AWAITING

M BRANICKY
08/23-4/05

p13: EXAMPLE OF LATENT VARIABLE

p19: [FIG 4.2] BEFORE/AFTER DISPERSIONS

p21: [FIG 4.3] AUCTION PLOTS

~~p21: KUTNER REF~~

p26: [FIG 4.7] BETTER IMAGE, TIME LAPSE?

?? p20: CAST ALG RESULTS

REGISTER
APPLY FOR FALL GRAD.
GRAD STUDIES PACKET

p38: SWARM DISPERSION EXAMPLE

?? p40: ref

p40: [TABLE 6.2] BEST FSM FOR DISPERSION

p40: [FIG 6.2] CLEAN FIGURE, DOTTED/SOLID INSTEAD OF COLOR

FIGS 6.4, 6.5, 6.6, 6.7, 6.8-11, 6.14
DIFFERENT SYMBOLS FOR DIFFERENT METRICS

FIG 6.12,3: REVERSAL OF BAR

APPENDICES

FIXES TO BIBLIOGRAPHY

SCREENSHOTS OF SWEEP

CAN YOU SEND PDF/LINK OF [19] MAK et al. AAAI. UUTC?

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 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, e.g. 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 can often 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 the 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 [29]. The basic flocking model consists of three simple steering behaviors that describe how an individual boid maneuvers based on the positions and velocities of 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 [28].

ref

While boids modeled the flocking behaviors of simple flying creatures, MASSIVE [31], a 3D animation system for AI-driven characters, 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 [16].

ref

Scheduling

One of the emergent behaviors that can arise through swarm intelligence is that of task assignment. One of the better known 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 [7]. 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 citebonabeau:SwarmSmarts. 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 known, 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 [7]. Through a simple gradient following behavior, the shortest path to a food source emerges.

Di Caro and Dorigo citecaro-antnet used biological-inspiration derived from the ant foraging to develop AntNet, an ant colony based algorithm in which sets of artificial ants 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

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 (e.g. $F=ma$, 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 environments 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.

put in exampl probe usage

Due to the data models of modern programming languages, which restrict access to memory based on a coarse 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. An example of how `Probes` might be used would be the construction of a diagnostic interface for a simulation. The `Probes` would provide the user interface with the current status of each `Agent` in the simulation, allowing the user to monitor the progress of each `Agent` individually or as a swarm. Additionally, the user would be able to direct changes in the environment, such as the availability of some resource, or even explicit changes in the behavior of single or multiple agents.

removed line about no standard form for probes

Chapter 4

SWEEP Applications

went with bolding, the li just didn't loo right

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, **chemical cloud tracking** is presented as a more comprehensive demonstration. In the scenario, a swarm of UAVs (unmanned air vehicles) 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 user-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 [17], forming an ad-hoc wireless network using autonomous UAVs or blimps [5], 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 agent's 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. Algorithm 4.1 provides a pseudo-code description of the dispersion algorithm.

The goal of each agent is to achieve and maintain a constant minimum and maximum distance from all neighboring agents. **Figure 4.1** shows the basic situations that will arise during the execution of a dispersion

Algorithm 1 Simple dispersion

```
1: if Evaluate(pos) == false then
2:     move randomly
3: else
4:     Do not move
5: end if
```

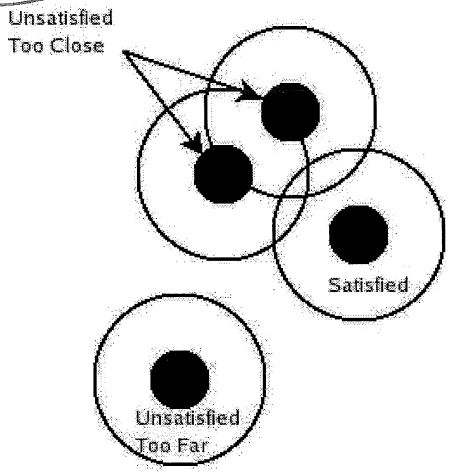


Figure 4.1: An example of the dispersion algorithm in action

algorithm. As shown in the figure, an agent can have unsatisfied positional constraints if they are either too close or too far from other agents. An agent's constraints will only be satisfied when their none of their neighbors are too far away or too close.

Put dispersion algorithm in better form, P in new dispersion figu Cleaned up th text referring the figure

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	20×20, 30×30, 50×50
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

For each region size, 50 simulations were run using different random initial positionings for each trial. The region boundaries provide an upper limit on the total area covered by the swarm, thus region size can

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.

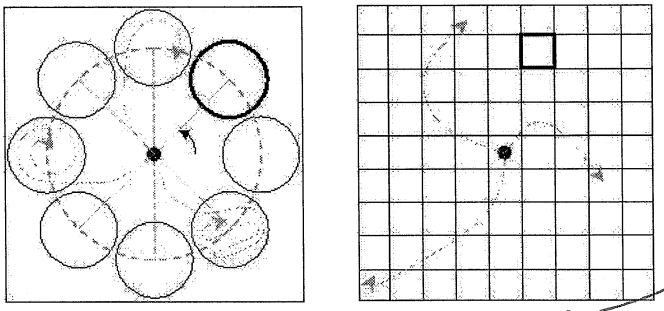
Typically, in reducing the power, size and cost of a UAV, the communication and radar capabilities of the UAV are comparably diminished. These restrictions greatly impact emergent behavior search strategies attempting to magnify the capabilities of the swarm. For example, a swarm of UAVs with low-resolution sensor footprints could locate an object smaller than their sensor footprint by merging their individual sensor readings while still only requiring minimal communication overhead [6].

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.



(a) Radially Symmetrical Search (b) Grid Symmetrical Search

Figure 4.6: Examples of symmetrical search algorithms for UAVs. In **Figure 4.6(a)**, the symmetric sub-regions form a defense perimeter around a central point. This strategy assumes that the area within the perimeter is clear and that the threat is coming from outside the perimeter. In **Figure 4.6(b)**, it is assumed the threat can be anywhere, thus each UAV is responsible for a set of subregions.

$1/2$ 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, i.e. flying perpendicular to the wind direction. For example, the symmetric sub-region search would have each UAV search their sub-region using this kind of cross-cutting pattern, shown in **Figure 4.6(a)**. 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.

Any reference info here cam from Wright-I AFB. Can I j reference that personal communicatio

Cloud Mapping

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. There are two types of mapping strategies we considered: inside-outside and dispersion.

The inside-outside method is very straightforward, as illustrated in **Figure 4.7(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

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 a 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 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 [6], "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.

We examined cases of the chemical cloud scenario with UAV swarms of sizes 5 through 15 in a region that measured 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 4.8-4.11**.

As the figures indicate, there is a performance increase when the number of agents is increased for the same-sized cloud. **Figure 4.8** 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 4.9 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 4.10 illustrates the average hit percentage of a UAV swarm of size n for any sized cloud. **Figure 4.11** 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 4.10** and **Figure 4.11**, there exists a linear improvement in the performance of the UAV swarm with the addition of each agent.

clear page

FORCE 4.8 + 4.9 ON SAME PAGE

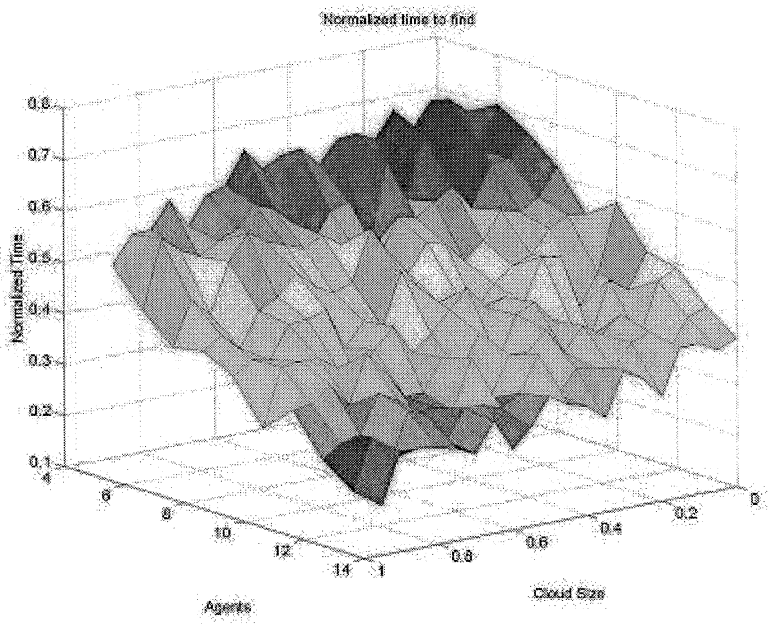


Figure 4.8: This figure shows the normalized amount of time taken by a UAV swarm to locate a chemical cloud. As expected, larger swarms were able to find similarly sized chemical clouds faster than smaller sized swarms.

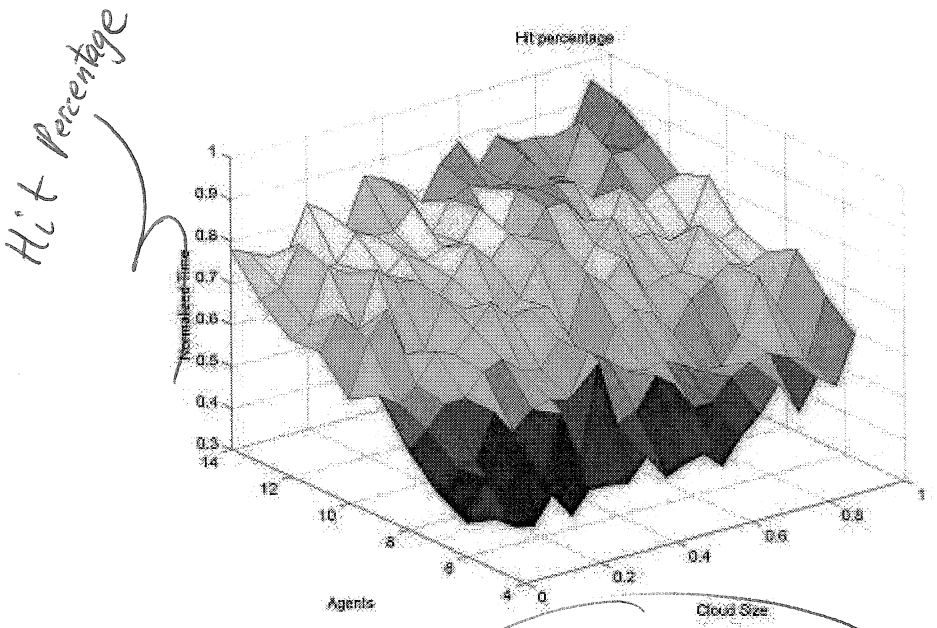


Figure 4.9: This figure shows the percentage of times that the UAV swarm was able to successfully detect the chemical cloud. An increase in the number of UAVs in the swarm increases the swarm's chances of finding the chemical cloud because more of territory is being covered.

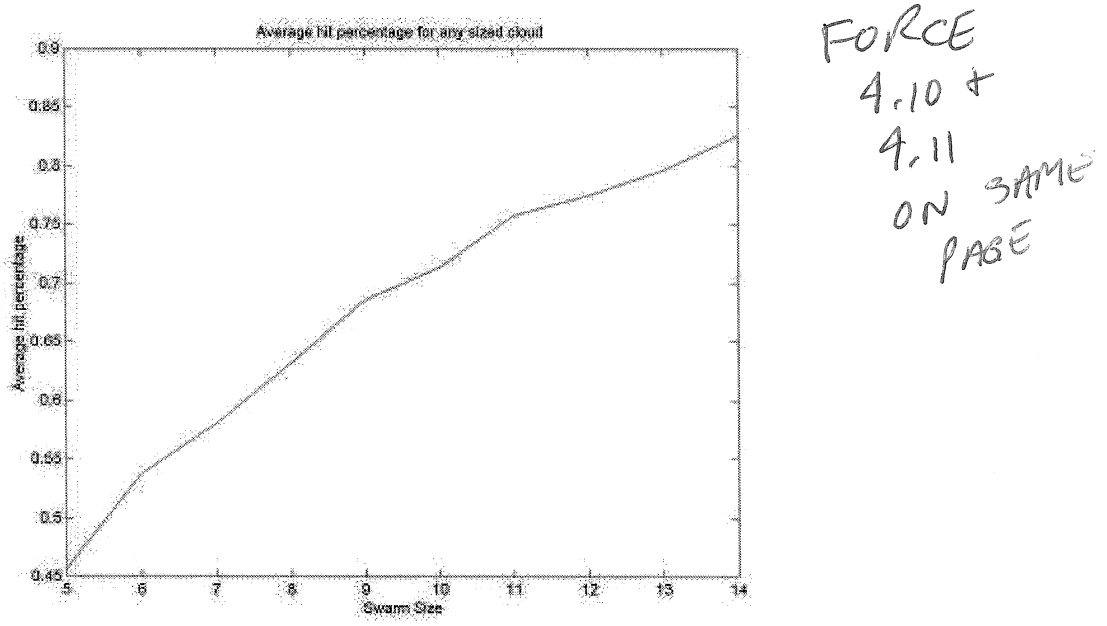


Figure 4.10: This figure shows the average hit percentage of a UAV swarm of size n for any sized cloud. With the maximum number of agents, the chances of a UAV swarm finding a cloud of indeterminate size is 83%.

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 [14]. 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. Algorithm 2 is a psuedo-code representation of basic execution flow of ECS.

added psudo-c

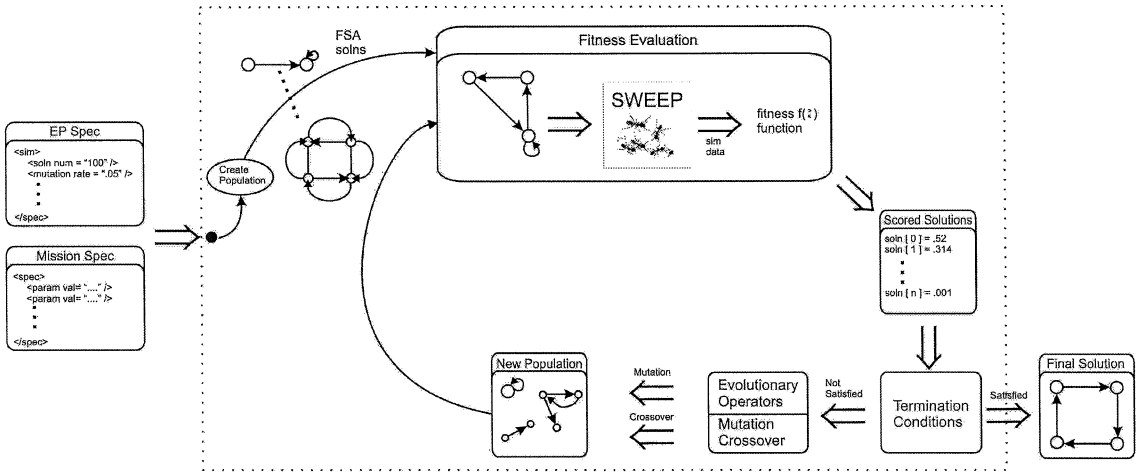


Figure 5.1: Conceptual overview of ECS.

Algorithm 2 Psuedo-code for the basic functionality of ECS

Require: isValid(parameters.xml)
1: initialize(parameters.xml)
2: mutations = generateMutations()
3: sweep = createSWEEP()
4: population = generatePopulation()
5: gen $\leftarrow$ 0
6: done $\leftarrow$ `false`
7: **while** $\neg$done **do**
8: population = population $\cup$ createRandomSolution() {Helps prevent homogenization}
9: **for all** m in mutations **do**
10: population = population $\cup$ m.selectAndMutate()
11: **end for**
12: **for all** x in population **do**
13: sweep.eval(x)
14: **end for**
15: **if** $\exists x \in$ population | isSolution(x) == `true` **then**
16: done $\leftarrow$ `true`
17: **end if**
18: gen $\leftarrow$ gen +1
19: **if** gen $\geq$ max_generations **then**
20: done $\leftarrow$ `true`
21: **end if**
22: **end while**

5.3 System Parameters

ECS has many configurable parameters that must be set in order to function. **Table 5.1** shows a sample set of parameters for evolving a dispersion algorithm. The *Objective* exists for documentation purposes and is a short description of the behavior that is being evolved. *Maximum Generations* specifies the upper limit on the number of generations that ECS can execute before terminating execution. The *Population Size* specifies both the number of candidate solutions to initially create and the number of solutions that will be allowed to survive between generations as selected through elitism. Finally, the *Number of Simulations* indiciates the number of SWEEP simulations to average over when calculating fitness scores.

Each mutation is applied a specific number of times each generation. The resulting mutated solutions are then rolled back into the population to compete for survival. When specifying how to apply a mutation, a selection method and selection size must be specified. Currently ECS supports *random selection*, *elite selection*, which selects the top n solutions and *roulette selection*, which selects solutions based on their fitness relative to all other solutions.

The specified *Actions* and *Sensors* are the primary components used to build candidate solutions. Any valid SWEEP *Action* or *Sensor* is able to be included. Currently, no validity checking is performed to ensure the compatibility of the *Actions* and *Sensors*.

Finally, the parameters related to simulating the candidate solutions with SWEEP are specified. The number and type of parameters will vary based on the simulations needs of the fitness function used. In this case, the swarm size, environment size, and maximum number of time steps are specified.

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

Table 5.1: An example set of parameters for evolving dispersion.

5.4 Solution Representation

tried to clear the transition explanation

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 (e.g. `on-object==true`). The boolean condition compares a target sensor value with an actual 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 [2]. 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.

added a bit ab the xml file

A small section of an XML encoded state machine is shown in **Figure 5.2**. Each `<state>` contains at least one `<transition>`, and only one `<default>`. The `<default>` specifies the action to execute and state to transition to when none of the defined transitions fire. For this work, the default action is `move-random` and the the default next state is always the current state (e.g. the default next state for state A is A). As seen in the figure, another benefit of the XML encoding is that extra information, such as specific rule builder classes, can be embedded within the solution without code modification or performance issues. Complete examples of XML-encoded state machines can be found in Appendix B, Appendix C and Appendix D.

Added exampl of XML file

5.5 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 5.2** lists the mutations defined in this work.

The *AddState* and *AddTransition* mutations apply large changes to the structure of a candidate state

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 o their 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

6.1.1 Description

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.

6.1.2 Implementation

For this work, a swarm of agents is 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 $d_{min} < d_{max}$. 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. The agent behaviors related to movement are: *move-up*, *move-down*, *move-left*, *move-right*, *move-random*, and *move-none*.

Each agent has the ability to sense the presence of agents in the near vecinity. The neighbor sensors are defined for the four regions around the agent (**Figure 6.1**): above, below, left and right. Encoded in each sensor is the information related to the dispersion criteria, i.e. d_{min} and d_{max}. Thus, each sensor returns `true` only when there are agents in the region that are violating the dispersion criteria of the sensing agent. Hence, the goal for each individual agent is to find a state where all of their sensors report `false`. The agent sensors available are: *neighbor-above*, *neighbor-below*, *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

I think I clea this up. Afte looking at it, realized that I origianlly h was just not good descript of what was happening.

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

The evolved solutions are scored using scenarios with randomized initial 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	30
Number of Sims	5
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-above	
neighbor-below	
neighbor-left	
neighbor-right	
Simulation	
Number of Agents	100
Environment	50x50 grid
Maximum time	400

Table 6.1: Parameters for evolving dispersion

6.1.3 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 [2]. 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. Fortunately though in this case, the fitness function as defined does produce enough information to drive evolution towards an acceptable solution.

ref

A typical evolutionary run for dispersion is shown in Figure **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. The resulting evolved state machine for dispersion is shown in **Table 6.2**.

I just cut out the section on the dynamic dispersion. One reason is that I can't seem to find the source code of the solution. The second reason is that the version of the system that I generated it on is about 6–7 generations out of date. Finally, the only reason that I tried this was because I accidentally forgot to give the agents a *move-none* behavior. I did not do any real in-depth analysis and just used the same fitness function, which in retrospect probably doesn't fit the problem all that well.

Table 6.2: Evolved dispersion state machine

State	Condition	Action	Next State

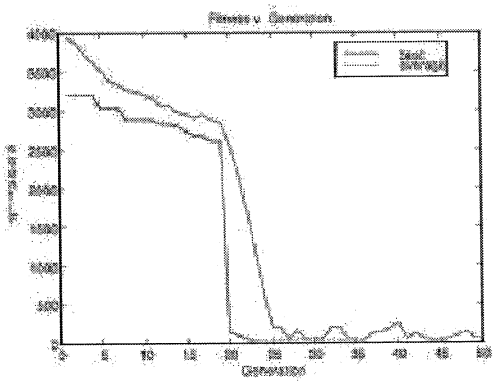


Figure 6.2: Best and average fitness scores resulting from evolving a dispersion algorithm.

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.

complete redo this section, formerly called state machine primatives

Agent Behaviors

Table 6.3 and **Table 6.4** 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 equipped 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 basic agent behavior related to movement: *move-up*, *move-down*, *move-left*, *move-right* and *move-random*. Since the focus of this work is the evolution of high-level algorithms, a utility *move-to* behavior is provided to navigate an agent to a location. The *move-to-goal* behavior navigates the agent to the known goal location, while *move-to-object_C* and *move-to-object_D* navigates an agent toward the nearest type C or D object, respectively.

The *pick-up* and *put-down* behaviors are used to interact with type C objects. An agent can only pick up a type C object when they are physically on top of the object, and an object can only be put down when the agent is in the goal area. The *first-attack* and *second-attack* behaviors are are used to interact with type D objects. Each type D object requires two separate "attacks" to be completely destroyed. Each agent has the ability to execute both attack actions. Additionaly, a side-effect of each attack is that upon a successful attack, the perfoming agent becomes disabled for the duration of the simulation.

Finally, agents can communicate through the *broadcast_C* and *broadcast_D* behaviors which broadcast to agents in a defined range the location of the nearest type C or D object, respectively.

error, thus the overall goal is to minimize the error measures. Additionally, all metrics are evaluated on the final timestep of the SWEEP simulation used to generate the raw fitness data.

when fitness gets collected

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 6.5: Basic fitness metrics for an object manipulation scenario.

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 6.5**, 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.

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 [19].

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 metrics are used to construct a set of composite metrics that do not exhibit the sequentiality conflicts seen between the raw metrics. **Table 6.6** describes these composite metrics. The use of these composite metrics enables the use of a radix sort [7] to establish a fitness-based ordering. The standard radix sort [7] moves from the least significant position to the most significant position, resulting in a sorted list. For this work, the most significant metric is m_1 and proceeds to the least significant metric, m_8. For boolean metrics (m_1–m_4), `true` preceeds `false`. **Figure 6.3** shows a simple example of how lexicographic sorting would be applied to a small population of solutions.

added example for radix sort

SO, WHICH ONE IS IT?

6.2.3 Results

Viable solutions to each of the three scenarios outlined are successfully evolved. The performance for the object destruction and object collection scenarios were 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 weighting individual metrics, which led to the development of "expert" solutions [19]. 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 eventually they dominated reproduction, resulting in an almost homogeneous population. 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 radix ranking method. As seen in **Figure 6.8** and **Figure 6.9**, the radix ranking 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.

Table 6.11: 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	-	F
	!near-object_C	put-down	B
E	on-goal	move-random	G
	on-object_C	-	B
	!near-object_C	move-to-goal	B
	!on-object_C	move-to-object_C	C
F	!holding-object_C	-	E
	holding-object_C	move-to-object_C	B
G	!holding-object_C	move-to-object_C	G

FORCE TABLE 6.10 & 6.11 ON SAME PAGE IF POSSIBLE

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 % PERIOD IS MISSING; SENTENCE IS FRAGMENT

CLARIFY WRT WHAT?

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

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 the 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 6.12**, and a simplified version of the algorithm is shown in **Table 6.13**. The simplifications include the removal of unreachable states and non-firing transitions, and the removal 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.

6.3 Conclusion

In this chapter, swarm algorithms have been successfully evolved for agent dispersion, object collection and object destruction. Composite fitness metrics, created through the fusion of raw fitness data, and a radix-based ranking algorithm were used to address the multi-objective nature of the object manipulation problem. In evolving the object collection and object destruction algorithms, first the core functionality (collection and destruction, repectively) was evolved. Then, evolution continued and optimized for execution speed. The object manipulation scenario followed this trend of first evolving core functionalies, then optimizing for speed. But unlike the evolution of the object collection and destruction algorithms, the complexity of the object manipulation problem prevented the consistent evolution of an algorithm that performed the object manipulation task quickly.

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 summarize what has been demonstrated ~~in this work~~ herein 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 [13]. 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 the version of SWEEP developed in this work, aside from those focused on in this work, include chemical cloud detection with UAVs [19], using swarm reasoning to address the four-color mapping problem [31, 26], leveraging group behavior to explore the Martian landscape using "tumbleweeds" [18], and enhancing swarm algorithm