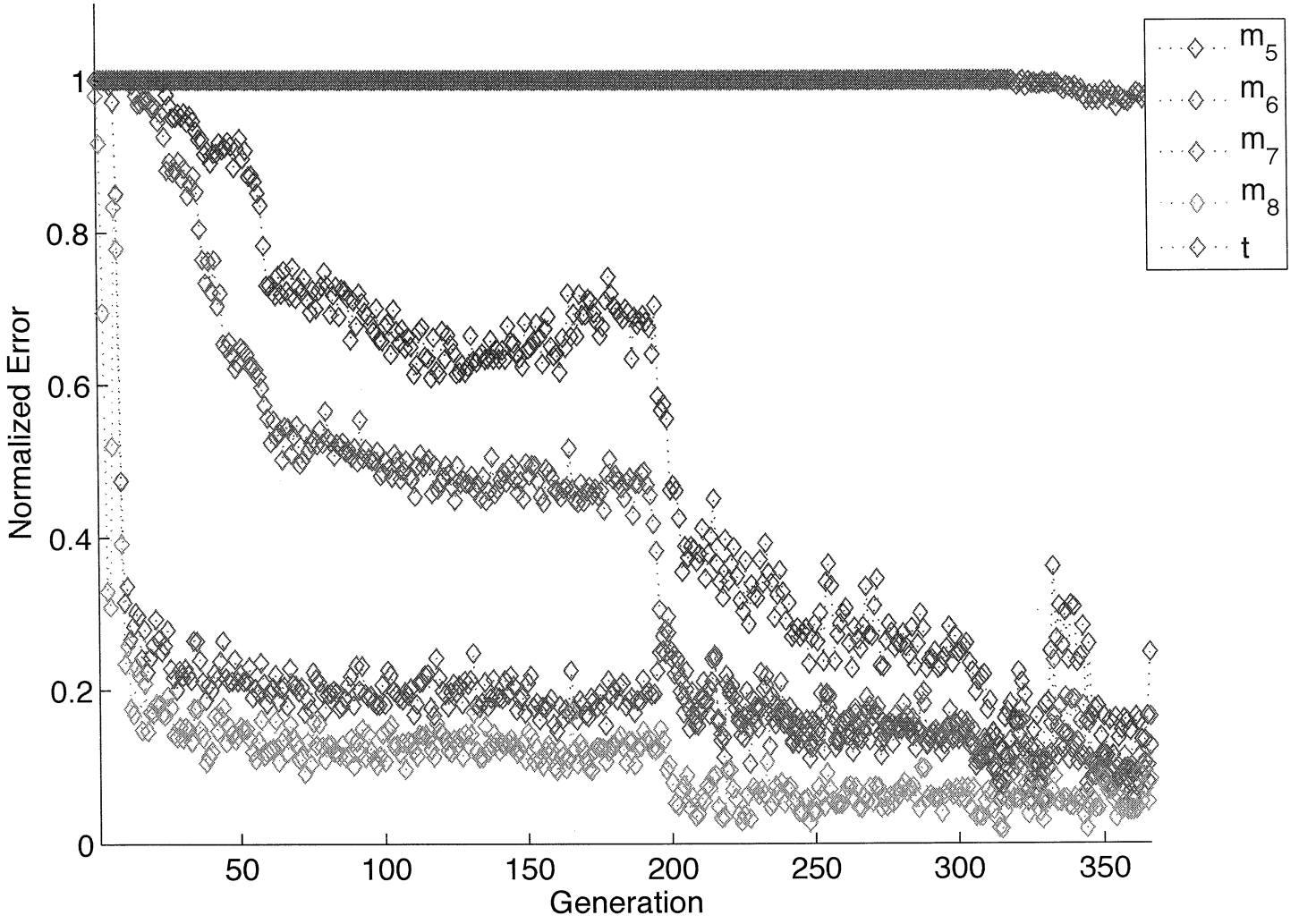


Figure 6.9: Mean error over time for object manipulation is shown. Notice the effect of the lexicographic scoring and how neither of the two main tasks (collection or destruction) take a commanding lead over the other task. Also, unlike the object collection and destruction evolutions, due to the larger search space the mean fitness does not begin to increase after a good solution is found.

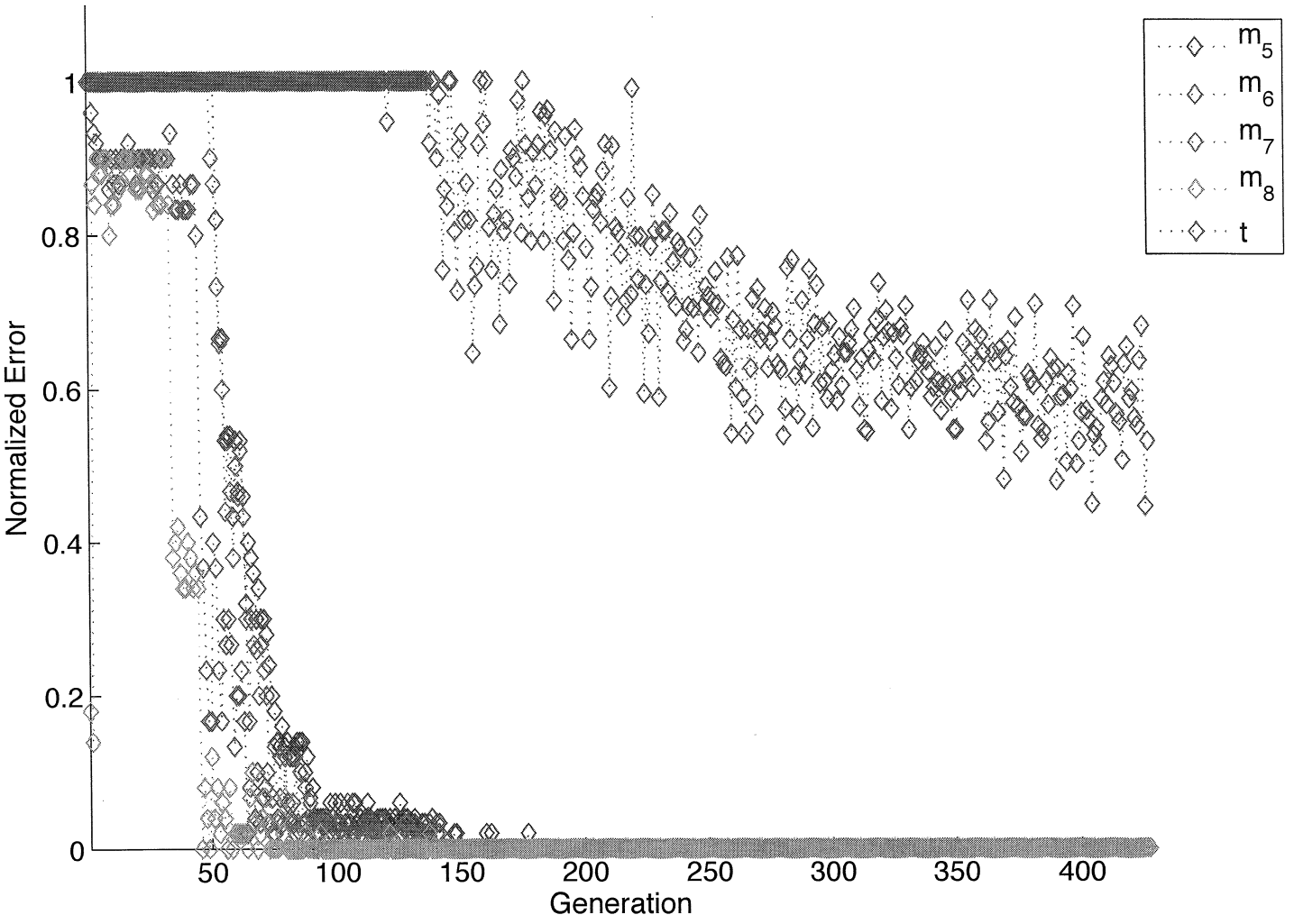


Figure 6.10: Progress of the best solution over time for another object manipulation evolution is shown. In generation 160, a solution that minimizes to 0 the error measures m_5-m_8 is found. Around generation 160, a modest amount of progress in time optimization is achieved, resulting in a solution that is faster than the one evolved in **Figure 6.8**.

Best of 100 sims.

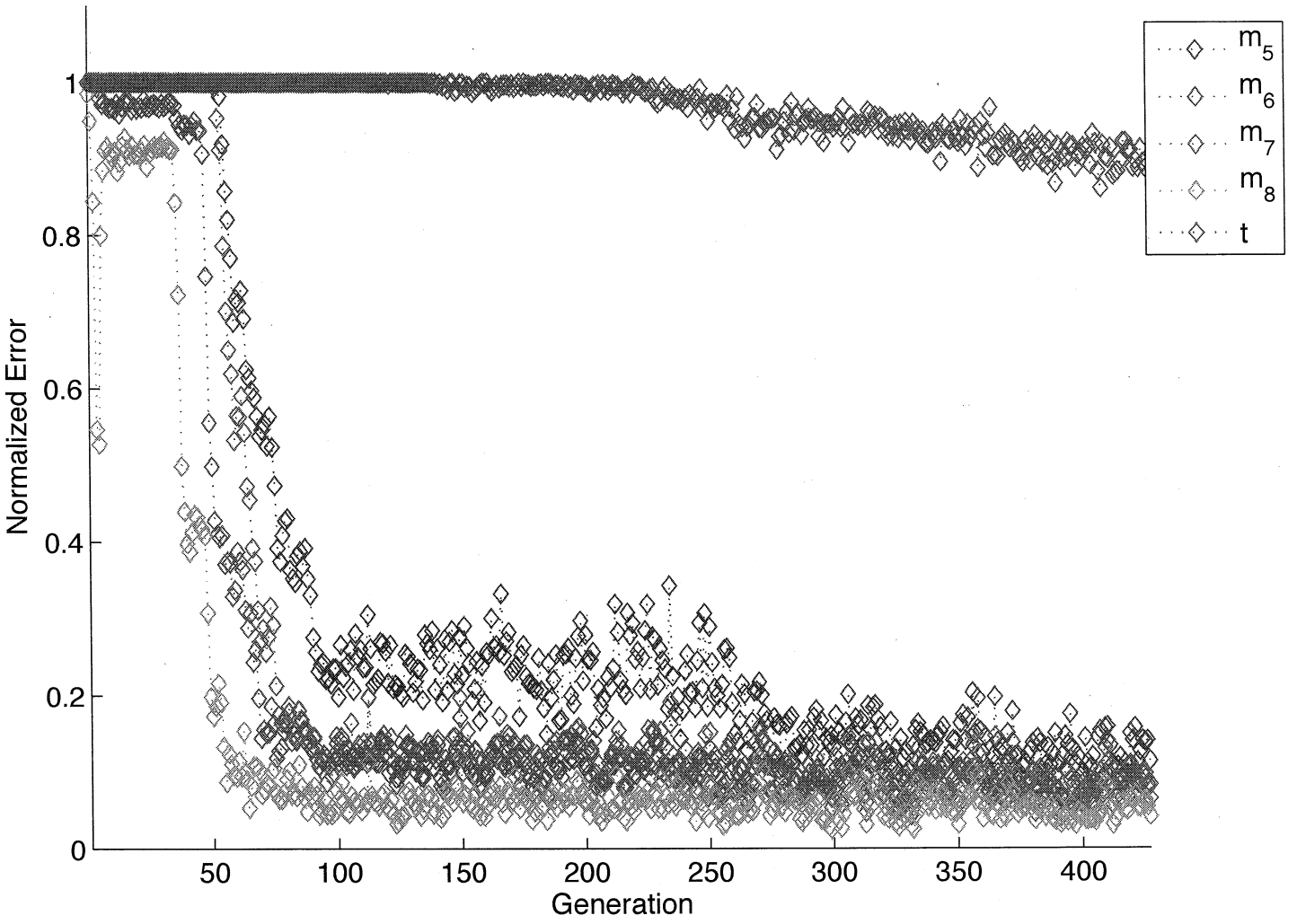
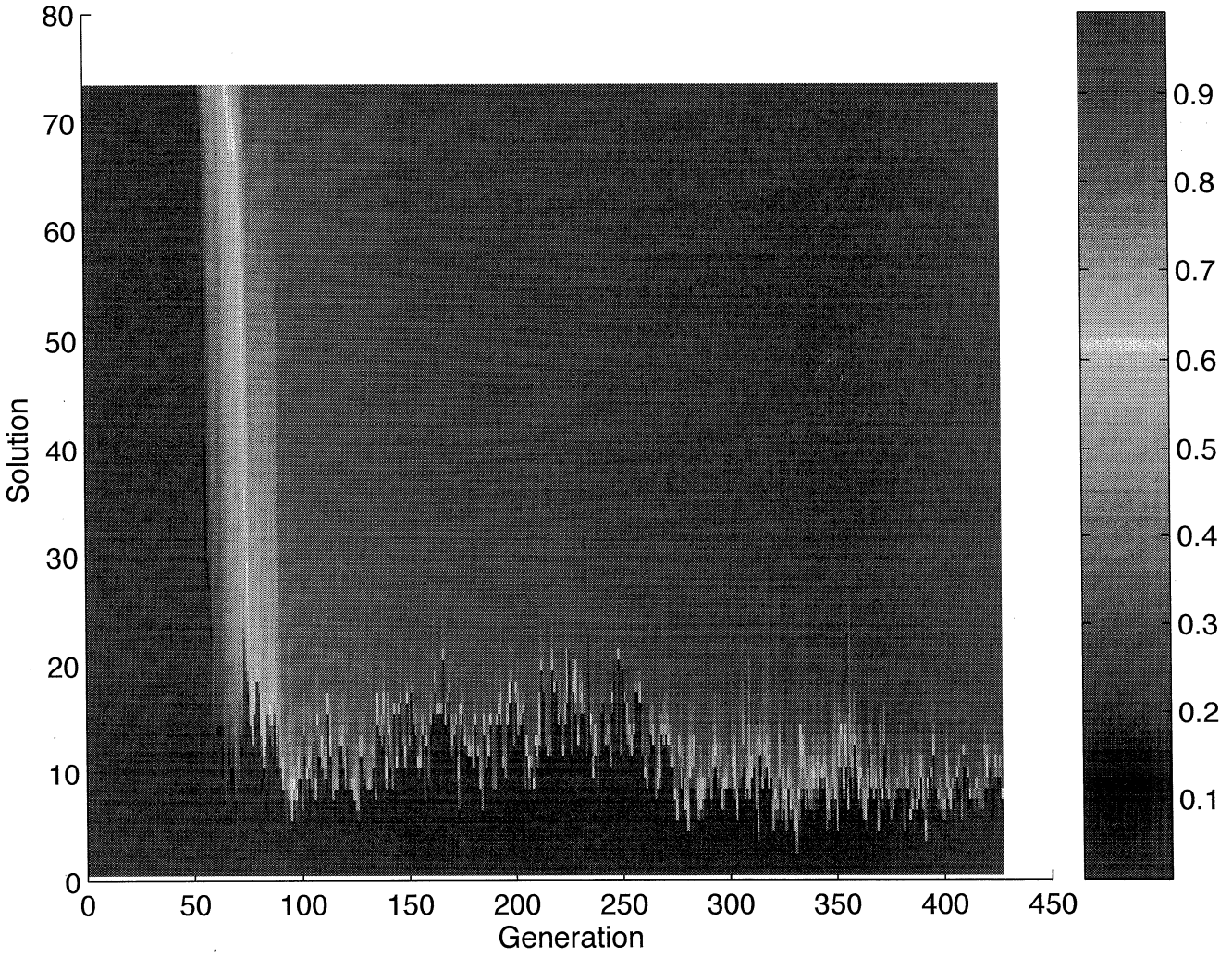
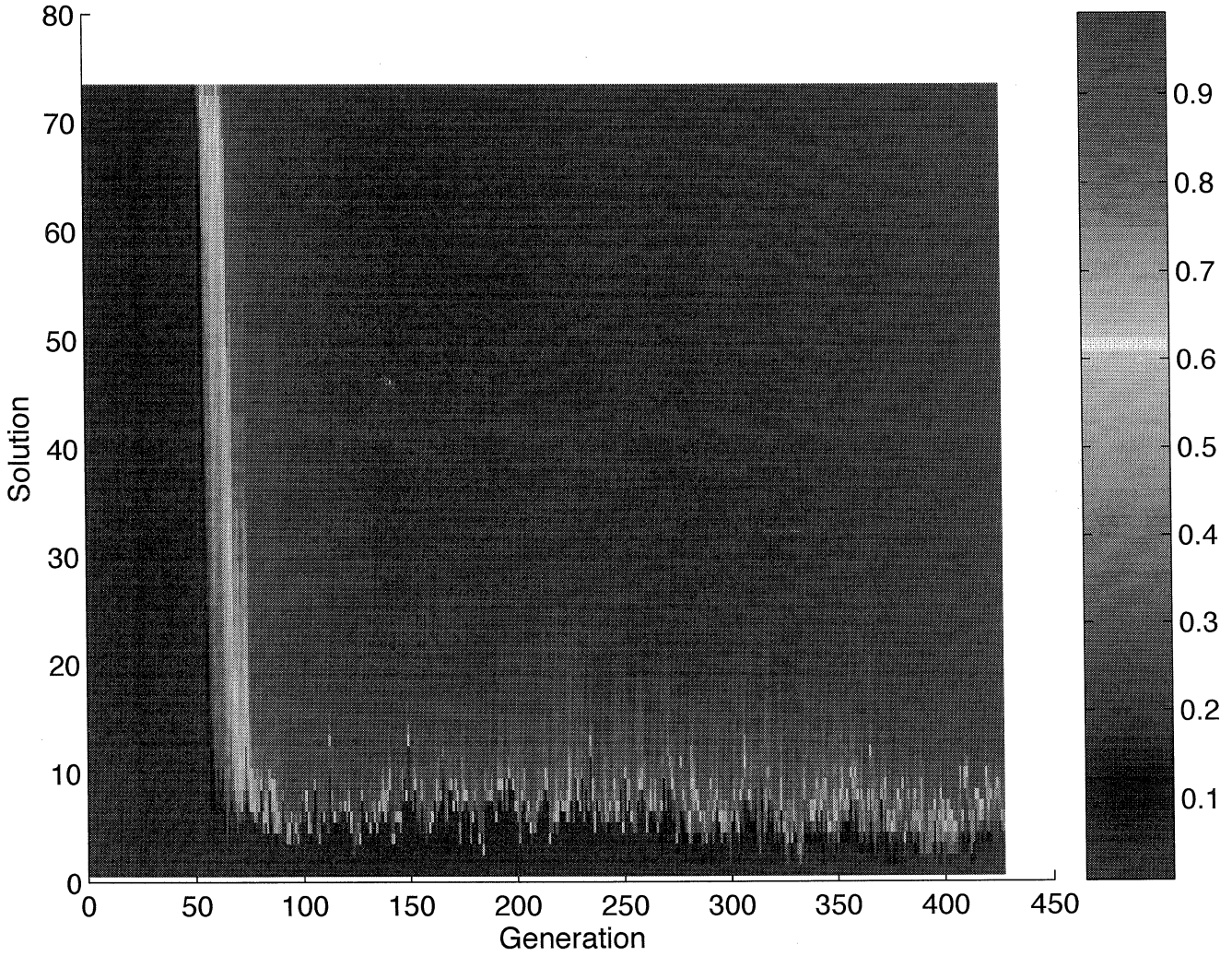


Figure 6.11: Mean error over time for the second object manipulation evolution is shown. Again, notice the effect of the lexicographic scoring and how neither of the two main tasks (collection or destruction) take a commanding lead over the other task. Also, unlike the object collection and destruction evolutions, due to the larger search space the mean fitness does not begin to increase after a good solution is found.



The bar on the right needs to be reversed

Figure 6.12: The fitness of every solution, indexed by generation, with repect to metric m_5.



The bar on the right needs to be reversed

Figure 6.13: The fitness of every solution, indexed by generation, with repect to metric m_7.

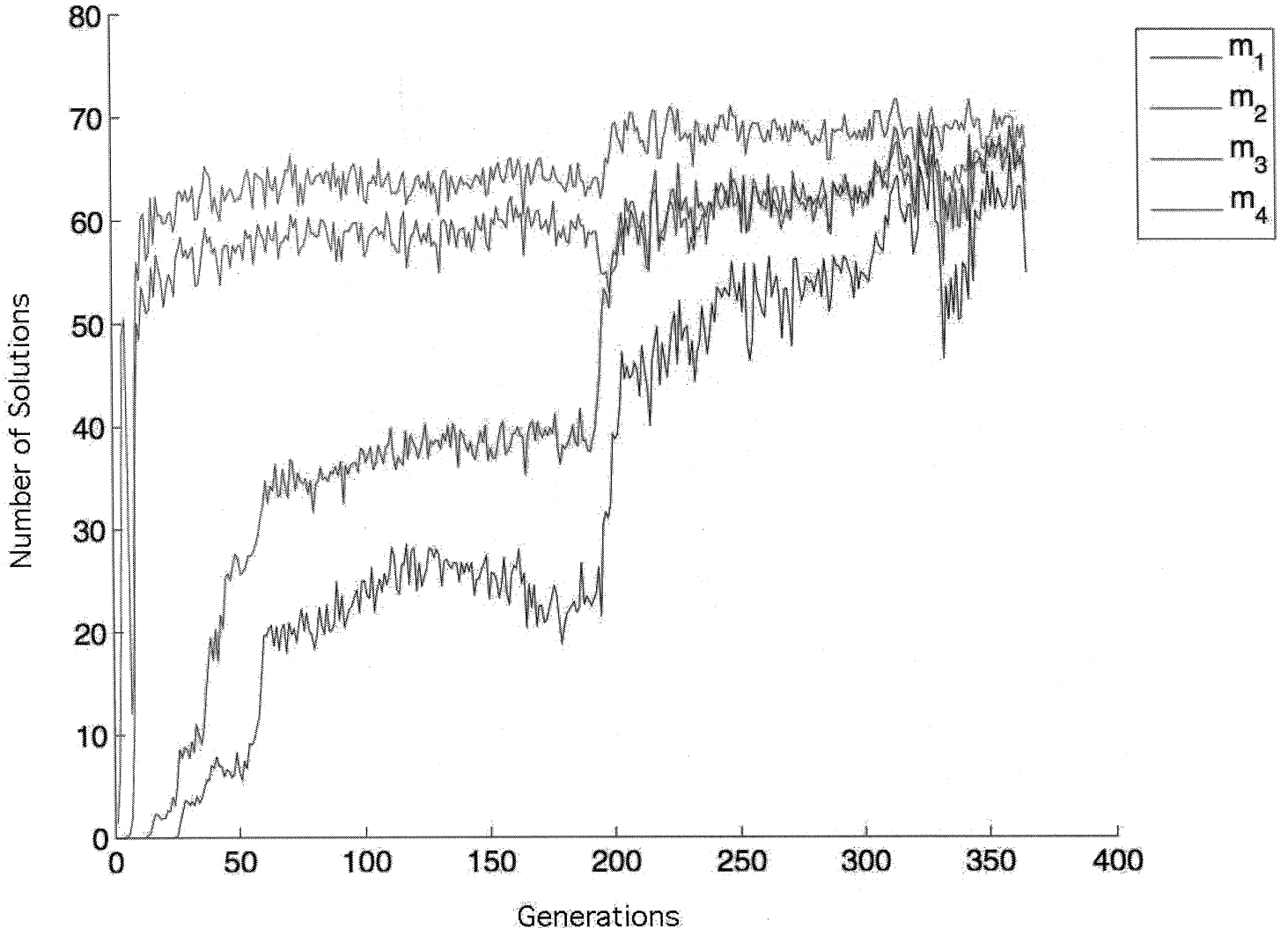


Figure 6.14: Number of solutions satisfying criteria m_1, m_2, m_3, and m_4 for Object Manipulation.

of ~~100 runs~~?
73 x 4 SWEEP SIMS

Chapter 7

Conclusion

In this work, a general purpose swarm experimentation platform, SWEEP, has been constructed and interfaced with an evolutionary computing system to demonstrate the feasibility of autogenerating swarm algorithms given a set of high-level objectives. The functionality of SWEEP has been validated through the implementation of several scenarios including object manipulation and UAV chemical cloud detection. The utility of fusing SWEEP with an evolutionary computing system has been demonstrated through the successful evolution of swarm algorithms for dispersion, object collection, and object destruction. The evolved algorithms exhibited performance that approximated the runtime performance of hand-coded solutions. The following sections will outline what has been demonstrated in this work and outline topics for future work.

7.1 SWEEP

The SWEEP simulation platform developed in this work is an extension of the original SWEEP implementation created by Palmer and Hantak [12]. Several vast improvements have been made to the core SWEEP functionalities. The parsing of the simulation specification files has been reimplemented to leverage XML and the object-oriented design of the parsing engine allows for the runtime replacement of any parsing component. The SWEEP object model has been extended and refined, resulting in a core set of functionalities that can be extended and customized. Finally, the SWEEP platform is now better suited to handle larger and more complex problems. Some of the larger-scale problems that have been addressed using SWEEP, aside from those focused on in this work, include chemical cloud detection with UAVs [16], using swarm reasoning to address the four-color mapping problem [26, 22], leveraging group behavior to explore the Martian landscape using "tumbleweeds" [15], and enhancing swarm algorithm programming through the use of aspect-oriented programming [17].

7.2 Autogeneration of Swarm Algorithms through Evolution

This work established the feasibility of autogenerating swarm algorithms through the implementation of an evolutionary computing algorithm that used a finite state machine solution encoding and utilized SWEEP as a fitness evaluation tool. Additionally, this work presented a lexicographical ranking mechanism that uses fused fitness data to address the balancing of multiple evolutionary objectives. Four scenarios were examined: dispersion, object destruction, object collection, and object manipulation. For each of the scenarios, the system was able to successfully evolve algorithms that fully satisfied the fitness criteria. Additionally, the ranking algorithm was successfully able to balance the evolutionary progress, effectively preventing the survival of expert solutions that fully address one objective but ignore others.

The use of solely high-level objectives as fitness criteria, though intuitive, does not make for an effective fitness function. The result is hundreds of generations of ineffective solutions that produce little or no evolutionary progress. This is because the fitness function does not apply enough pressure to adequately drive selection for both reproduction and mutation. Selection for reproduction and/or mutation basically says that some solution has performed well enough, relative to other solutions in that generation, that its genes should contribute in some capacity to the next generation of solutions. Implicit in this is that eventually a surviving solution will be replaced by a better performing solution created through the application of evolutionary operators.

Thus, this implies that small changes made by mutations would have at least a small but noticeable impact on the fitness of a solution. Through examination of the state machine representation and the associated mutations, it is clear that small changes made by a mutation will have little effect on a solution's fitness when only accounting for high-level goals. For example, take some generic mutation for a state machine. If the transition affected by the mutation is never fired, there is no noticeable change in the fitness of the solution. If the targeted transition is fired, but does not directly contribute to the achievement of a high-level goal, then again the mutation is ineffective. Thus, the only way for the mutation to make a noticeable change in the fitness of a solution is through making a change that directly enables the solution to improve its ability to achieve a high-level goal. So, starting with a population of completely random solutions and using only high-level goal information, the only way for a mutation to affect a positive change in fitness is to apply a random change to a solution that was randomly generated as almost functional; this event is quite unlikely.

To address these issues, first the high-level objectives were decomposed into multiple smaller subgoals. For the individual object collection and object destruction scenarios, simple lexicographical ranking of the subgoals outperformed a weighted ranking scheme in that it prevented the population domination of a sub-optimal solution with a high-scoring yet low-priority subgoal. Additionally, the splitting of the objectives provided a sufficient granularity in the scoring and ranking to allow the effects of the mutation operators to

Bibliography

[1] Payman Arabshahi. Swarm intelligence resources page, 2002.

[2] Thomas Baeck, David Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators.* Institute of Physics Publishing, 2000.

[3] Tucker Balch and Ronald Arkin. Behavior-based formation control for multi-robot teams. *Transactions on Robotics and Automation*, 1999.

[4] G. Beni. The concept of cellular robotic systems. In *Proceedings 1988 IEEE International Symposium on Intelligent Control*, pages 57–62. IEEE Computer Society Press, 1988.

[5] Eric Bonabeau and Guy Theraulaz. Swarm smarts. *Scientific American*, March 2002.

[6] Timothy Brown, Brian Argrow, Cory Dixon, Sheetalkumar Doshi, Roshan-George Thekkekunnel, and Daniel Henkel. Ad hoc uav ground network (augnet). In *AIAA 3rd Unmanned Unlimited Technical Conference*, Chicago, IL, September 2004. American Institute of Aeronautics and Astronautics.

[7] G. Di Caro and M. Dorigo. AntNet: a mobile agents approach to adaptive routing. Technical Report IRIDIA/97-12, Université Libre de Bruxelles, Belgium, 1997.

[8] Bruce Clough. Uav swarming? so what are those swarms, what are the implications, and how do we handle them?, 2000.

[9] Eric. *Swarm Intelligence ...* Someone, 1990.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1994.

[11] Paul Gaudiano, Eric Bonabeau, and Ben Shargel. Evolving behaviors for a swarm of unmanned air vehicles. In *IEEE Swarm Intelligence Symposium*, 2005.

be more easily observed and exploited by the system.

When addressing the object manipulation scenario, a combination of object collection and destruction, simple lexicographical scoring presented a challenge as giving lexicographic preference to either the collection or destruction objectives resulted in the evolution of "expert" solutions that completely addressed one objective but not the other. Thus, the evolutionary task then became transforming the expert solutions into general solutions that addressed all the objectives. What was needed was a scoring or ranking method that could balance the evolutionary progress of the population between multiple independent objectives.

In order to overcome the challenge of balancing the evolutionary progress, a data fusion technique was employed that used the raw fitness metrics to produce additional composite metrics, listed in **Table 6.5**. These composite metrics were compatible with lexicographic ranking and successfully balanced the evolutionary progress between the collection and destruction fitness metrics, resulting in the system converging to a complete solution.

7.3 Future Work

The are several paths for future work based on the results presented in this thesis. Both SWEEP and the evolutionary computing algorithm are plain vanilla implementations, focusing on clarity and correctness rather than optimal runtime performance. By extension of the core SWEEP architecture, more computationally efficient methods of simulation can be implemented. Additionally, a continuation of the work with aspect-oriented swarm programming [17] can provide SWEEP with even an more flexible probing mechanism, resulting in the construction of richer fitness functions for evolution.

As seen through the object manipulation results from Chapter 6, evolution excels at evolving feasible solutions, but can at times struggle with evolving feasible solutions that are efficient. Further work could investigate the possibility of utilizing evolution to generate feasible solutions, then piping those solutions into some other optimization algorithm, such as simulated annealing, for runtime performance optimization. This would result in the creation of a flexible toolchain for the generation and optimization of swarm algorithms.

Currently, the definition of emergent behavior is not standardized, and is primarily relegated to an "I'll know it when I see it" method of identification. A formalized definition of emergent behavior is essential to further work in swarm theory. From this definition, methods for detecting and measuring the occasion of emergent behaviors can then begin to be constructed. These methods would be an invaluable addition to a set of tools for the autogeneration of swarm algorithms. This is perhaps the most exciting future topic in swarm theory as a complete definition will require a truly renaissance approach, spanning from complex systems and information theory to sociology and philosophy.

[12] Chad Hantak and Daniel Palmer (advisor). Implementing a swarm intelligence software laboratory using finite state automata. In *Proceedings 12th National Conference on Undergraduate Research (NCUR)*, volume 3, pages 984–988. National Conference on Undergraduate Research, 1998.

[13] Orbital Research Inc. Auto-adaptive fuzzy rule base control of autonomous swarms. Sbir phase i final report, Kirtland AFB Space Vehicles Directorate, 1999.

[14] Major General Kenneth R. Isreal. Modeling and simulation employed in the predator unmanned aerial vehicle program. Technical report, Defense Airborne Reconnaissance Office, March 1997.

[15] Richard. M. Kolacinski, Daniel W. Palmer, Patrick M. Cloutier, and Jason E. Schatz. Biologically inspired design for low cost exploration of space: Swarms of martian rovers based on the russian thistle. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics*, volume 11, pages 32–38, 2003.

[16] Michael A. Kovacina, Daniel W. Palmer, Ravi Vaidyanathan, and Richard M. Kolacinski. Intelligent/adaptive operators and representation modulation for evolutionary programming. In *AIAA 3rd Unmanned Unlimited Technical Conference*, Chicago, IL, September 2004. American Institute of Aeronautics and Astronautics.

[17] Peter T. Kovacina. Use of aspect-oriented programming with swarm algorithms. In *Proceedings 19th National Conference on Undergraduate Research (NCUR)*. National Conference on Undergraduate Research, 2005.

[18] Jonathan Koza. *Genetic Programming:On the Programming of Computers by Means of Natural Selection.* Bradford Books, 1992.

[19] M. Anthony Lewis and George Bekey. The behavioral self-organization of nanorobots using local rules. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, 1992.

[20] Daniel Palmer, Chad Hantak, and Michael Kovacina. Impact of behavior influence on decentralized control strategies for swarms of simple, autonomous mobile agents. In *Workshop: Biomechanics Meets Robotics Modeling and Simulation of Motion*, 1999.

[21] Daniel Palmer, Marc Kirschenbaum, Jon Muron, and Kelly Zajac. Decentralized cooperative auction for multiple agent task allocation using synchronized random number generators. In *Proceedings of the 2003 International Conference on Intelligent Robots and Systems*, 2003.

[22] Daniel Palmer, Marc Kirschenbaum, Jason Shifflet, and Linda Seiter. Swarm reasoning. In *IEEE Swarm Intelligence Symposium*, 2005. CITY, DATE, PAGES

[23] Daniel W. Palmer, Marc Kirschenbaum, Jon P. Murton, Michael A. Kovacina, Daniel H. Steinberg, Sam N. Calabrese, Kelly M. Zajac, Chad M. Hantak, and Jason E. Schatz. Using a collection of humans as an execution testbed for swarm algorithms. In *Proceedings of the IEEE Swarm Intelligence Symposium*, pages 58–64. Institute of Electrical and Electronics Engineers, 2003.

[24] H.V.D. Parunak, M. Purcell, and R OConnell. Digital pheromones for autonomous coordination of swarming uavs. In *Proceedings of the AIAA's First Technical Conference and Workshop on Unmanned Aerospace Vehicles, Systems, and Operations*, 2002. CITY, DATE

[25] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987. COMPLETE JOURNAL CITATION

[26] Jason Shifflet. Swarm-based reasoning for the four color mapping problem. In *Proceedings 19th National Conference on Undergraduate Research (NCUR)*. National Conference on Undergraduate Research, 2005. CITY, DATE, PAGES

[27] M. Trahan, J. Wagner, K. Stantz, P. Gray, and R. Robinett. Swarms of uavs and fighter aircraft. In *Proceedings of the Second International Conference on Nonlinear Problems in Aviation and Aerospace*, volume 2, pages 745–752, 1998. CITY, DATE

[28] Gang Yang and Vikram Kapila. Optimal path planning for unmanned air vehicles with kinematic and tactical constraints. Provided to Orbital Research, 2002. [MORE]

[29] Nahum Zaera, Dave Cliff, and Janet Bruten. (not) evolving collective behaviours in synthetic fish. Technical Report HPL-96-04, Hewlett-Packard Laboratories, 1996.