Microbiological Inspired Swarm Intelligence for Optimization in Dynamic Environments

by

Gavan Acton
Department of Computer Science

A thesis submitted for COSC 4235 in partial fulfillment
of the requirements for the degree of
Honours Bacherlor of Science Computer Science

Algoma University College
Sault Ste Marie, Ontario, Canada
April 11, 2007 © Gavan Acton 2007

Abstract

In this paper we propose, implement and test a new approach to Dynamic Optimization inspired by microbiological swarms. Our approach makes use of the strengths of real bacteria, namely self organization, adaptation and natural selection to perform optimization. Finally, we test and show that our swarm significantly outperforms a state of the art approach by achieving comparable optimization in environments moving three times the speed.

Acknowledgement

I would like to thanks all the people who have helped me with their time, insight and most of all patience namely Dr Simon Xu, Proffesor Danny Reid, Dr Jay Rajnovich, Dr Edna James, Dr Vitorino Ramos and my wife Lindsay Page.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

A Dynamic Environment(DE), poses a difficult task to optimization algorithms(OA) as the environment is continually changing. As a result, good solutions to a problem may vary from iteration to iteration. And so, when it's possible to model the DE, off line methods are the best as they can efficiently approximate the best solution [1]. However, in cases where the DE is constantly in flux with open ended novelty, Evolutionary Computation(EC) and Swarm Intelligence(SI) methods have shown to be adept at tracking the progress of optimal solutions through the search space [21].

Many types of SI based on the characteristics of biological swarms(Birds, Ants), have been used for Dynamic Optimization(DO). However, no attempts have been made to use the characteristics of microbiological organisms. A microbiological organism like bacteria live in a highly dynamic world, where changing temperatures, food sources, and predators are factors that require a wide variety of skills to survive. And so, the minute to minute life of a cell is a type of implicit optimization problem that is known as foraging [20]. Here, cells attempt to maximize their environment by finding adequate food sources and breeding areas while minimizing the energy they expend finding these locations. Therefore, as their foraging is a dynamic optimization problem, it makes sense that cells would have a host of skills that could be applied to optimization algorithms in dynamic environments. There are several important traits of the micro-organisms that allow them to be effective at foraging in a dynamic environment.

The reason we find bacteria living in the near boiling water in underwater hotspots and other places like our counter tops is due to their adaptability. Adaptation is achieved by changes of the characteristics of a cell through a process called mutation. Mutation modifies the cell's genome which results in a change in the physical structure of the cell as well as the decision making processes. These changes may result in a novel feature that could help the cell survive. Mutation is also very important as it diversifies the genome base of the whole colony [2]. This gives the colony greater flexibility and resilience as the colony is less likely to be wiped out by a sudden change in the environment. A well known example of this is the anti-biotics resistant micro-organisms, who have developed traits of resistance through mutation. Adaptation allows micro-organisms to gain characteristics to better survive in a dynamic world. Dynamic optimization algorithms that use characteristics of adaptation could benefit by gaining better methods to track optima but also have diversity to successfully survive a drastic change to the environment.

An important trait of microbiological swarms is that they are constantly under the pressure of natural selection. This process only selects the most fit individuals in a population through their competition for a finite amount of resource. Mutation here is the process that allows for natural selection to take place by discovering characteristics that enable the cell survive in a competitive environment. Natural selection acts as a filter only allowing cells who are the most specialized to their environment to reproduce. This specialization of the genome through natural selection could be applied to DO where it could increase the efficiency of an optimization algorithm by developing characteristics that suit the environment.

Another reason that bacteria are effective at surviving in a constantly changing world is their ability self organize. With a wide range of communication abilities bacteria have the ability to work together for a common purpose giving the colony greater abilities to survive in the dynamic conditions of their environment. One amazing example of this is the actions taken by a bacterial colony when placed under oxidative stress (lack of oxygen). In this condition, spontaneous self organization occurs where the cells aggregated such that a convection pattern emerges allowing each cell to be momentarily exposed to oxygen rich fluid [2]. This example of 'bioconvection' is one example of many displaying how bacterial colonies have excellent abilities to work together to increase their survival. Our hope here is that the DO algorithm with characteristics of self organization should be to find optimal solutions in the search space.

Finally, cells and bacteria may be the most studied organisms on the planet and consequently there exists an enormous amount of literature describing their processes. This gives us a wealth of information that is readily available without needing to enter a biology lab. And so, we can pick and choose various features of microbiological organisms to inspire efficient DO algorithms.

In this paper,we propose a novel algorithm based on the characteristics of a microbiological swarm for the purpose of DO. Furthermore, as a proof of concept, we implement, test and compare this algorithm in a benchmark test in order to determine its effectiveness against other state of the art algorithms.

Our paper is structured as follows. Chapter 2 provides the reader with the fundamental concept of optimization as a method for problem solving. Furthermore, it outlines characteristics and concepts that are used to evaluate both optimization algorithms and optimization problems This is followed by chapter 3 with a discussion of modern optimization approaches with emphasis on systems using SI. Finally we discuss our proposal(chapter 5), its design and implementation(chapter 6) as well as the results from a series of tests(chapter 7). The paper ends in chapter 8 with discussion of future avenues of research work and concluding thoughts.

Chapter 2

Optimization Theory

In the automated search for a solution to a complex problem, a brute force search through the problem space is often too expensive to perform. Optimization Theory(OT) is the branch of computer science that deals with finding optimal algorithms(OA) that attempt to minimize the search time and maximize the solution. The core assumption of the optimization process is that good solutions will be grouped together in the search space. And so, information about current solutions should be used to locate new and better solutions without having to randomly guess at their locations. Typically, OA have been successfully applied to real world problems as well as NP-Complete problems such as the Travelling Salesman Problem(TSP) with great success. Recently, new types of optimization algorithms based on a form of Artificial Intelligence(AI) have been developed, resulting in state of the art performance[7].There are many differences in how these algorithms perform optimization, however the fundamental concepts of optimization theory are at the heart of their solutions.

In this chapter we discuss the concepts of optimization as well as characterize the types of problems optimization algorithms work with.

2.1 Optimization Definitions

A formal definition of optimization is the following [14]. In any optimization problem P, there exists a solution space S that contains every possible solution to the P and a **feasible** solution space F such that $F \subseteq S$. The feasible solution space contains all solutions to P which meet some condition. The objective of any optimization algorithm is to find a solution $s \in F$ that will minimize a cost function c as described by Equation 2.1.

$$c(S) = min\{c(T) : T \in F\} \tag{2.1}$$

Also, each problem P must have an **objective function** f that defines the quantity to be minimized or maximized. This function acts upon a **set of variables** $x = \{x_1, x_2, ..., x_n\}$ such that $f(x_1, x_2, ..., x_n) \rightarrow s$. Here f maps a sequence onto a candidate solution. The **dimension** of P is defined as the number of elements in x, or $||x|| = n$. We also define the **landscape** of P as the shape of the solution set. Finally, the **neighbourhood** N of any candidate solution is all other candidate solutions $c$such that $s, c \in S$ and $|s - c| \ < \ r$. Where r is the euclidean distance $r^2 = x_1^2 + x_2^{2,...,} x_n^2$.

2.2 Optimization Method Classes

Optimization algorithms search for an optimum solution by iteratively transforming a current candidate solution until an acceptable solution is found[13]. A defining feature differentiating these two methods of OA is the type of solution the algorithm finds from iteration to iteration. The first class of algorithms are known as local search algorithms. These algorithms use local information in the search space to locate optima. As a result, local search algorithms will only converge on a local optima, with no guarantee it will be the globally optimal. On the other hand, a global search algorithms do not restrict their view to neighbourhoods but rather explore the entire search space.

The OA methods can be further broken into two more classes by the type of transition rules used at each iteration. Rules that contain a random element are said to be stochastic and are defined as Stochastic methods. The outcome from each iteration to the next cannot be predicted. Deterministic Methods do not contain randomness and the outcome of each iteration can be predicted.

A final important classification is based on the environment in which our OA is to work. We define Static Optimization(SO) to be all optimization which happen on static environments. That is, for each iteration of the search, the values in the search space are unchanging. Dynamic Optimization(DO) is optimization where the search space is in flux during the search process. The objective of DO is different from SO with an additional constraint to deal with a environment whose optima may be moving. And so, the aim is to track the progression of optima through the search space as closely as possible while minimizing the cost of the search (Equation 2.1) [1].

2.3 Optimality Conditions

In any optimization problem space, there are areas where the solutions are of higher and lower quality. Mathematically, we define the areas as optima of different types depending on if we are performing a maximization or minimization of the problem space. In the case of maximization we define the global maximum as the best solution of all the set of candidate solutions. A local maximum as a best solution within a neighbourhood N. And a weak local maximum as the best of the worst within a neighbourhood. Figure 2.3 illustrates these cases.

2.4 Optimization Process: Exploration vs Exploitation

As a final thought to the process of optimization, OA must strike the right balance between two contradictory objectives of Exploration and Exploitation [21]. Exploration is the search characteristic of an algorithm that aims to find a diverse set of solutions

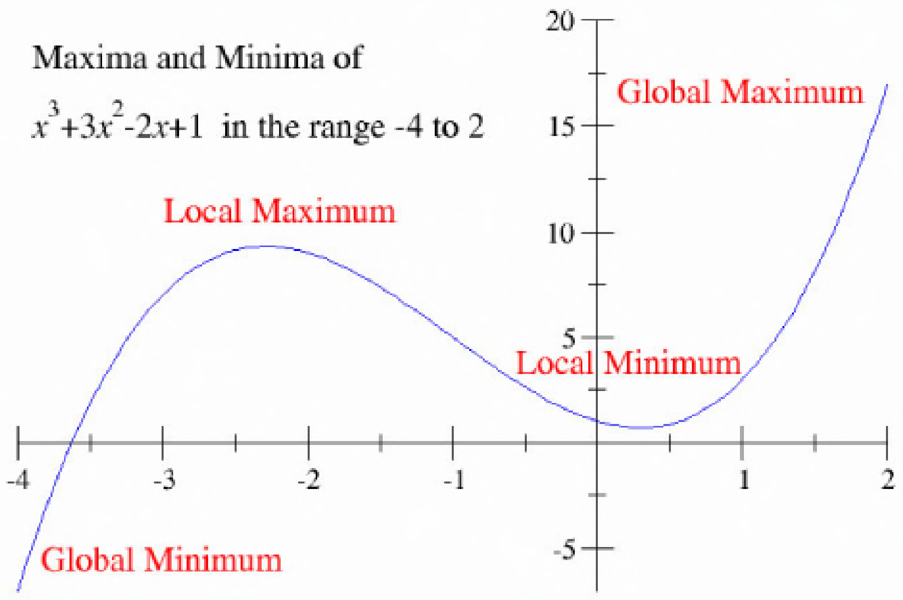


Figure 2.1: Optima for Maximization Problem.

[13]. This results in generally slower convergence onto optima but gives the algorithm a greater opportunity to locate global optima and avoid being trapped in local ones. The extreme case of this is the purely exploratory algorithm. This search process can be thought of as a random search, as the current position in the search space has no effect on next position. On the other hand, Exploitation is the search characteristic where an algorithm makes the most use of current solutions[21]. In this case, the algorithm will converge upon optima at a quicker rate as the search is focused within the neighbourhood of the candidate solution. However, the risk of having too much of the exploitation characteristic in the algorithm is that it will converge on a local optima and will not have the exploratory capacity to discover new solutions outside the local optima. As a result, a balance of exploration and exploration is needed for optimization algorithms to be efficient in wide variety of problems.

2.5 Characteristics of Optimization Problems

Now, in each optimization problem, there exist six important characteristics [13].Each of these serve to define the problem space. As a result, variations among these characteristics can be good predictors of how hard a problem will be for an optimization algorithm.

- The **number of variables** in x that are used in the objective function. As we increase the number of variables, we increase the number of candidate solutions in S by the size of each dimension. Any problem with more than one variable is called multivariate or multi-dimensional.

- The **types of variables** in x plays an important role in characterizing the problem. If all the variables in x are elements of the real numbers $x_j \in \Re$ then we characterize the problem as being a continuous value problem with a theoretically infinite search space only bounded a computers precision. When $x_i \in Z$ then we say that problem is an integer or a discrete optimization problem. A modified type of discrete optimization are combinatorial optimization problems that attempt to find the best permutation of a set of variables. An example of this type of problem is the TSP, or other graph problems

- The **degree of non-linearity of the objective function** describes what type of function is being used but also the shape of the **fitness landscape**. Linear objective functions are of the form

  $$f = aX + b$$

  and can be seen to have simple gradients and that are easy to optimize as once a candidate solution s is found, another solution of equal or better quality will be in the near neighbourhood of s. Quadratic objective functions will take the form

$$a_1x^2 + a_1x + x = 0$$

and will have characterized with more optima generally harder to optimize. The most difficult type of objective function are non linear their environment provides the least amount of information that can be derived by a current location.

- The **number of optima** can serve to characterize the problem and its environment. When only one clear solution exists, we can consider the problem to be unimodal. However, in the cases that more than one optima exists, we consider the problem to be multi-modal.

Each of these characteristics are important to defining an optimization problems difficulty. In the case of Dynamic Environments, addition characteristics are required.

2.6 Characteristics of Dynamic Optimization Problems

While there have been a few authors that have defined various characteristics of dynamic environments [13] [1] [8] [9], the following four characteristics are agreed upon.

- the **frequency of change** of an environment describes how often the environment changes. This description can also be defined as how many evaluations of the objective function can be made between each change to the environment. In the example of finding the optimal route home, the frequency of change is very high and optimal routes may change minute by minute. However, if the optimization algorithm only received updated information about the environment every hour, our frequency of change would be very low.

- The **severity of change** is a defined as the euclidean distance between an optimum's previous location and its current location in one update to the environment. The severity is calculated by (distance moved)/(max movement possible) in the environment.

- **Predictability of change** is the characteristic of dynamic environments that defines how much randomness is in the environment. If the changes are purely random then it is said to have no predictability. However, if there exists a pattern to the changes, the environment is said to have a high predictability.

- **Cycle length / Cycle Accuracy** describes the cyclic nature of some environments. For problems that have a period to the movement in a specific pattern, it is possible to describe the cycle length as the number of updates to the environment for a complete cycle to occur. The Cycle accuracy defines how close the next of each cycle is from one another.

Each of these characteristics in DO can be used a marker for the difficulty of a given problem. Because each of these characteristics introduce very difficult dynamics for optimization algorithms, many different approaches have been attempted.

Chapter 3

Modern Approaches

One approach to DO problems is an active field of AI known as Swarm Intelligence(SI). It is abstractly based on biological swarms such as ants, fish and birds. SI examines the collective intelligence or problem solving that stems from the interaction of swarm members. Ants, for example have the ability to collectively find the shortest path between a food source and their colony through the use of two simple mechanisms [7]. The first is, that as they walk, they deposit a chemical known as pheromone. The second mechanism is that they are probabilistically more likely to follow trails with higher pheromones levels. As a result, shorter trails to and from the food source will accumulate more pheromones, as ants will make more passes to and from the colony on them. This results in shorter trails having higher pheromone levels which in turn attract more ants along to the trail. And so, through distributed decision making, the ant colony collectively finds the shortest path between two points. In the following section we will discuss the important qualities of SI as well as how SI has been applied towards DO.

One of the main qualities of SI is that it is based in Self-Organization(SO). Biological systems self organize through two types of feedback loops. A positive feedback loop is a process that will bring greater change to a system [10]. In an ant colony, the positive feedback loop is the accumulation of pheromone on a trail, which in turn attracts more ants to the trail. The negative feedback loop is the evaporation of pheromone from the trail. SO systems are often very sensitive to the parameters of the feedback loops. For example, if the evaporation rate of pheromone is too high, then a trail can never be formed. On the other hand, if the evaporation rate is to low, new paths will never be examined even if they are of a better quality [7]. And so, the SI systems use feed back loops to organize their agents towards a common purpose.

The second quality of SI is the property of emergence. This is understood as the idea that the 'whole is greater then the sum of its parts'. The emergent property of the ants pheromone laying and trail following is that a shortest paths is found. However, clearly ants are not aware that they are finding the shortest path but are rather simply responding to stimuli in the environment. And so, a property of the SI problem solving is that agents cannot be explicitly programmed to solve the problem, but the solution must emerges from their interaction. Unfortunately, humans are not very good at predicting the outcomes complex adaptive systems like swarms [12] and so SI researchers often make use of biological swarms to develop applications[13].

Optimization algorithms that use SI have a population of agents that move through the search space. As each position in the search space represents a candidate solution the problem. A form of parallel optimization process occurs as the swarm uses information about its members/and or the environment to make decisions on how to move. SI based optimization algorithms are considered to be from the stochastic optimization class as randomness is used to reflect the probabilistic nature of the swarms [13].

3.1 Principles of Swarm Intelligence

Above and beyond the qualities of SI swarms listed above are six basic principles [19]. Each of these principles are important as they outline the basic rules that OA based in SI must follow in order to be successful at optimization.

- The first is the **proximity principle** and is that the group should be able to make elementary space and time computations. In other words each agent should have the ability to make decisions based on its local environment as well as move through it.

- The second is the **quality principle**. This defines that the group should be able to assess the quality factors in the environment. For example, an agent should be able to asses the strength of pheromones are in an area.
- The third is the **principle of diverse response** which outlines that the group should not allocate all of its resources along excessively narrow lines. The swarm should allocate its' resources along as many lines as possible insuring against sudden changes to its environment.
- The fourth is the **principle of stability**. The group should not shift its behaviour from one mode to another upon every fluctuation of the environment
- Finally, the fifth is the **principle of adaptability**. This is, if changing behaviour will likely give a good return on the time spent switching, then it should be done.

Each of these principles are important they reflect a problem solving method that is used by real swarms in nature. Optimization Algorithms implementing these principles, have been successful at DO as they can be flexible in different environments [13].

3.2 Particle Swarm Optimization (PSO)

One approach to dynamic optimization problems is called Particle Swarm Optimization(PSO) and is based on the collision avoidance of birds[13]. The main concept is that a swarm of particles, fly through a problem space by emulating the success of neighbouring individuals [13]. We will first outline the basic PSO model for static optimization and discuss how it has been modified for dynamic environments.

3.2.1 Basic PSO

The basic PSO is a swarm where each agent known as a particle is only aware of its current position, its current direction vector, as well as two vectors known as the cognitive component and social component.

The cognitive component $c_1 r_1(lbest - x_i)$, points from the particle's current position towards the particles best found position in the search space. This position is known as *lbest* The cognitive vector is scaled by a constant c_1 and an random value r between [0,1]. The constant c_1 is a user defined variable that can be modified for different behaviour. This constant would be changed from problem to problem to increase the effectiveness of the algorithm. The random value r[0,1] has the effect of adding randomness to the particles response at each time step. As the particle moves in a direction away from its personal best, the magnitude of the cognitive vector will increase. However, if the particle continues to find a personal best locations at each iteration, there will be little attraction towards the *lbest* position. If the particle's path is unsuccessful at finding better locations, it will experience greater attraction towards the *lbest* position. This attraction dynamic encourages the particle to explore the area around its best found position.

The social component acts much like the cognitive component but is attracted to the best found position among all particles. The social component is defined as $c_2 r_2(gbest - x_i)$ where *gbest* is the position with highest value among all of the particles. Like the cognitive component, the social component is scaled by a c_2 value and a random between [0,1]. The social component ensure that the particles fly towards a position that is optimal within the whole swarm.

Now, the movement occurs by calculating a movement vector composed of the previous direction + the cognitive and social components (Equation 3.1 and respectively Equation (3.2)

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{3.1}$$

$$v_{ij}(t+1) = v_{ij} + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[gbest_j(t) x_{ij}(t)] \tag{3.2}$$

The basic PSO is described in pseudocode in Algorithm 1 based on a description in [13] . The optimization process of the basic PSO can be thought of as an

Algorithm 1: Basic PSO Algorithm

```
let f be our fitness function let S be our search space
Create and initialize a n dimensional swarm S
begin
    while not done do
        foreach particle $p \in S$ do
            set the personal best position
            if $f(p_x) > f(S_{pbest})$ then
                $p_{pbest} = p_x$
            set the global position
            if $f(p_x) > f(S_{gbest})$ then
                $S_{gbest} = f(p_x)$
        foreach particle $p \in S$ do
            update velocity using equation 3.2
            update position using equation 3.1
end
```

initial exploration phase with acceleration towards an optima by all particles. The exploitation process begins as the particles decelerate as they moves upwards into an optima. This exploitation gradually shrinks the distance between each particle by reducing to the diameter of the swarm and as well as its diversity [9].

3.2.2 PSO Variants for Dynamic Environments

PSO Variants have been successfully applied to DO problems. The basic PSO has two issues that need to be addressed in order for the swarm to perform well on tracking tasks in the dynamic environment.

The first issue is a outdated memory problem. As the environment moves, both the value of the cognitive and social components may be out of date and incorrect [8]. A simple solution to this problem relies on the particles being aware that a change has been made. In this case, a simple re-evaluation of the gbest and pbest informs the particle of the change in the quality of its location [6].

The second issue that PSO must deal with in dynamic environments is that as a swarm converges on an optimum, it looses diversity as the average diameter of the swarm shrinks [6]. In the case where the optimum has not moved outside the diameter of the swarm, after few iterations the swarm will re converge on the peak. However, in the case that the severity is great enough that the new position of the peak is outside the diameter of the swarm, the lack of diversity will often not allow the swarm to move quickly enough to find the moved optimum. It may be helpful to imagine each particle to be very close together resulting in very short difference vectors between the gbest and the current position. The result is that short vectors do not allow for large displacements as defined by the update equations 3.2, 3.1. In order to deal with this case, some techniques attempt to diversify the swarm when a change has occurred [8]. However it has been suggested that more permanent solutions will be a better approach [8].

One attempt to deal with DO is to augment the social component of the particles. In these approaches, particles are informed of the quality of the position of a group of other particles within the swarm. The group that the particle belongs to is called a subswarm who effect the particle by drawing the social component vector towards the best position of group. Investigations on different subswarm topologies have been done with many interesting findings [16],[17], [13]. Unfortunately, there is some recent evidence that fully informed particles(aware of all other particles) are less able to perform in dynamic multi-modal and static environments [25].

Other types of PSO schemes have been the use of a 'inertia' weight that essentially act as a weight pulling particles in their current direction making them less susceptible to local optimum[18].

One example of a very successful approach is Branke's Self Organizing Scouts(SOS) [6]. The idea here is that a colony of charged swarms are initialized into the environment. The analogy is to atoms where a charged particle will repel other particles of the same type. The SOS employs a principle of exclusion where an optima is only allowed to have one swarm tracking it. Furthermore, the SOS systems makes use of anti-convergence which ensures the there will be at least one free swarm patrolling the area. The result is a swarm that is effective at monitoring multiple peaks while at the same time being flexible enough to discover new optima should they appear in the environment.

Afinal example of a PSO variant, makes use of evolutionary mechanisms such as mutation and speciation to modify how the PSO acts during the optimization process [18]. Here the PSO uses mutation by continually changing the subswarm that particles belong to. This is complemented with speciation that occurs as the swarm reaches a converged threshold based on the diameter of the swarm. When the threshold is reached, the swarm is considered to have converged and its particles split into subswarms of charged particles forcing some particles to move away from the center of the swarm. The result is that new subswarms are formed. Each new subswarm can then converge onto new optima and and split again. This converging/splitting process continues until the maximum number of subswarms is reached. This in turn gives the entire swarm a better ability to track movement of optima as well as provide greater flexibility when the environment has drastic changes [18].

Particle Swarm Optimization is a well researched and well developed area for DO with many successful implementations. The strenght of the PSO approach is its simplicity and efficiency of the algorithm. Each iteration requires few operations to construct the movement vector(Equation 3.2) and update its position. Even with additional features such as subswarms and charged particles the PSO algorithms remain a simple yet effective method to perform DO.

3.3 Self Regulating Swarms

A very unique SI based approach to DO is proposed by Ramos in [21]. Here he implements an innovative technique that hybridizes Ant Colony Optimization(ACO) with a simple evolutionary mechanism of reproduction to achieve high tracking rates in dynamic environments.

Ramos's swarm called the Self Regulating Swarm(SRS) is a population based optimization algorithm where each agents(ants) moves thought the search space depositing pheromones based on the fitness of the location. Areas of poor quality receive little to no pheromones while areas of higher fitness receive more pheromones. This creates a pheromone field around the landscape whose greatest pheromone densities are in areas of higher fitness. As each ant is attracted to the pheromones in its neighbourhood, this causes the exploitation of areas of higher fitness as it produces an auto catalytic positive feedback loop like the ones we describe in the principles of Self-Organization. As a result, tracking occurs as ants explore then exploit the landscape by the laying and then following pheromones trails in the environment.

Ramos also includes very non-antlike features that we view as micro biologically based. The first feature is environmental pressure. In the real world, the environment produces pressure towards successful behaviour. Cells in excellent habitats reproduce faster and live longer then cells in inhabitable habitats. In the case of the SRS, each individual i in the population receives a relative fitness based on the fitness of its current location given by the Equation 3.3

$$rf = \frac{\Delta_i}{\Delta_{max}} \tag{3.3}$$

Where $\Delta_{max} = |z_{max} - z_{min}|$ is the difference between the maximum fitness z_{max} and the minimum fitness z_{min} of the entire population. In the case where we wish to maximize the environment then $\Delta_i = |z_i - z_{min}|$. When we wish to minimize the environment then the equations complement is $\Delta_i = |z_i - z_{max}|$. This relative fitness value between zero and one acts as environmental pressure by scaling the probability of reproduction and the amount of pheromone deposited by the ant as seen in Equations 3.5 and 3.4.

$$T = \eta + p\frac{\Delta_{[i]}}{\Delta_{max}} \tag{3.4}$$

$$P^{*} = P^{**}(n)\left[\frac{\Delta_{[i]}}{\Delta_{max}}\right] \tag{3.5}$$

The pheromone depositions rate of an ant is based in part on its relative fitness(Equation 3.4). At each time step, an ant will add to the its current location a constant amount of pheromone N plus an additional amount scaled by its relative fitness.

The strengths of the SRS is its relatively simple agents(ants) that perform simple updates, yet the emergent behaviour from the swarm is state of the art tracking in dynamic environments [21]. Furthermore, the use of the relative fitness allows for a continual pruning of the swarm agents based on fitness of their location in the environment. Poor areas are not explored in detail while areas of high fitness are exploited by the swarm.

The main weakness of the SRS is that it does not have method to adapt to the environment past its initial settings. For example, the pheromone evaporation rate is a fixed constant. In environments of high severity where optima move often and by large amounts, the pheromone field left by the ants is constantly out of date with the current environment. As a result, swarming will occur towards locations that no longer contain optima. This is equivalent to the outdated memory issue in PSO's optimization process. However, in the case of the SRS, there is no method to re-evalute past locations as this would require the re-revaluations of the entire pheromone field and therefore would be prohibitabily expensive to perform. As a result of not being able to adapt to the environment, the SRS cannot be flexible and track optima in wide variety of environments.

In closing this chapter, it is important to note that our focus on DO is restricted to approaches using SI. There exist many other approaches including Genetic Algorithms, and other forms of evolutionary computing techniques to perform DO. The reader interested in learning more about these types of approaches are encouraged to read the following papers [5], [15], [26], [11], [24].

Chapter 4

Bacterial Background

As our intent is to develop an optimization algorithm based on the characteristics of microbiological swarms, we must have a good understanding of how these swarm work. With this in mind, in this chapter we carefully examine in detail how a specific microbiological organism (E.Coli) achieves self-organization and adaptability in a dynamic environment. Many of these characteristics will then be applied to our algorithm in following chapter.

One of the most well studied types of bacteria is E.Coli. It is composed of a plasma membrane, cell wall and a capsule which contains the cytoplasm and the nucleoid. In addition to these it also has a rotating flagellum used to move in the environment and a pili for transferring genes to other cells. The E.Coli cell has the width of 1 nanometer, is 2 nanometre in length and weighs approximately 1 picogram [20]. E.Coli are social bacteria as they aggregate to form colonies of millions of cells.

Colonies are extremely adaptable to various environments due to mutation. The E.Coli cell has genome of 4,639,211 genetic letters which are arranged into 4,288 genes[20]. Reproductive mutation occurs at a rate of 10^7 per gene, per generation. And considering that a bacteria like E.Coli will split into two separate cells every 20 minutes, new generations are very frequent. As a result, there is a constant trial and error process where new cells are created with new characteristics that are then tested by their survival in the environment. Mutation will also occur due to the environmental stress. Recent studies show that when bacteria are exposed to an oxidative stress(lack of oxygen) they may spontaneously mutate [2]. This form of latent mutation allows cells to attempt a last effort in order to survive in an environment that they are not well equipped. If the mutation is successful, this characteristic will be reproduced in the cell's offspring as it reproduces. And so, mutation causes a constant diversification of the gene pool. This results in E.Coli colonies have the ability to withstand rapid changes in their environment but also have an efficient mechanism for finding important characteristics to survive.

Bacteria move through the environment in a motion pattern called taxes. These motion patterns can be caused by a host of different stimuli(light, heat, oxygen,...) and result in the cell swimming up or down the stimuli gradient. Figure 4.1 illustrates a stimuli gradient where a cell will swim towards the higher concentration of stimuli. Chemotaxes is the movement pattern in response to chemicals in the envi-

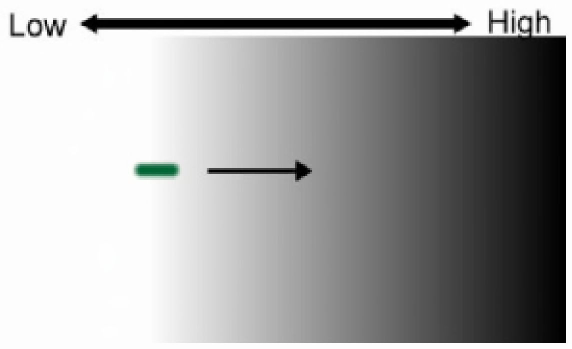


Figure 4.1: Stimuli Gradient

ronment and occurs by the bacteria sensing an attractive or repellent chemical. For example under experimental conditions, when peptone(food) is introduced into the environment, bacteria will quickly move towards the source at a mean speed of 10-20 nanometre/second! This is equivalent to a person swimming 100 meter freestyle in the range of five to ten seconds. As the cell swims, its movement is subject to random changes in direction of about 30degrees per second [20].

Intra-Cellular communication happens in at least three methods in order to self organize. The first method is called chemotactic signalling which is the secretion of attractant/repellent into the environment in response to stimuli. Chemoattractants induce a cell to swim towards its source while chemorepellents induce a cell to swing away from its source. As a cell detects a new source of food, it will secrete a chemoattractant chemical known as cAMP that diffuses into the environment [10]. In response to the chemoattractant, other cells release their own cAMP and swarm up the gradient. This self-organizing process, called relay, effectively alerts the colony of the discovery of a food source as well as its general location [10]. Furthermore, chemotactic signalling can happen in various forms that include short and long range signals. Long range chemorepellent can help a colony expand outwards in search of new food sources. On the other hand, short range signals allow the cells to interact locally in order to create form in specific formations like those used to swarm towards a food source. Another method of intra-cellular communication is through gene activation and deactivation called gene expression. In this form of communication, cells may signal neighbouring cells to express specific genes. This causes a ripple effect among the bacteria where specific genes are temporarily modified in neighbouring cells. What happens next depends on which genes are expressed, however in the case of the Proteus Mirabilis bacteria, they stop dividing, grow in length and grow extra flagella [2]. They will then group together to form collective rafts which move efficiently on a hard surface. Once they have moved away form their initial location, they will differentiate back into their normal shape. This example illustrates the power of communication through gene-expression. Another form of communication is through conjugation where one cell will pass on information in the shape of a gene to another cell. This form of specific gene transfers allows the colonies to create a type of hereditary memory [2]. Therefore, bacteria have a wide variety communications channels which they use to act effectively in groups.

In the following chapter we outline our proposal based on the characteristics discussed above.

Chapter 5

Proposal

Many modern approaches have used the characteristics of insects and animals to create efficient optimization algorithms for dynamic environments. However, there has been no research on the application of microbiological swarming for DO. This proposal centres on important qualities of bacterial swarming namely, self-organization, adaptation and natural selection. In this chapter, we first the outline characteristics that a DO algorithm must have if it is based on the swarming of microbiological organisms. This is then followed by a detailed description of our algorithm based on the outlined characteristics.

5.1 Algorithm Requirements

With the basis for developing an algorithm using SI in mind, we will use a population of agents that we call cells, who have specific characteristics like real microbiological organisms found in nature. Here we outline specific characteristics that should be implemented in our agents.

The first characteristic our cells should have is the ability to evaluate its current position and release a chemoattractant if the location contains a good source of food. Chemorepellents should be released into the environment if the cell density in the area is above a threshold.

The second characteristic is that cells should have the ability to move through the search space in response to stimuli presented by its local environment. Cells should have no explicit memory of the environment or its previous states. Chemoattractants should steer the bacteria up the diffusion gradient while chemorepellents should steer the bacteria down the diffusion gradient. Any movement should be subject to an exploratory random movement dynamic. Furthermore, cells will need to move at each time interval in a direction and speed.

The third characteristic is that cells should have the ability to adapt to the environment through mutation. In this case, we will need to have a genome which controls various features including the speed of movement, the sensitivity to chemoattractant, etc. This genome should be subject to mutations during the process of reproduction but also environmental stress. Mutation will cause cells to behave differently depending on the parameters of each feature.

A fourth requirement is that reproduction should occur based on the amount of nutrients in the environment. This means each location in the problem space will have a fitness value. Cells who are in unfit locations should have lower reproduction rates then cells in locations of higher fitness. As a result of reproduction based on the fitness of the environment, cells whose genomes are well suited to the environment should reproduce more often as they will more often be in fitter locations.

Finally, our bacteria must be subject to natural selection. The fittest individuals should survive and be allowed to reproduce.

5.2 Algorithm Description

Our algorithm implements the requirements of specified in the above section. First of all our algorithm is a population based SI approach with agents that we call cells. Each cell has a basic life cycle of composed of six processes; birth, movement, signalling, reproduction, mutation and finally death. In each iteration of the algorithm, the cells perform decision making and actions based on the status of their local environment.

The pseudo code description of the algorithm is found below as Algorithm 2 and the details are explained in the subsequent subsections.

5.2.1 Cell Attributes

In order for the cells to adapt to the environment, we propose that each cell has a genome that controls how the cell behaves at each time step. To do this, we suggest seven characteristics each having an associated mutation rate. Each characteristic has the ability to modify the exploration and exploitation of the algorithm.

- **Step Size** (Δ): This represents how far a cell can move in one time step. Cell's whose step size approaches the movement in the environment should be better at remaining within an optimum at each step. For example, environments with higher severity should require moving a greater distance between each time steps to stay in a optima. Cell's with a large step size would better exploit the environment by remaining moving along with the optima.

- **Viewing Radius** (λ): This specifies how far the cell can sense signals in environment. In some environments such as low severity, a larger viewing radius should increase the exploitation of the cells movement by providing a more accurate depiction of all the chemotactic signals in the environment.

- **Max Age** (c.maxage): Each cell at birth is given maximum amount of time to live. However, at each time step, the cell has a probability to survive based on its current age(c.age). Equation 5.1 is first proposed by Ramos in [21]

$$P_{survival} = \frac{c.maxage - c.age}{c.maxage} \tag{5.1}$$

  Mutation of this characteristic will change how the cells may explore or exploit the search space. As raising the max age has the effect of letting cells live longer

Algorithm 2: Microbiological Swarm Optimization

```
let f be our fitness function
let S be our search space
begin
    initialize cells in random position inside the search space $S$
    while not done do
        /* find max and min height */
        foreach Cell $c \in S$ do
            $c_{fitness} = f(c_{pos})$
            if $c_{fitness} > z_{max}$ then
                $z_{max} = c_{fitness}$
            if $c_{fitness} < z_{min}$ then
                $z_{min} = c_{fitness}$
        $\Delta_{max} = z_{max} - z_{min}$
        foreach Cell $c \in S$ do
            $rf = c_{fitness}/\Delta_{max}$
            ;    /* Calculate probabilities for signalling,
            reproduction, mutation */
            $P_{reproduction} = h(harshness, \frac{\Delta_i}{\Delta_{max}})$
            $P_{mutate} = h(harshness, 1 - \frac{\Delta_i}{\Delta_{max}})$
            $CA = \sigma \frac{\Delta_i}{\Delta_{max}}$
            /* Add Chemo Attractant to the environment */
            S.addCA(CA)
            /* Reproduce */
            if $random[0, 1] < P_{reproduction}$ then
                S.addCell($c.split(P_{mutate})$)
            /* Mutate due to environmental pressure */
            if $random[0, 1] < P_{mutate}$ then
                c.mutate()
            ;         /* Move via Equations(5.5, 5.3, 5.4) */
            calculate $\hat{a}, \hat{b}$
            $c_{pos} += \vec{v} = \Delta((\alpha\hat{a} + \overline{\beta\hat{b})\gamma N[0, 1]})$
            $c_{age} -= 1$
end
```

and explore the search space for more iterations. This effectively increases the radius of search for the colony, increasing the probability that new locations of high fitness may be discovered due exploration farther away from the center of the colony.

- **Sensitivity to chemoattractants** ($cell^{sensA}$): is the sensitivity to chemoattractants within the viewing radius. As this variable changes, there will greater or less sensitivity to attractants in the environment. Mutations of this variable will allow the cells to ignore chemotactic signals in very high severity environments as the chemotactic map is continually out of date. The opposite is also true in that in environments with low severity, the chemotactic signals in the environment will accurately denote the fitness of the locations. And so, cells with a high sensitivity will quickly move towards the area of high fitness and reproduce more frequently then those who do not.

- **Sensitivity to chemorepellents** ($cell^{sensToR}$): is the sensitivity to chemorepellents within the viewing radius. This has the opposite effect of chemoattractants and will serve to push the cells away from high density locations.

- **Randomness of Movement** (Γ): represents how random the movement of the cell is at it moves in a direction. In certain environments, it may be beneficial to only move in on direction instead of an exploratory random walk. In this case, a reduction to the randomness of movement will aid the cells to track the optima.

- **Birth Radius** ($cell^{BirthRadius}$): This is the euclidean distance away that a offspring can be born from its parent. Depending on the environment, this may also play an important role by affecting the where reproduction will occur. In environments of high severity, having a larger birth radius will allow for cells to be born in or slightly ahead of a moving optima.

Each characteristic plays an important role in how each cell will behave. We propose that the mutation of the genome will allow the cell to develop behaviours better suited to the environment.

We will now describe how each cell will decide to move at each iteration of the algorithm. For the purpose of our calculations please read overbar of the vector to be the normalize vector with magnitude of 1. We define the normalized vector $\overline{x}$ as

$$\overline{x} = \frac{x}{|x|}$$

$$where$$

$$|x| = \sqrt[2]{x_1^2 + x_2^2 + ...x_n^2}$$

5.2.2 Movement

Movement occurs in each iteration of the algorithm by each cell. At each time step t a cell will make a decision based on local information contained in the environment. The next position is found by

$$p_{t+1} = p_t + \hat{v} \tag{5.2}$$

Equation 5.2, where p is our current position $\hat{v}$ is the movement vector. In line with the characteristics we have specified in the previous section. The decision process and its vector are based on three main factors: the location and strength of chemoattractants, chemorepellents and a random movement. We represent each of these features as vectors giving a cell both a direction and magnitude away from its current position.

Inorder to simulate the attraction of cells to chemoattractants, we create a chemoattractant vector $\hat{a}$ that points towards the greatest strength of chemoattractants within the viewing radius (Equation 5.3).

$$\hat{a} = \overline{\sum^{\lambda_A}((\overline{A_i^{pos} - cell^{pos}})(A_i^{conc}))(cell^{sensA})} \tag{5.3}$$

This vector has the effect of pushing the cell towards the chemoattractants in its local environment. To create this vector, each chemoattractant A has a position that is used to create a difference vector between the cell and chemoattractant. This vector points from the cell to the location of the chemoattractant. Once this is done, the difference vector is normalized and scaled by the concentration of A to simulate the effect of the chemoattractant on the cell. This process is done for each chemoattractant in the viewing radius creating a resultant vector whose direction is towards the greatest amount of attractant sensed by the cell. Now having the direction of attraction, the final step is to normalize the vector and scale it by the cell's sensitivity to chemoattractants. This has the effect of allowing the cell's genome to decide how much effect chemoattractants will have on the final direction of the cell. Figure 5.1 depicts this situation. Here, the difference vectors labeled with green tips point from the cell to the chemoattractants. The direction of greatest attraction in the environment is the solid red vector and the viewing radius of the cell is the blue circle.

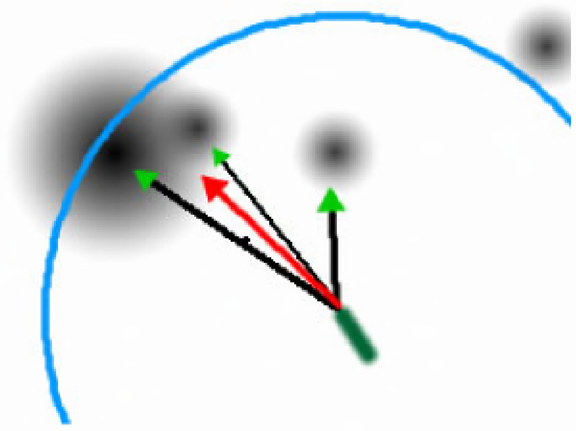


Figure 5.1: Chemoattractants effect on a cell

The second vector in the movement vector is the created through the chemorepellent process. Here we have slightly cheated as we assume that decision process of chemorepellents is based on the density of cells within an area. As a result, instead of having to place chemorepellents in the area, we implement this as the act of moving away from near cells when a density threshold is reached within the area. Much like the chemoattractant vector, here we calculate a difference vector between the cells and its neighbours and scale it by our threshold value as described in equation 5.4.

$$\hat{b} = \overline{\sum^{\lambda_R}((cell^{pos} - cell_i^{pos}))(cell^{sensToR})} \tag{5.4}$$

With equations 5.3 and 5.4 we can place them into the final movement vector 5.5

$$\vec{v} = \Delta(\overline{(\alpha\hat{a} + \beta\hat{b})\gamma N[0,1]}) \tag{5.5}$$

which adds our final features of the random movement through a normal distribution N[0,1] with a mean of zero and standard deviation of 1. This allows the normalized and scaled chemotactic signal vectors to be randomly adjusted. Once this is accomplished, we scale each vector by user defined constants(α, β, γ). Again, we allow the genome to decide how far the cell will move at each step by normalizing and scaling the movement vector based on the genome defined step size Δ.

5.2.3 Environmental Pressure

As cells in optimal regions of the search space should have higher reproduction rates, lower mutation rates and the ability to release more chemoattractants into the environment, we need a method to enforce a type of pressure from the environment. To do this, we use a simple ranking system to establish the relative rank of each cell against the other cells in the colony. This method was first seen in [21] and is described by Equation 3.3. The individual's rank is between zero(least fit) and 1 (the most fit). We augment this ranking system with a scaling function h that better simulates a real environment. The h function can be thought of as the harshness of the environment. In very harsh environments only a few cells will find enough food to reproduce while in less harsh environments the opposite is true. We use the relative rank combined with the environmental harshness to produce probabilities for reproduction and mutation The h function outlined in Equation 5.6

$$h(harshness, x) = \frac{e^{(harshness - x)}}{e^{harshness}} \tag{5.6}$$

scales the relative rank so that it is possible to give greater emphasis in different areas of the relative rank curve. That is, if a cell has a relative rank of 0.77 in an environment with harshness of 4, its scaled rank will be 0.4. Figure 5.2 demonstrates relative scaled rank values based on different values for the harshness.

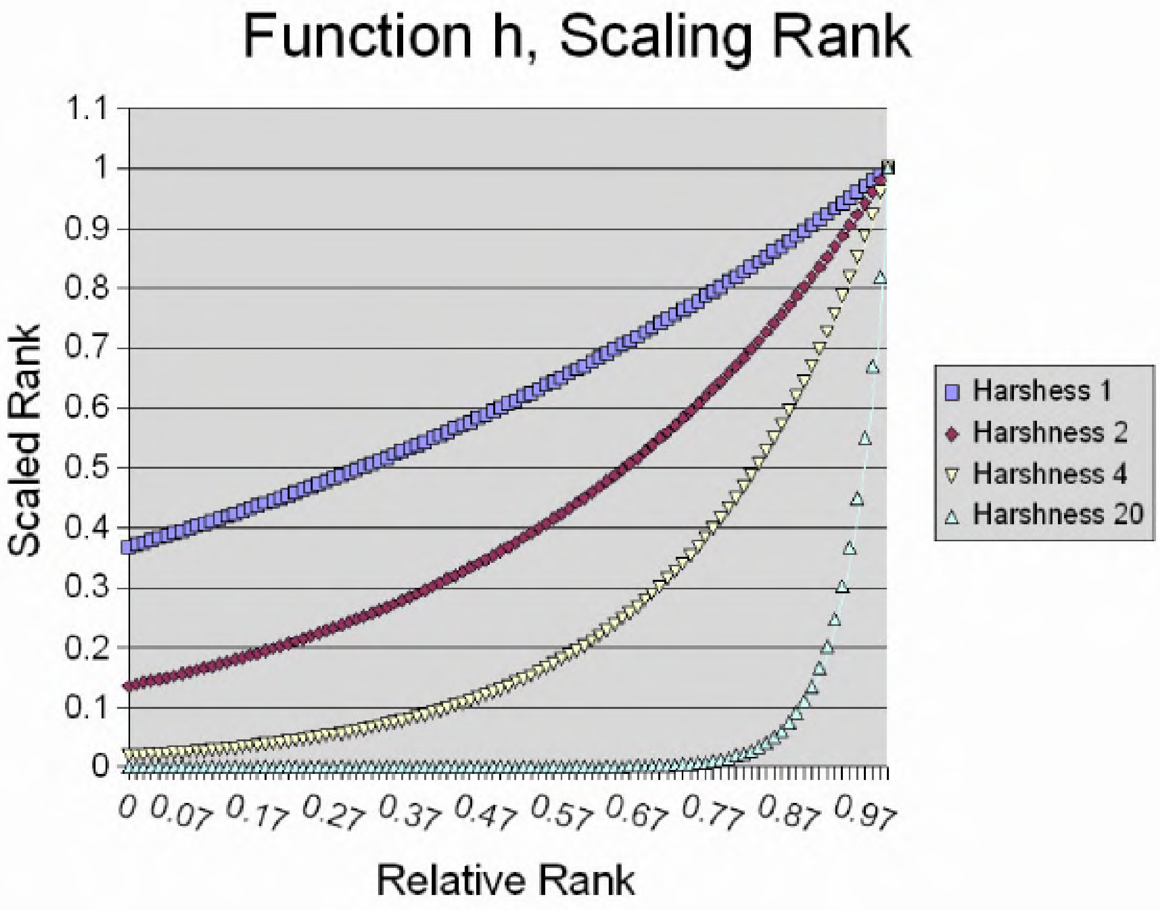


Figure 5.2: Probabilities values under the h function.

The function h has the interesting feature of modifying the exploration/exploitation of the algorithm. Higher values of the harshness will give greater reproductive success for cell in area's of higher fitness. This in turn results in less diversity of solutions as hence less exploration.

In the case of reproduction, we give higher probability of reproduction to cells that have higher ranks as seen in Equation 5.7.

$$P_{reproduction} = h(harshness, \frac{\Delta_i}{\Delta_{max}}) \tag{5.7}$$

This allows for the immediate exploitation of the environment based on rank. In high fitness areas we should see the highest birth rates as well as population. In the context of the optimization process, this allows the algorithm to focus its resources(agents) in areas of better quality.

On the other side of this token, we can use the relative rank described by Equation 3.3 to decide which cells should perform latent mutations due to environmental stress. We introduce this concept with the hope that this mutation process will diversify the genomes of the colony providing with greater adaptability but also specialization to the environment. To accomplish this, cells with lower fitness will have a higher probability to mutate at each time step. Here we again use the h function (Equation 5.6) to scale how frequent mutations will occur at different levels of the relative rank. The probability for mutation to occur at each time step is defined by Equation 5.8

$$P_{mutate} = h(harshness, 1 - \frac{\Delta_i}{\Delta_{max}}) \tag{5.8}$$

This mutation rate is inversely proportional to its rank and is important as it provides both adaptability but also stability to the colony by allowing for less mutation to occur to cells that are successfully finding areas of high fitness.

Finally, our environment plays an important role in the cell's ability to assess its current location by releasing chemoattractants at its current location. This was first seen in [7] and as the active deposition of pheromones by ants based on the quality of given movement. However, its application to dynamic optimization for optima tracking was made by [21]. We modify this mechanism slightly (Equation 5.9) such that the amount of chemoattractant placed into the environment is

$$CA = \sigma \frac{\Delta_i}{\Delta_{max}} \tag{5.9}$$

based on a constant amount scaled by the relative fitness of the cell. This allows cells to only release chemoattractants in the environment in locations that are of higher fitness. This should allow each cell to accurately inform other cells within a local environment of the quality of his position. The result here is that the chemotatic map created by the entire colony will reflect the fitness of the environment. Area's of high fitness should have the most chemoattractants and therefore have the greatest attractiveness to other cells.

5.2.4 Mutation

We integrate mutation into our algorithm with the hopes that our cells will develop traits which match the environment. For example, in an environment with a severity of 0.1, we would hope that the step of our cell would converge to 0.1. This may allow for a cell to better track the optima. To achieve proper mutation, instead of using a normalized Gaussian distribution for mutation probabilities as is the often the case for EC [1], we use a Cauchy Distribution which has less of its distribution in the center and more on its tails. This has been successfully applied towards PSO's with mutation characteristics as seen in [13]. This should give the algorithm greater mutation enabling faster adaptation to the environment. Our mutation operator (Equation 5.10) C[0,1], is probabilistically applied to each of the characteristic of the cell based on a individual characteristic's mutation rate.

$$cell^{char'} = cell^{char} + \delta C[0,1] \tag{5.10}$$

As we have discussed earlier, mutation occurs in our algorithm for two separate reasons. The first reason being through the process of reproduction which we will define soon. The second mutation will occur based on the relative fitness of the individual. In both cases, the h function and an associated harshness of the landscape will be used to decide if the mutation will occur.

5.2.5 Reproduction

The process of reproduction of the cells is reproduce at every time step depending on the relative rank. Those in locations of higher fitness will reproduce more frequently than those with locations with lower fitness. Furthermore, the location of birth is within a random radius of the current birth radius. This has the effect of ensuring that cells to not evaluate the exact same location on following iterations.

During the process of reproduction, as is the case in with real bacteria, the process is probabilistically subject to mutation. In the algorithm, the probability of the a mutation during reproduction is the same as probability that the cell will mutate described by Equation 5.8.

5.3 Comparison

Now that we have outlined the general framework of the algorithm, it is important to contrast it against the existing approaches to DO.

Our swarm is most similar to the SRS system that is a hybrid of Ant Colony Optimization and SI. The first reason our approaches are similar is that both swarms use the environment to store information. The SRS creates a pheromonal field in the environment by pheromone deposits by the ants. Our approach is similar as the cells deposit chemoattractants into the environment. In both cases information is stored in the environment and is used to attract the agents towards better areas of the search space.

Another similarity with the SRS system is that both systems use a reproduction operator. However, unlike the SRS, our approach allows cells to reproduce without the need of other cells to be within the immediate proximity. This allows our approach to immediately exploit new found regions of higher fitness.

A final similarity between the SRS system and our approach is the use of the a relative ranking system(Equation 3.3). In the case of the SRS, this ranking is directly tied into the reproduction and pheromone dropping procedures. Our approach is different as we scale the ranking and based on the h function(Equation 5.6) that represents and environment harshnesses. This vastly changes the ranking of each individual.

Of course, the major difference in our approach is the use of a genome and a simple mutation operator. Mutation of the genome allows for flexibility in a variety of environments. The result being that the swarm can adapt to the environments allowing for better tracking of optima.

Having now outlined the general framework of the algorithm its operation, we will discuss its design and implementation.

Chapter 6

Design and Implementation

The design and implementation of the our algorithm is described in this chapter. However, before we discuss the design of the system it is important to highlight an important design consideration.

6.1 Design Considerations

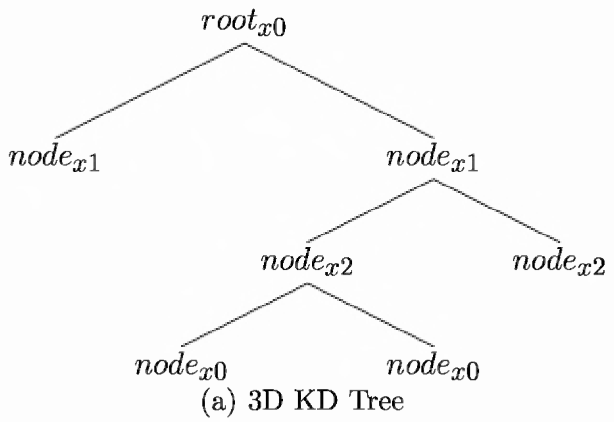
The implementation of the algorithm must take in many considerations in order to account for the multi-dimensionality of optimization problems. The key issue for our algorithm is the method that we keep track of our multidimensional chemoattractants. Because each cell, is required to look at the chemotactic environment to make a decisions on where to move, a search operation into the environment is required. And so, the implementation of a data structure containing the chemoattractractants has a significant impact on the complexity of the algorithm.

One possible implementation is using a NxN array which breaks up the environment into discrete pieces. One such implementation is [21] who implemented their system in a 2D NxN Array capable of holding a pheromone value. The strength of this approach is that local searches in 2D only require searching through the eight adjacent cells to a particular position. However, as dimensions increase, such searches become increasingly expensive to perform. For example, a local search in 50 dimensions is $8 * 50 = 400$ queries. The other weakness of a grid based implementation is that it restricts the possible positions that our agents can move to in continues value problems. This is because we must associated each position(i,j) with some area of the reals. Finally the third and possible greatest weakness is the space requirement to hold very large environments in memory. Each added dimension exponentially grows the problem space by the Equation 6.1

$$size = N^d \tag{6.1}$$

For example a problem with 32 locations per dimension with 50 dimensions has memory requirement of $32^{50} = 1.809251394 * 10^{75}$ array units. Clearly this is infeasible for real world applications.

An alternate method is to store chemoattractants in a multidimensional search data structure like a KD-Tree. Essentially, a KD-Tree is a binary tree where each level of the tree partitions a dimension of the search space. As you can see by Figure 6.1 we have three dimensions being $X0, X1, X2$. Each node is labeled with the dimesion's value that a new node would compare against. Now, as we traverse down the tree we



(a) 3D KD Tree

Figure 6.1: Multi-DimensionalKD-Tree (3D)

perform a comparison with a new dimension at each level of the tree. As with a binary tree the comparison identifies on what side of current node the new node should be. The traveral down the tree continues until an empty sub tree found where the new node is then inserted. A Java based insert method can be found in the Appendix A.1.1.

The performance of KD-Trees is impressive mimicking the classic binary tree performance of O(log_2N) for both insertion and and search [3]. Important to our discussion is the range query. This returns all items within a given range. This query is used every iterations for cells searching for the chemotactic signals with their viewing radius. The expected performance of the range query is O($(log_2N + F)$ with a worst case of O($d\dot{l}og_2N^{1-1/d} + F$) where d is the depth of the tree, N is the number of points in the tree and F is the number of points within our range [4].

An important note is that performance of the KD-Tree is based on a random insertions. In the worst case, a KD-Tree can degenerate to $O(n)$ performance for insertions and queries. As a result, certain steps to add randomness to the insertion process should be taken [23].

6.2 Design

The basic design of the system can be shown in the UML diagram of Figure 6.2. The diagram has been simplified in order to illustrate the overall functioning of the system.

The main package is the ca.log2n.gav.asrs.optimizer as it contains the Optimizer and Environment classes. The Optimizer class is the heart of the optimization process as it performs the main algorithm described in Algorithm 2. It has access to the supporting data structures such as KD-Trees that hold both the cells and the chemoattractants. The Environment class that is responsible for simulating a n-dimensional linear update environment and providing the fitness of a given position.

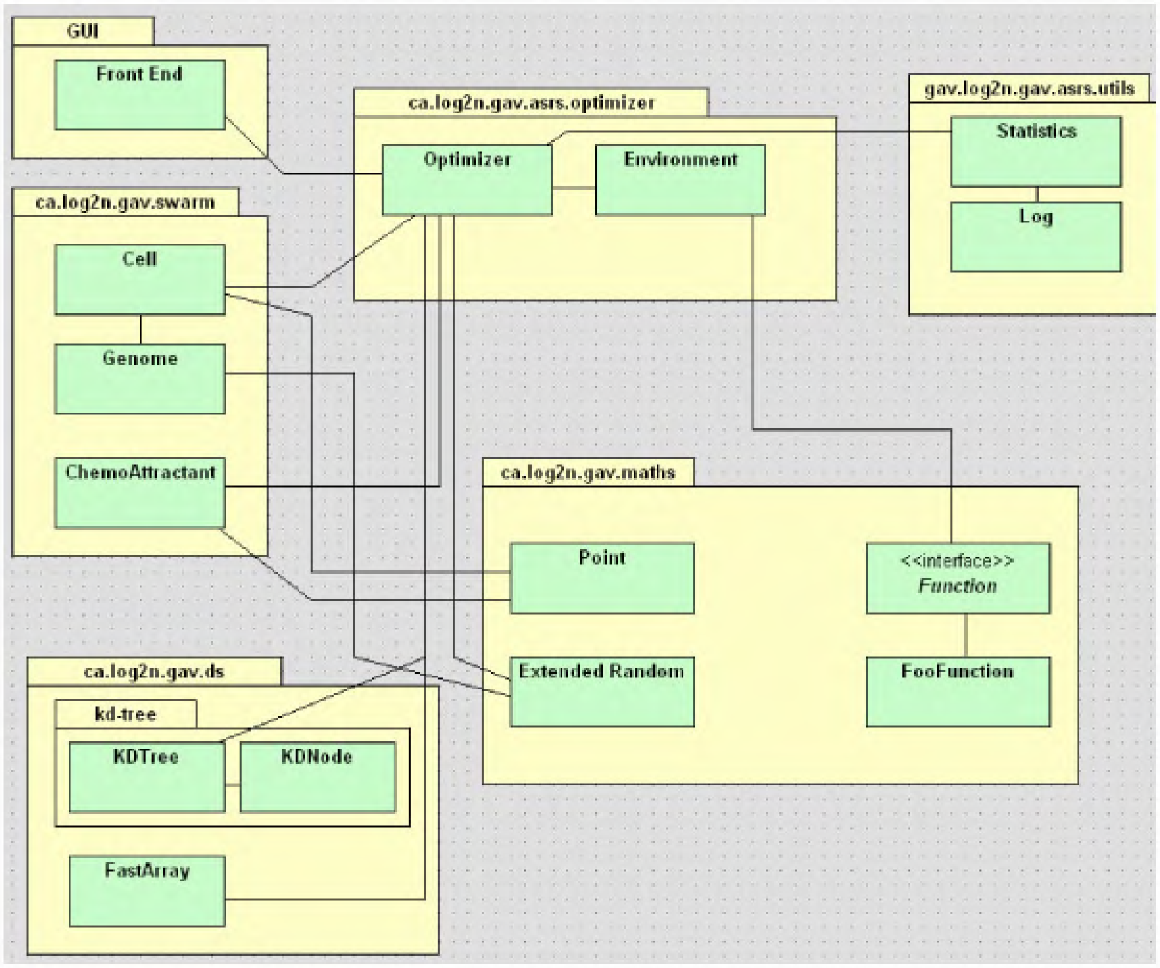


Figure 6.2: Simplified UML

Within the ca.log2n.gav.asrs.swarm package, are the cell, genome and chemoattractant classes. The Cell class is responsible to hold a current location and direction but also a genome. The genome class contains variables for each of the outlined characteristics but also the mutation operator that will randomly modify the genome based on the Cauchy Distribution. The Chemoattractant class defines a the characteristics of the chemoattractant in the environment(size, diffusion rate, ect).

The ca.log2n.gav.asrs.maths package contains the classes used for mathematical purposes. A very important class is the Point class found in this package. This class is a generalizable point for n-dimensions. This class contains many of the methods used for the addition,subtraction but also a host of vector type operations such as scaling and normalizing. The ExtendedRandom class is responsible for all of the random methods including one for the Cauchy Distribution. The final class that we discuss in this package is the Function Interface. This class defines an interface so that any function can quickly be added to the function list by simply implementing the interface. This is the function that we want to optimize during our optimization process. Designing at this level of abstraction is very powerful as it can allow users to define very complicated fitness functions. For example, during the later stages of the programming, we implemented a fitness function whose goal was to return how well our optimizer performed over a test with a given set of parameters. In other words, we were able to run the optimizer on itself to discover a set of good initial parameters.

The data structures supporting the system are the KDTree and the FastArray that are found in the package ca.log2n.gav.ds. These data structures are used to maintain the list of cells and chemoattractants in the search space.

Finally, the ca.log2n.gav.asrs.utils package contains the classes required to test the system. The Statistic class needs to be able to keep track of the information regarding the status of the cells at each iteration. The statistic groups the information into two sections being optimization progress, and genome status. For the optimization progress, the statistic class keeps track of the maximum, minimum, average, variance and standard deviation of both position and fitness. The statistic class also keeps track of the average, variance and standard deviation regarding the cells genome characteristics.

The log class has the role of writing status information collected by the Statistics class to file.

6.3 Implementation

Our algorithm was implemented in the Java Programming Language and is composed of approximately 10,000 lines of code which we wrote all components include the main algorithm, supporting data structures for n-dimensional operations, an environment simulator as well as a 3D visualizer and statistic capabilities. The algorithm is implemented to perform searches in n dimensional dynamic environments. The Visualizer is written as an applet using JOGL(Java Open GL) and can be see in Figure 6.3. The source code for the classes described above can be found in the appendix.

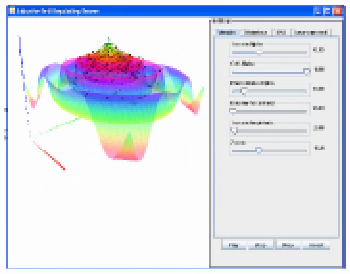


Figure 6.3: Optimization Vizualizer

As the Cauchy Probability Distribution is not implemented in the java.util.Random class, we implemented it in the ExtendedRandom class based on the geometric description found at [27].This class extends java.util.Random providing all the required random functionality required by the system. The implementation of KD-Tree is a generic point based KD-Tree that is described in the previous section. It implements the Iterator interface that uses the FastArray class. The FastArray is class provides very quick insert and delete operations on the assumption that the order of elements is not important to the user. When an Iterator is needed, the KD-Tree may check to see if the enumeration already has been performed. If not, then a FastArray is created with the contents of the tree. Otherwise, the Iterator simply returns the existing FastArray . This provides quick access to the objects within the KD-Tree without having to traverse the tree.

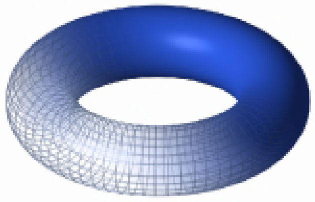


Figure 6.4: Toroidal Search Space.

In order to guard the KD-Tree against degeneration to $O(n)$ performance, we rebuild the tree at each iteration in an order based on two types of traversals. We create an insertion list by probabilistically performing a depth first or breadth first traveral of the tree. Once this is complete a the new tree is created from traversal list.

We implement the search space in the Environment Class as a n dimensional toroid(Figure 6.4)to minimize the effects of boundaries as described by [21]. This has the effect ensuring that the search process of agents can continue when the boundaries of a dimension has been crossed.

Chapter 7

Testing and Discussion

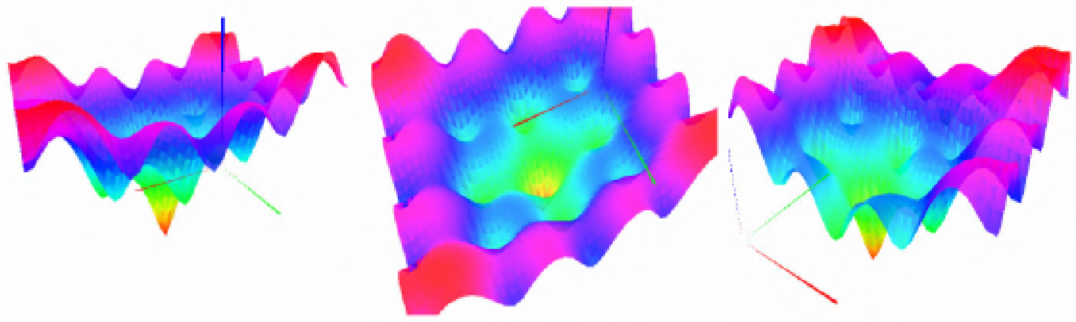
The purpose of our tests are to evaluate the performance of our swarming algorithm in dynamic environments as well as coming to an understanding if our cells mutation property was successful in adapting to the environment. Continuing the research of [21], who have both proposed state of the art algorithms in terms of performance and flexibility for DO, we utilize the same test in order to establish our relative performance. The test evaluates the tracking and flexibility of the algorithm in a dynamic multi-modal environment. It is important to note that for each test we used the same initial genome configuration. Any difference between the results are due the effect of the mutation operations given the environment

7.1 Ackley Function

The Ackley function (Equation 7.1 is a multi-modal function that has been used by Ramos [21] to gauge the quality of his SRS system.

$$f(\vec{X}) = -20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - a_i)^2}) - \exp(\frac{1}{n}\sum_{i=0}^{n}\cos(2\pi(x_i - a_i))) + 20 + e \quad (7.1)$$

Here we will attempt to minimize the function over the domain $-2 \le x_i \le 2$ for n = 2. As can been seen in figure 7.1, the minimum height of this function is zero. We also use the linear update function to update the environment at each time step. This linear function acts on the environment by pushing the landscape by a vector at each time step. The severity is calculated by $(magnitudeofmovement)/(sizeofenvironment)$.

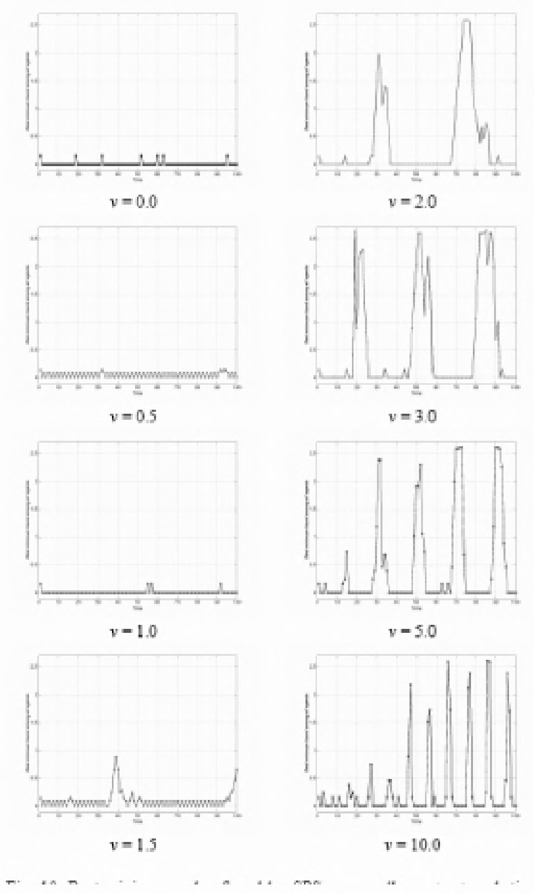


(a) $s = 0.1, T = 0$ (b) $s = 0.1, T = 2$ (c) $s = 0.1, T = 4$

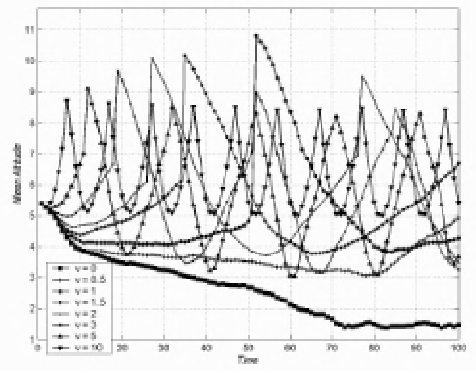
Figure 7.1: Ackley Function

Ramos defines the velocity v as the step size the environment takes compared to his ant habitat size (100X100) and so, the velocity is v = severity* 100; We perform a series of tests at v = 0, v=0.5, v = 1, v = 1.5, v=2, v = 3, v = 5, v=10, v=15, v=30 and v=40 with an update frequency uf =0 and linear dynamics; And compare against the best found and the average found. The results from the SRS are displayed in Figure 7.2 in order to compare our results. It is important to note that we run our test for about 1000 iterations, instead of the hundred that Ramos did. This in order to discover if our swarm is adapting the environment. We believe that we can assume that the Ramos's swarm will continue act in the same manner as there no mechanisms for mutation in his SRS. Our max population size for the test is set as 1000.

Figure 7.3 shows our results for velocities v=0. Our swarm was able to quickly located the minima and stay on it. We see stability of the genome until iteration 200, at which point we see mutation and a continual reduction in the average height which coincides with a lowering of the genome's average step rate. Essentially, the cells are learning that they do not have to move very far to be successful. The result of best



(a) Best Found



(b) Average

Figure 7.2: SRS on Ackley v=0.5, v = 1, v = 1.5, v=2, v = 3, v = 5, and v=10

found and the average are similar to those found by Ramos. We omit the results from speed test for velocities v=0.5,1 as we continue to see similar results to v=0.

Our results begin to be more interesting at velocity v=2 (Figure 7.4) with perfect tracking rates throughout the course of the run with a series of successful mutations at iteration 80 to 144 that lower the average height. Here our results begin to deviate from those found by Ramos as his swarm lost track of the optima twice where our swarm perfectly tracked the optima.

At v=5 (Figure 7.5) initially the swarm experiences the same oscillatory behaviour as the SRS demonstrated in its test. However, unlike the SRS, successful mutations occur at iteration 631 which result in good tracking until about iteration

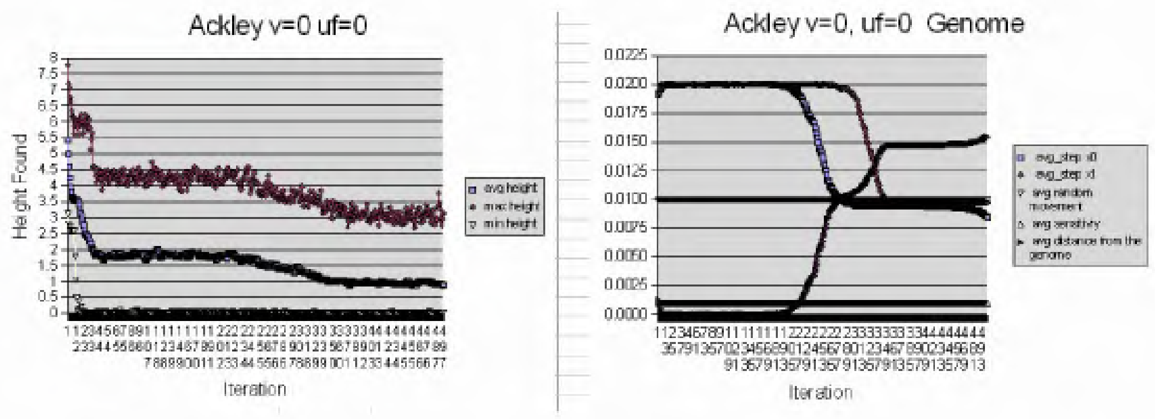


Figure 7.3: Our Results on Ackley v=0

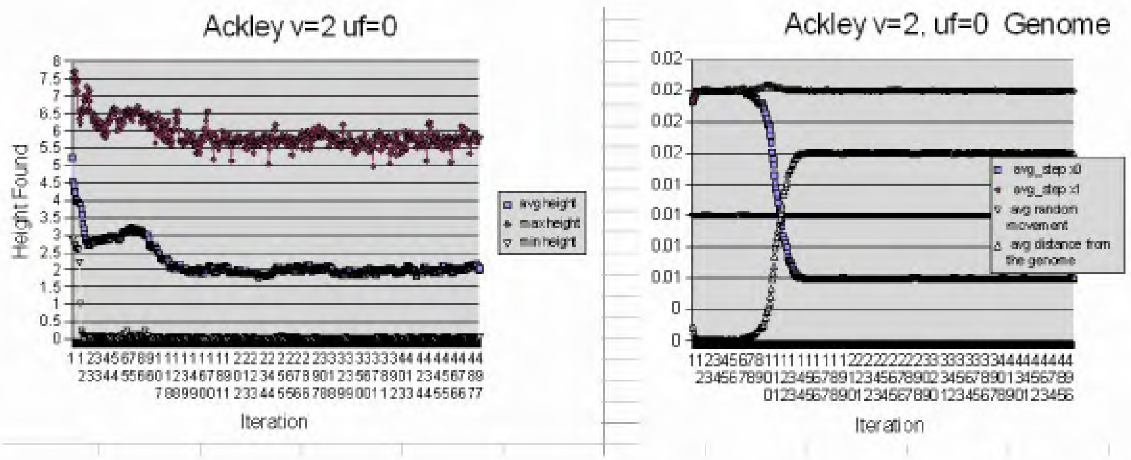


Figure 7.4: Our Results on Ackley v=2

800 where it becomes near perfect. The genome mutation records show changes occurring until iteration 813 where we see stabilization. This is echoed by the average height converging on a height of three for the last 200 iterations.

At v=10, (Figure 7.6) where the SRS was unable to perform any tracking, our swarm was able to mutate and successfully track the minima again showing the strength of our approach. We see slow and steady mutations along the length of the run with successful mutations occurring at iteration 500 and 900. Both series of mutations allowed for a substantial increase in the tracking rates.

We conducted a series of tests outside the scope of what has been performed on the SRS as our tracking was still much better then the SRS at the higher speeds.

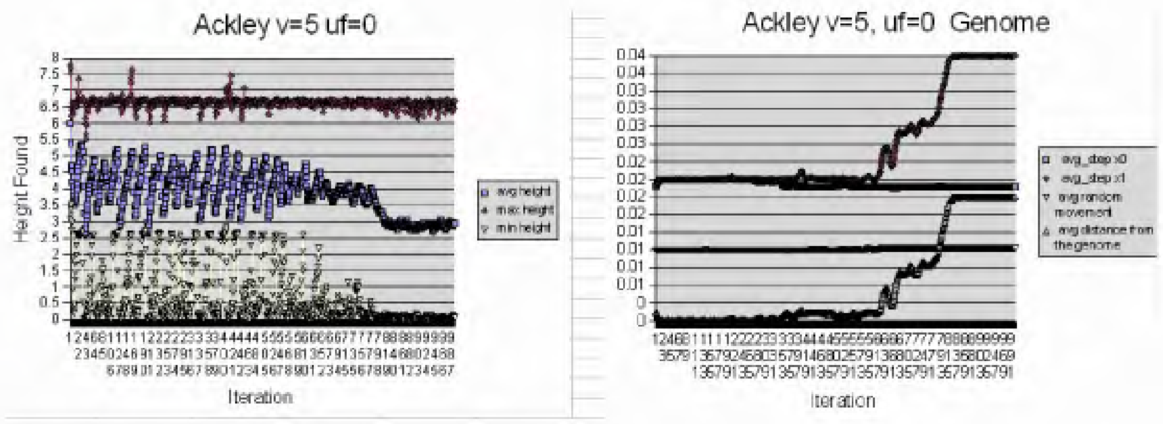


Figure 7.5: Our Results on Ackley v=5

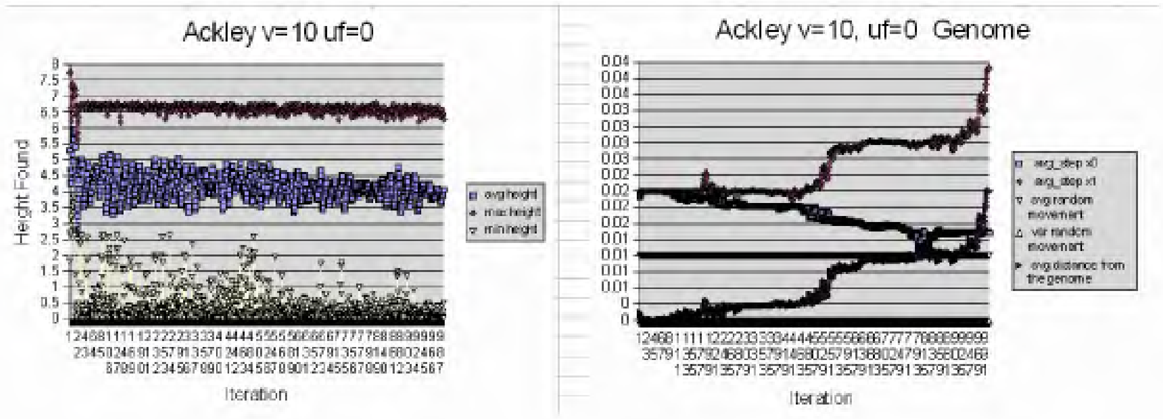


Figure 7.6: Our Results on Ackley v=10

We examine the tracking at v = 15, 30 and 40.

At v=15, (Figure 7.7) our swarm continues to show it adaptability with excellent tracking in the later portion of the run. We continued to see successful mutation that enabled the swarm to track the minima across the landscape. It is important to note that our swarm at v=15 is performing comparably if not better then the SRS at v=5. And so, our swarm is tracking optima moving three times as quickly across the environment.

At v=30, (Figure 7.8) the swarm continues to adapt to environment although with less success then before. We see a series of expanding intervals where the swarm appears to be tracking the optima only to loose it it later on. Strong evidence of this occurs between the iterations 568-631 and then 735-864. This is also echoed with a

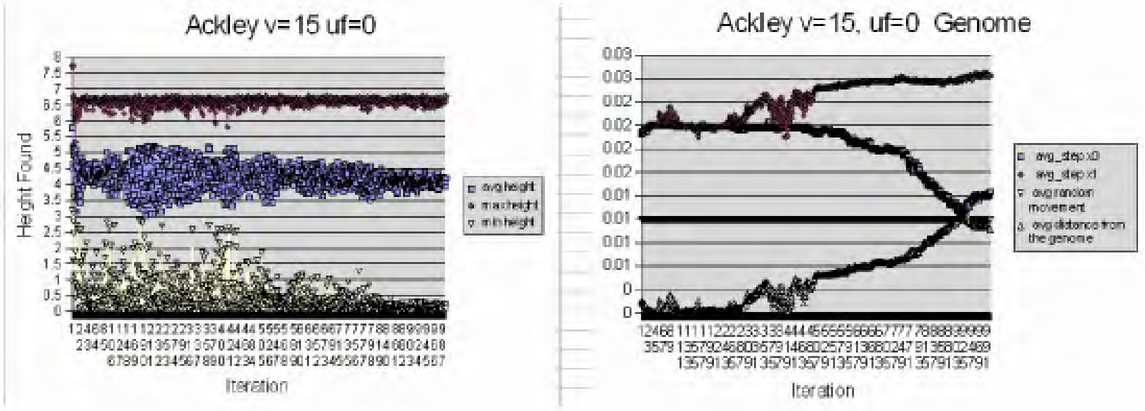


Figure 7.7: Our Results on Ackley v=15

contraction of the average height around four during those periods.

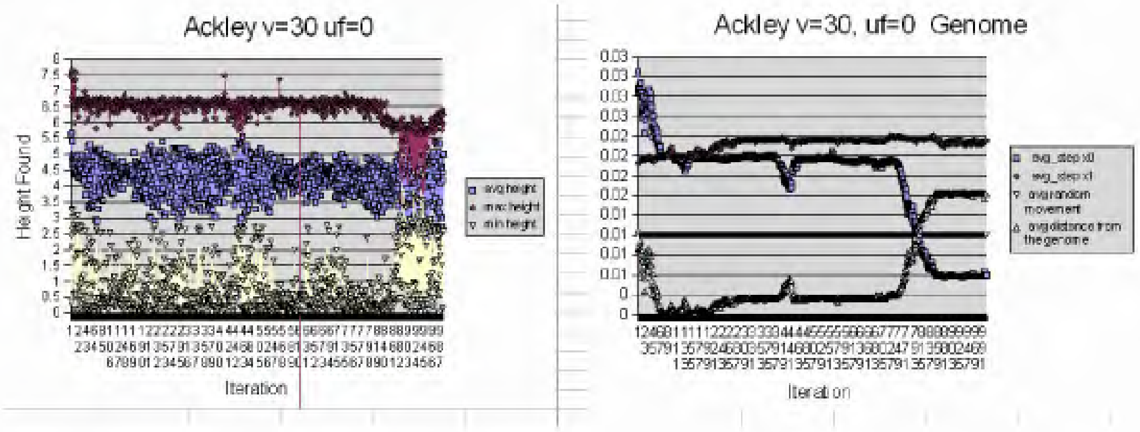


Figure 7.8: Our Results on Ackley v=30

Finally our last test at v=40, (Figure 7.9) our the swarm we are unable to discern the any tracking is occurring through the data. We do see a series of mutations around iteration 400 onward but we are unable to distinguish if the mutations are successful.

Figure 7.10 examines the progression through space of the swarm in an example of a v=5 where the swarm is successful in adapting to the environment. As we can see the initial swarm at T=0, we have an explosion around the global minimum. At T= 5 we see a wide plume trailing where cells are moving towards chemoattractant signals in the environment. This continues until T = 50 where we begin to see a small

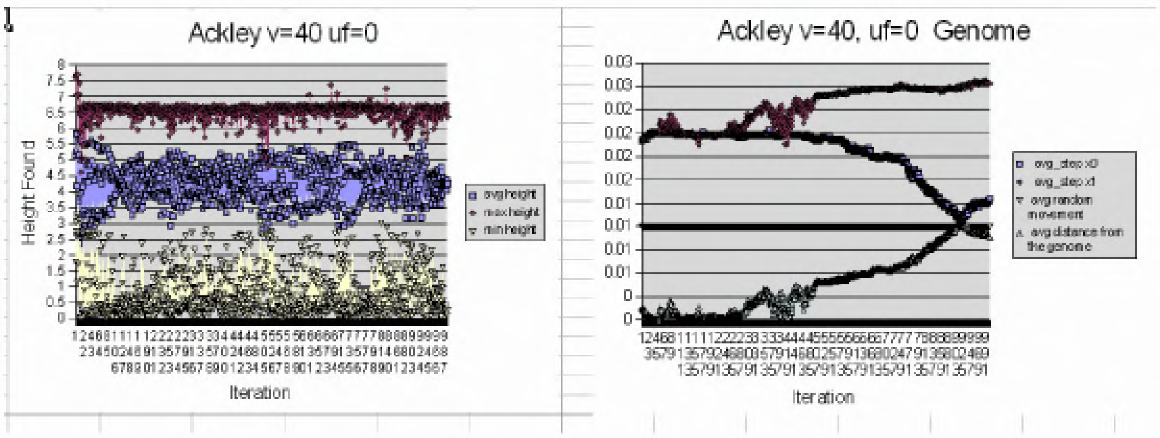
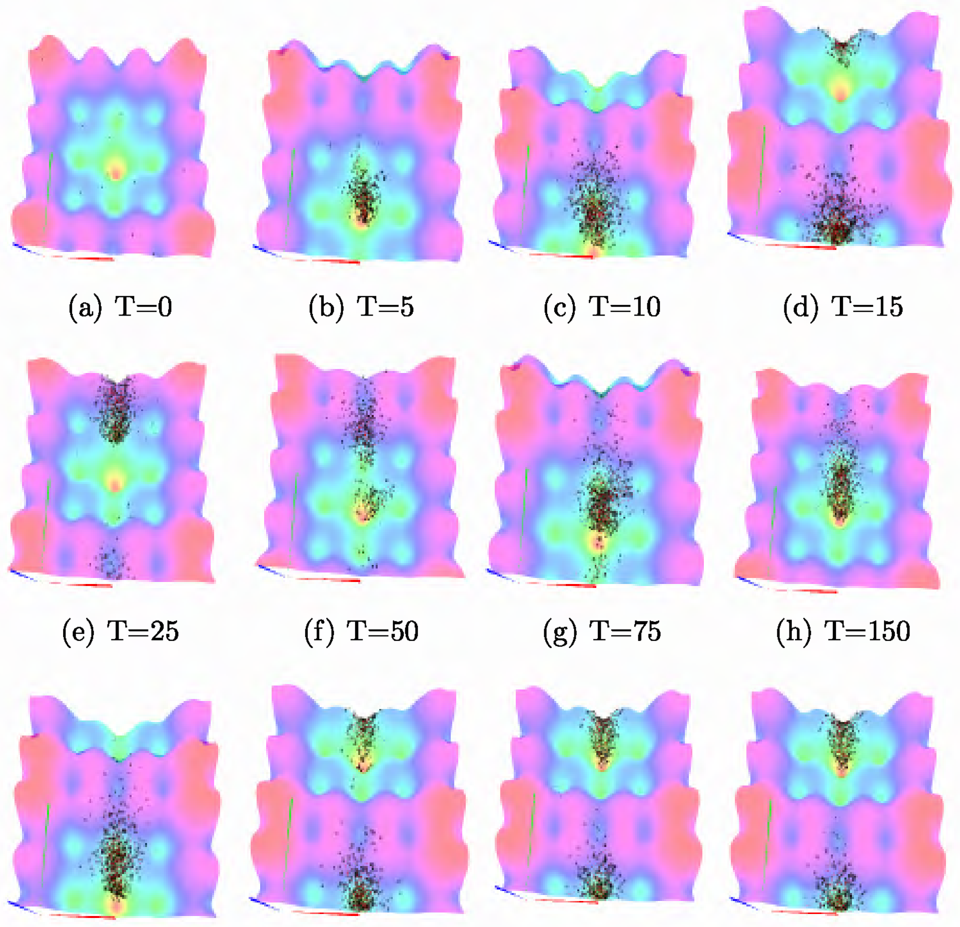


Figure 7.9: Our Results on Ackley v=40

group of cells finally relocating the minimum. Once this occurs, we notice that the plume following the minimum is much narrower.



(a) T=0 (b) T=5 (c) T=10 (d) T=15

(e) T=25 (f) T=50 (g) T=75 (h) T=150

(i) T=175 (j) T=225 (k) T=250 (l) T=500

Figure 7.10: Swarm Progression on Ackley v=5

7.2 Summery

We were quite please with the results achieved above as it validates our rational that characteristics of microbiological swarm can indeed be used in DO problems.

One key feature of the microbiological swarm is its adaptability to the environment. Adaptability plays an important role in the solutions to the Ackley function tests allowing for better tracking rates then the SRS system. This serves as an important proof of concept that it is possible to use the principles of microbiological evolution for the adaptability of SI algorithms. However, further research needs to be conducted on increasing this adaptability while maintaining some genetic stability in the colony. We often found that a high mutation rate had a destabilizing effect on the colony which produced no tracking in the environment. Unfortunately, too little mutation did not appear to change the quality of the tracking of optima within the scope of our tests. However, the mutation rates that we did choose were successful in demonstrating the strength of adaptation in DO.

Another key feature of the swarm that played an important role is the environmental pressure that served as the catalyst for change through both reproduction and mutation. Implemented as a combination of relative ranking and scaling function H(Equation 5.6). It allowed for the immediate exploitation of the environment through reproduction in high fitness areas. Furthermore, the long term effect of environmental pressure produced greater exploitation of the environment throughout the discovery of good traits for cells.

One our main concerns is the remarkable success of the algorithm in higher velocities may not be generalizable outside of linear dynamic functions. We believe that our results are partly due to the toroidal environment continually repeating the same path of across the landscape. This allowed for the reinforcement of cells who stayed within the path of the optima. In the case of environments with unpredictable movements, learning may occur at a slower rate as the cells would not be continually exposed to the path of the optima.

Chapter 8

Future Work and Conclusion

Future work could involve minimizing the effect of the toroidal borders by reducing the severity of the environment and the step size by a similar ratio. In a sense, this expands the toroid minimizing the effect of the boundaries. This should give those doing the research more conclusive results on the adaptability of our technique. Furthermore, more testing should be done on other multi-modal functions with different environment update mechanisms.

Another avenue of future research could be the use of self-adaptive methods[1], that allow for the mutation rates of the cells to be dynamically modified based on rate of progress of the optimization. Here, the algorithm would start with high rates of mutations and slowly increase or slow down the mutation rates based on the success of the tracking.

Finally, the future work should also focus on using the swarm for real world applications such as training of neural networks, scheduling tasks or perhaps as a form of game AI. In games, AI opponents often have many parameters that change their behaviour [22]. It would be possible to have our optimization algorithm tweak these parameters in order to provide a dynamic scaling of difficulty to human opponents. This could make games more engaging by not overwhelming or boring the player with game play that is either to hard or too easy.

In conclusion, dynamic environments problems are generally difficult to optimize as the solution space is in constant flux. As a result, the optimization algorithms must be capable of tracking optima through the search space that is continually changing. As microbiological organism live and perform a type of optimization know as foraging, their characteristics of self-organization, adaptation and the process of natural selection have shown to be effective in our implementation of a DO algorithm.

We compared the effectiveness of our algorithm against another state of the art algorithm in a series of tests on the multi-modal function Ackley. Our results was very impressive outperforming the other algorithm in all tests of difficulty. At lower speeds, our swarm performed comparably. However, as the speed increased, we were able to track optima moving at speeds three time greater then what had been previously achieved.

The main reason for the gain in performance was the mutation of the genome contained in each cell. Successful mutations allowed cells to better track the optima by the modification of their step size, viewing radius, maximum age, randomness of movement as well as their sensitivity to attractants and repellents in the environment. As the cells ability to locate and track the optima was increased through mutation, their likelihood of reproducing was also increased. This resulted in the entire swarm taking on successful characteristics that allowed it to better track moving optima through the search space. The Microbiological inspired swarm continually showed its adaptability in finding novel solutions to the dynamic optimization problem.

And so at the time of writing, with no known algorithms having performed DO using a Microbiological inspired swarm. It would appear that this work is an important and powerful proof of concept that has advanced the state of the art in DO.

Bibliography

[1] ANGELINE, P., ANGELINE, P., REYNOLDS, R., MCDONNELL, J., AND EBERHART, R. Tracking Extrema in Dynamic Environments. *Proceedings of the 6th International Conference on Evolutionary Programming 1213* (1997). 1, 6, 9, 34, 53

[2] BEN-JACOB, E. Bacterial self-organization: co-enhancement of complexification and adaptability in a dynamic environment. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences 361*, 1807 (2003), 1283–1312. 2, 3, 21, 23

[3] BENTLEY, J. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 39

[4] BENTLEY, J., AND STANAT, D. Analysis of Range Searches in Quad Trees. *Information Processing Letters 3*, 6 (1975), 170–173. 39

[5] BIERWIRTH, C., AND MATTFELD, D. Production Scheduling and Rescheduling with Genetic Algorithms. *Evolutionary Computation 7*, 1 (1999), 1–17. 20

[6] BLACKWELL, T. Swarms in dynamic environments. *Genetic and Evolutionary Computation Conf., Chicago, USA* (2003). 16, 17

[7] BONABEAU, E., DORIGO, M., AND THERAULAZ, G. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999. 4, 11, 12, 33

[8] BRANKE, J. Evolutionary approaches to dynamic optimization problems-updated survey. *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems* (2001), 27–30. 9, 16

[9] BRANKE, J., ET AL. *Evolutionary Optimization in Dynamic Environments.* Kluwer Academic Publishers, 2001. 9, 15

[10] CAMAZINE, S., DENEUBOURG, J., FRANKS, N., SNEYD, J., BONABEAU, E., AND THERAULAZ, G. *Self-Organization in Biological Systems.* Princeton University Press, 2003. 11, 23

[11] COBB, H. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. *NRL Memorandum Report 6760* (1990), 523–529. 20

[12] CRUTCHFIELD, J. Is anything ever new? Considering emergence. *Complexity: Metaphors, Models, and Reality* (1994), 515–537. 12

[13] Engelbrecht, A. *Fundamentals of Computational Swarm Intelligence*. Wiley, 2005. 5, 7, 8, 9, 12, 13, 15, 16, 34

[14] Goodrich, M., and Tamassia, R. *Algorithm design*. Wiley New York, 2002. 4

[15] Grefenstette, J. Genetic algorithms for changing environments. *Parallel Problem Solving from Nature 2* (1992), 137–144. 20

[16] Kennedy, J. Small worlds and mega-minds: effects of neighborhood topology onparticle swarm performance. *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on 3* (1999). 16

[17] Kennedy, J., and Mendes, R. Population structure and particle swarm performance. *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on 2* (2002). 16

[18] Li, X., Branke, J., and Blackwell, T. Particle swarm with speciation and adaptation in a dynamic environment. *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (2006), 51–58. 17

[19] Millonas, M. Swarms, phase transitions, and collective intelligence. *Artificial Life III* (1994). 12

[20] Passino, K. Biomimicry of bacterial foraging for distributed optimization andcontrol. *Control Systems Magazine, IEEE 22*, 3 (2002), 52–67. 1, 21, 22

[21] Ramos, V., Fernandes, C., and Rosa, A. Societal Implicit Memory and his Speed on Tracking Extrema in Dynamic Environments using Self-Regulatory Swarms. *Journal of Systems Architecture, Farooq M. and Menezes R.(Eds.), special issue on Nature Inspired Applied Systems, Elsevier, Summer* (2006). 1, 6, 7, 18, 19, 26, 31, 34, 37, 43, 44

[22] Rollings, A., and Morris, D. *Game Architecture and Design: A New Edition*. New Riders, 2004. 53

[23] Samet, H. *Foundations of Multidimensional and Metric Data Structures*. Elsevier, 2006. 39

[24] Stephens, C., Garcia Olmedo, I., Mora Vargas, J., and Waelbroeck, H. Self-adaptation in evolving systems. *Artif Life 4*, 2 (1998), 183–201. 20

[25] Sutton, A., Whitley, D., Lunacek, M., and Howe, A. PSO and multifunnel landscapes: how cooperation might limit exploration. *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (2006), 75–82. 16

[26] Vavak, F., Jukes, K., and Fogarty, T. Learning the local search range for genetic optimisation innonstationary environments. *Evolutionary Computation, 1997., IEEE International Conference on* (1997), 355–360. 20

[27] WEISSTEIN, ERIC W. "cauchy distribution." from mathworld–a wolfram web resource. 42

Appendix A

Source Code

Within this section is listed some of the core source files used to implement our system. Many files have been omitted due to their size. The complete source for this project can be found at http://gav.log2n.ca/research/

A.1 Data Structures Package(ca.log2n.gav.ds)

A.1.1 KDTree Class

```
package ca.log2n.gav.ds.kdtree;
import java.util.Iterator;
import ca.log2n.gav.ds.FastArray;
import ca.log2n.gav.maths.geometry.Point;
public class KDTree<T> {
  KDNode root;
  int size;
  // private ArrayList list;
  private FastArray<T> list;
  public KDTree() {
    root = null;
    size = 0;
    // list =new ArrayList();
    list = new FastArray<T>(200);
  }
  public void insert(Point point, T value) {
    // insert into array.
    list.add(value);
    if (root == null) {
      root = new KDNode(0, point, value);
      size++;
      return;
    }
    put(root, point, value);
    size++;
  }
  private void put(KDNode node, Point point, T value) {
```

```
  if (node.compareTo(point) > 0) // if the point is greater
      then our node
  // position
  {
    // move right
    if (node.getRight() != null)
      put(node.getRight(), point, value);
    else
      node.setRight(new KDNode<T>(getNextDiscriminator(node),
          point,
           value));
  } else if (node.compareTo(point) <= 0) {
    // move left
    if (node.getLeft() != null)
      put(node.getLeft(), point, value);
    else
      node.setLeft(new KDNode<T>(getNextDiscriminator(node),
          point,
           value));
  }
}
private int getNextDiscriminator(KDNode node) {
  if (node.discriminator + 1 >= node.getPoint().getDim())
    return 0;
  return node.discriminator + 1;
}
public Iterator<T> getRange(Point point, double radius) {
  // List<T> list = new LinkedList();
  FastArray<T> alist = new FastArray<T>(this.size);
  if (root != null)
    rangeSearch(root, point, radius, alist);
  return alist.iterator();
}
public FastArray<T> getRangeAsFastArray(Point point, double
    radius) {
  // List<T> list = new LinkedList();
  FastArray<T> alist = new FastArray<T>(this.size);
  if (root != null)
    rangeSearch(root, point, radius, list);
  return list;
}
private void rangeSearch(KDNode node, Point point, double
    radius,
     FastArray list) {
  // is node within range?
  if (radius > node.getPoint().distanceFrom(point))
    list.add(node.getValue());
  // check left and right trees.
  if (node.getLeft() != null)
    if (node.compareLeft(point, radius)) // nodes to left are
        within
      // range
      rangeSearch(node.getLeft(), point, radius, list);
  if (node.getRight() != null)
    if (node.compareRight(point, radius))
      rangeSearch(node.getRight(), point, radius, list);
}
public Iterator<T> BreadthFirstEnumeration() {
  FastArray<T> list = new FastArray<T>(this.size);
  if (this.size > 0)
    BreadthFirstTraversal(root, list);
  return list.iterator();
}
public Iterator<T> DepthFirstEnumeration() {
  FastArray<T> list = new FastArray<T>(this.size);
  if (this.size > 0)
    DepthFirstTraversal(root, list);
  return list.iterator();
}
private void DepthFirstTraversal(KDNode node, FastArray<T> list
    ) {
  if (node.getLeft() != null)
    DepthFirstTraversal(node.getLeft(), list);
  if (node.getRight() != null)
    DepthFirstTraversal(node.getRight(), list);
  list.add((T) node.getValue());
}
private void BreadthFirstTraversal(KDNode node, FastArray<T>
    list) {
  list.add((T) node.getValue());
  if (node.getLeft() != null)
    DepthFirstTraversal(node.getLeft(), list);
  if (node.getRight() != null)
    DepthFirstTraversal(node.getRight(), list);
}
public Iterator<T> getLastTraveral() {
  return list.iterator();
}
public T[] toArrayBreadthFirstTraversal() {
  FastArray<T> list = new FastArray<T>(this.size);
  if (this.size > 0)
    BreadthFirstTraversal(root, list);
  return (T[]) list.toArray();
}
```

```
  public T[] toArrayDepthFirstTraversal() {
    FastArray<T> list = new FastArray<T>(this.size);
    if (this.size > 0)
      DepthFirstTraversal(root, list);
    return (T[]) list.toArray();
  }
  public int size() {
    return size;
  }
}
```

A.1.2 KDNode Class

```
package ca.log2n.gav.ds.kdtree;
import ca.log2n.gav.maths.geometry.Point;
public class KDNode<T> {
  int discriminator; // is the dimension used for comparing
  Point point; // is an point describing the location of the node
  T value; // is the value of the node.
  KDNode left;
  KDNode right;
  final int LEFT;
  final int RIGHT;
  /*
   * KDNODE
   */
  /**
   * default constructor
   */
  public KDNode(int discriminator, Point point, T value) {
    this.discriminator = discriminator;
    this.point = point;
    this.value = value;
    left = null;
    right = null;
    LEFT = -1;
    RIGHT = 1;
  }
  /**
   * compares one node to another using the discriminant of the
      current node.
   */
  public int compareTo(Point point) {
    if (this.point.getDiscriminant(discriminator) < point
        .getDiscriminant(discriminator))
      return 1;
  if (this.point.getDiscriminant(discriminator) > point
      .getDiscriminant(discriminator))
    return -1;
  // not less then or greater must be equal
  return 0;
}
/**
 * compares a node within a range
 *
 * @param point
 *            the center of the point
 * @param radius
 *            of the node
 * @return true if the current node is within the radius of the
     point, false
 *         otherwise
 */
public int compareTo(Point point, double radius) {
  if (this.point.getDiscriminant(discriminator) + radius <
     point
      .getDiscriminant(discriminator))
    return 1;
  if (this.point.getDiscriminant(discriminator) + radius >
     point
      .getDiscriminant(discriminator))
    return -1;
  // not less then or greater must be equal
  return 0;
}
/**
 * checks to see if a point is within a give radius from this
    point
 *
 * @param point
 * @param radius
 * @return true is the point is within range of the radius to
    the right,
 *         false otherwise
 */
public boolean compareRight(Point point, double radius) {
  if (this.point.getDiscriminant(discriminator) + radius <
     point
      .getDiscriminant(discriminator))
    return false;
  return true;
}
/**
```

```
 * checks to see if a point is within a give radius from this
     point
 *
 * @param point
 * @param radius
 * @return true is the point is within range of the radius to
     the left ,
 *         false otherwise
 */
public boolean compareLeft(Point point, double radius) {
  if (this.point.getDiscriminant(discriminator) - radius >
      point
       .getDiscriminant(discriminator))
    return false;
  return true;
}
public void setDiscriminator(int discriminator) {
  this.discriminator = discriminator;
}
public int getDiscriminator() {
  return discriminator;
}
public Point getPoint() {
  return point;
}
public T getValue() {
  return (T) value;
}
public void setPoint(Point point) {
  this.point = point;
}
public void setValue(T value) {
  this.value = value;
}
public void setLeft(KDNode left) {
  this.left = left;
}
public KDNode getLeft() {
  return left;
}
public void setRight(KDNode right) {
  this.right = right;
}
public KDNode getRight() {
  return right;
}
public String toString() {
    return "Discriminator: " + this.discriminator + " Point: "
        + point.toString() + " Value: " + value.toString();
  }
}
```

A.1.3 FastArray Class

```
package ca.log2n.gav.ds;
import java.util.Iterator;
public class FastArray<T> {
  private T[] array;
  /**
   * represents the index of the last element in the array set to
       public
   * status so no method call is required.
   */
  public int last;
  public FastArray(int size) {
    array = (T[]) new Object[size];
    last = -1;
  }
  public T at(int index) {
    return array[index];
  }
  /**
   * \ simple add at tail
   *
   * @param obj
   */
  public void add(T obj) {
    last++;
    if (last < array.length) {
      array[last] = obj;
    } else {
      doubleArray();
      array[last] = obj;
    }
  }
  /**
   * nice and quick delete operator assumes ordering of array is
       not important
   * places elements at position last to position i essentially
       deleting i
   * from the array.
   *
   * @param i
```

```
 *               the cell you want to delete
 */
public void delete(int i) {
  if (i > -1 && i < last + 1) {
    array[i] = array[last];
    last--;
  } else {
    throw new IndexOutOfBoundsException("INDEX: " + i + "LAST:
       " + last);
  }
}
public void doubleArray() {
  T[] temp = (T[]) new Object[array.length * 2];
  for (int i = 0; i < array.length; i++)
    temp[i] = array[i];
  array = temp;
}
public int size() {
  return last + 1;
}
public Iterator iterator() {
  return new FastArrayIterator();
}
public T[] toArray() {
  return array;
}
/**
 * Iterator for the class.
 *
 * @author Gav
 *
 */
public class FastArrayIterator<T> implements Iterator {
  int in = -1;
  public int size() {
    return last + 1;
  }
  public boolean hasNext() {
    return in < last;
  }
  public void remove() {
    delete(in);
  }
  public T next() {
    return (T) at(++in);
  }
}
```

```
}
```

A.2 Optimizer Package(ca.log2n.gav.asrs.optimizer)

A.2.1 Optimizer Class

```
package ca.log2n.gav.asrs.optimizer;
import java.util.ArrayList;
import java.util.Iterator;
import ca.log2n.gav.asrs.swarm.Cell;
import ca.log2n.gav.asrs.swarm.Genome;
import ca.log2n.gav.asrs.swarm.ChemoAttractants;
import ca.log2n.gav.ds.kdtree.KDTree;
import ca.log2n.gav.maths.geometry.Point;
import ca.log2n.gav.maths.random.ExtendedRandom;
import ca.log2n.gav.utils.log.Log;
import ca.log2n.gav.utils.log.Statistics;
public class Optimizer {
  Log log;
  boolean login = true;
  Environment env;
  ExtendedRandom rand;
  int iterationCount;
  int maxIterations;
  // Data Structures.
  KDTree<Cell> cells;
  KDTree<ChemoAttractants> ChemoAttractants;
  // Opt Settings.
  int maxcells;
  int mincells;
  // function options
  boolean maximize;
  boolean active;
  boolean usingChemoAttractant;
  boolean usingBirths;
  boolean usingCellDensity;
  boolean usingMutations;
  boolean usingChemoAttractantAbsorbtion;
  boolean usingFixedIterations;
  boolean usingHarshness;
  boolean usingStats;
  Statistics stats;
  double dPheremoneScale;
  double dDensityScale;
  double movementScale;
  double harshness;
  // enviromnent stats.
  double dMaxHeight;
double dMinHeight;
double dDeltaHeight;
Point pAvgPosition;
double dAvg;
double dSum;
public Optimizer(Environment env, boolean maximize, int
   mincells,
    int maxcells) {
  this.env = env;
  // this.maximize = maximize; // / maximize or minimize??
  this.maximize = false;
  this.maxcells = maxcells;
  this.mincells = mincells;
  active = false;
  cells = new KDTree<Cell>();
  this.maxcells = maxcells;
  ChemoAttractants = new KDTree<ChemoAttractants>();
  // log = new Log();
  usingChemoAttractant = true;
  usingBirths = true;
  usingCellDensity = false;
  usingMutations = true;
  usingChemoAttractantAbsorbtion = false;
  usingFixedIterations = false;
  usingHarshness = true;
  usingStats = false;
  // stats = new Statistics(env, this, "trial 0");
  maxIterations = 400;
  dPheremoneScale = 1;
  dDensityScale = 1;
  movementScale = 1;
  harshness = .7;
  dMaxHeight = 0;
  dMinHeight = 0;
  dDeltaHeight = 0;
  pAvgPosition = new Point(env.getEnvironmentSize().getDim());
  dAvg = 0;
  dSum = 0;
  rand = new ExtendedRandom(System.nanoTime());
  initialize();
}
public void initialize(Genome g) {
  Point envSize = env.getEnvironmentSize();
  for (int i = 0; i < mincells + 1; i++) {
    Point randomPoint = new Point(envSize.getDim());
    for (int j = 0; j < env.getEnvironmentSize().getDim(); j++)
        {
            randomPoint.setDiscriminant(j, envSize.getDiscriminant(j)
                * rand.nextFloat());
          }
          Cell cell = new Cell(randomPoint, g);
          cell.name = i + "";
          initPoint = randomPoint;
          cells.insert(randomPoint, cell);
        }
      }
      Point initPoint;
      public void initialize() {
        initialize(new Genome());
      }
      public void update() {
        if (usingFixedIterations && maxIterations < iterationCount) {
          return;
        }
        iterationCount++;
        env.update();
        Point tempPosition = new Point(env.getEnvironmentSize().
           getDim());
        double iterationSum = 0.0;
        // update each cell position
        //
        // / looking at ChemoAttractants
        // -> cell absorbs pheremone from env.
        // / -> cell releases(relays) pheronome at area ( effect
           lessen
        // attraction to an area )
        // / looking at cell density.
        // -> avoid high areas??
        // -> reproduce or not. ( too much density implies no
           reproduction )
        // (effect rep probabilty)
        // having cells unable to move serves no purpose.
        // reproduce if conditions adequate.
        // -> mutation rate based on env.
        // -> lower fitness mutate more with greater severity.
        // updating ChemoAttractant
        // -> radius increases as time passes
        // -> strenght diminshes with time(based on radius)?
        // history of env.
        // if change drastic??
        // probability of death based on env? Not just Higher
           mutation rates?
        // REQUIRED
        Iterator<Cell> cellArray;
```

```
Iterator<ChemoAttractants> pheArray;
// FOR EACH ChemoAttractant GET THE LIST OF CELLS WITHIN VIEW
   .
// ADD SOME RANDOMNESS TO THE UPDATE SEQUENCE
// helps keep KD-Tree's balanced.
float ran = rand.nextFloat();
if (ran > 0.85) {
  cellArray = cells.BreadthFirstEnumeration();
  pheArray = ChemoAttractants.BreadthFirstEnumeration();
} else if (ran > 0.65) {
  cellArray = cells.DepthFirstEnumeration();
  pheArray = ChemoAttractants.DepthFirstEnumeration();
} else {
  cellArray = cells.getLastTraveral();
  pheArray = ChemoAttractants.getLastTraveral();
}
// CALCULATE FITNESS, and EXAMINE AREA FOR REPRODUCTION DUE
   TO DENSITY
int count = 0;
int id = 0;
while (cellArray.hasNext()) {
  double fitness;
  // get current cell
  Cell currCell = cellArray.next();
  // get its fitness
  fitness = env.fitnessAt(currCell.getCurrentLocation());
  currCell.setCurrentFitness(fitness);
  iterationSum += fitness; // used for stats.
  // ASSUME MIN AND MAX HEIGHT ARE THE VALUE OF THE FIRST
     CELL.
  if (count == 0) {
    this.dMaxHeight = env.fitnessAt(currCell.
       getCurrentLocation());
    this.dMinHeight = env.fitnessAt(currCell.
       getCurrentLocation());
    count++;
  }
  // do enviromnent check
  if (fitness > this.dMaxHeight)
    this.dMaxHeight = fitness;
  if (fitness < this.dMinHeight)
    this.dMinHeight = fitness;
}
dSum += iterationSum / (float) cells.size();
ArrayList<Cell> birthsArray = new ArrayList();
ArrayList<ChemoAttractants> newChemoAttractantDropArray =
   new ArrayList();
```

```
// / because the iterator in FastArray is implemented simply,
    getting
// another iterator is very inexpensive
cellArray = cells.getLastTraveral();
pheArray = ChemoAttractants.getLastTraveral();
// caculate the delta height(delta_max)
dDeltaHeight = this.dMaxHeight - this.dMinHeight;
while (cellArray.hasNext()) {
  // grab object we need during this iteration
  Cell currCell = cellArray.next();
  Genome currGenome = currCell.getGenome();
  Point movementVector = currCell.getMovementVector();
  Point currPosition = currCell.getCurrentLocation();
  // / the three invluences to our movement vector.
  Point pChemoAttractantVector = new Point(env.
      getEnvironmentSize()
       .getDim());
  Point pDensityVector = new Point(env.getEnvironmentSize().
      getDim());
  double rating;
  // IF WE ARE MAXIMIZING THE ENVIRONMENT or min - calculate
      the
  // rating for the cell.
  if (maximize)
    rating = (currCell.getCurrentFitness() - this.dMinHeight)
        / this.dDeltaHeight;
  else
    rating = (this.dMaxHeight - currCell.getCurrentFitness())
        / this.dDeltaHeight;
  // evaluate location and drop ChemoAttractant..
  if (rand.nextFloat() < rating) // ChemoAttractant
  {
    ChemoAttractants p = currCell.getChemoAttractantDrop();
    p.setCurrValue(p.getInitialValue() * rating);
    newChemoAttractantDropArray.add(p);
  }
  // construct chemoattractant vector
  if (usingChemoAttractant) {
    Iterator<ChemoAttractants> phe = ChemoAttractants.
        getRange(
          currCell.getCurrentLocation(), currCell.getGenome()
              .getViewingRadius());
    pChemoAttractantVector = new Point(env.getEnvironmentSize
        ()
         .getDim());
    while (phe.hasNext()) {
      ChemoAttractants p = phe.next();
      // get vector from cell to ChemoAttractant
      Point diff = p.getLocation().clone();
      diff.sub(currPosition);
      diff.normalize(); // bring back to 1
      diff.scale(p.getCurrValue()); // adjust scale
      // appropriately with
      // randomness
      pChemoAttractantVector.add(diff);
    }// end inner while
    pChemoAttractantVector.normalize();
    pChemoAttractantVector.scale(currCell.getGenome()
        .getChemoAttractantSensitivity());
    currCell.setPChemoAttractantVector(pChemoAttractantVector
       );// keeps
                                         // track
    // of last
    // ChemoAttractant
    // movement (
    // DOES NOT
    // INFLUENCE
    // MOVEMENT!!)
  }// end if there are more cells.
  // construct cell density vector( equivalent of
     chemorepellent.
  if (usingCellDensity) {
    Iterator<Cell> near = cells.getRange(currCell
        .getCurrentLocation(), currCell.getGenome()
        .getViewingRadius());
    Point diff;
    pDensityVector = new Point(env.getEnvironmentSize().
       getDim());
    while (near.hasNext()) {
      diff = currPosition.clone();
      diff.sub(near.next().getCurrentLocation());
      pDensityVector.add(diff);
    }
  }
  // update position
  pChemoAttractantVector.scale(dPheremoneScale);
  movementVector.add(pChemoAttractantVector);
  pDensityVector.scale(dDensityScale);
  movementVector.add(pDensityVector);
  // get gaussian movement
  currCell.moveGaussian(currCell.getGenome()
      .getRandomnessOfMovement());
  // scale and move
```

```
movementVector.scale((currCell.getGenome().getMovementStep
    ()));
currCell.setCurrentLocation(env.addPoints(currCell
    .getCurrentLocation(), currCell.getMovementVector()));
movementVector.scale(30);
if (usingBirths) {
  currCell.liveALittle();
  double probabilityOfLife = (currCell.
      getLifeStepsRemaining()/ currCell.getGenome().
      getLifeSteps());
  //double probabilityOfLife = rating;
  if (usingHarshness) {
    probabilityOfLife = harsh(harshness, probabilityOfLife)
        ;
  }
  if (rand.nextFloat() > probabilityOfLife) {
    // kill off cell.
    if (cells.size() > this.mincells) {
      if (usingBirths)
        currCell.setLifeStepsRemaining(0);
    }
  }
  double birthsPerCell = 1;
  // if (usingHarshness) {
  // birthsPerCell = (maxcells - cells.size())
  // / (double) cells.size();
  // }
  for (int i = 0; i < birthsPerCell; i++) {
    if (rand.nextFloat() < rating) // reproduce???
    {
      if (this.maxcells > cells.size() + birthsArray.size()
          ) {
        Cell offSpring = currCell.clone();
        // create new offspring
        // mutate offspring as required.
        // add to bithing array
        offSpring.setCurrentLocation(env.addPoints(
            offSpring.getCurrentLocation(), env
                .getRandomPoint(currGenome
                    .getDBirthRadius())));
        // mutate offspring
        if (usingMutations)
          if (rand.nextFloat() < harsh(10, 1 - rating)) //
              mutate
          // cell
          {
            offSpring.getGenome().mutate();
```

```
                }
              // add into birthArray
              birthsArray.add(offSpring);
            }
          }
        }
      }
      // mutate based on environmental stress
      if (usingMutations)
        if (rand.nextFloat() < harsh(4, 1 - rating)) // mutate
            cell
        {
          currGenome.mutate();
        }
      // if(login)
      // log.log("Before: "+currCell.getCurrentLocation() + "
         Moving : " +
      // movementVector);
      // currCell.setCurrentLocation(env.addPoints(currCell.
         getCurrentLocation(),
      // currCell.getMovementVector()));
      // if(login)
      // log.log("After: "+currCell.getCurrentLocation());
      //
      // tempPosition.add(currCell.getCurrentLocation());
    }
    if (usingStats)
      stats.collectStats();
    // log.log("/////////////////////////");
    // log.log("Cells: "+ this.cells.size() + " Births: " +
    // birthsArray.size());
    // log.log("Min: "+ this.dMinHeight+"," + this.env.
       getDMinFound());
    // log.log("Max: "+ this.dMaxHeight+","+this.env.getDMaxFound
       () );
    // log.log("Delta: "+ this.dDeltaHeight);
    // log.log("ChemoAttractant: " + this.ChemoAttractants.size()
        + " new:
    // +" +
    // newChemoAttractantDropArray.size());
    // GET READY FOR NEXT ITERATION
    cellArray = cells.getLastTraveral();
    //
    // MOVEMENT VECTOR OF EACH CELL MUST BE SET TO ZERO
    // DELETE ANY DEAD CELLS
    //
    KDTree<Cell> tempTree = new KDTree();
    while (cellArray.hasNext()) {
      Cell currCell = cellArray.next();
      if (currCell.getLifeStepsRemaining() > 0)
        tempTree.insert(currCell.getCurrentLocation(), currCell);
    }
    for (int i = 0; i < birthsArray.size(); i++)
      tempTree.insert(birthsArray.get(i).getCurrentLocation(),
          birthsArray.get(i));
    cells = tempTree;
    pheArray = ChemoAttractants.getLastTraveral();
    KDTree<ChemoAttractants> tempChemoAttractants = new KDTree();
    while (pheArray.hasNext()) {
      ChemoAttractants p = pheArray.next();
      if (p.getCurrValue() > p.getValueLimit()) {
        p.update();
        tempChemoAttractants.insert(p.getLocation(), p);
      }
    }
    for (int i = 0; i < newChemoAttractantDropArray.size(); i++)
       {
      tempChemoAttractants.insert(newChemoAttractantDropArray.get
          (i)
          .getLocation(), newChemoAttractantDropArray.get(i));
    }
    ChemoAttractants = tempChemoAttractants;
    // update average positions
    pAvgPosition.scale(0); // reset Avg point
    tempPosition.scale(1 / (float) cells.size()); // get avg
    pAvgPosition.add(tempPosition);
  }
  public void doIteration(int n) {
    iterationCount = 0;
    this.usingFixedIterations = true;
    this.maxIterations = n;
    while (this.hasMoreIterations())
      update();
  }
  public Iterator getCells() {
    return cells.getLastTraveral();
  }
  public Iterator getChemoAttractant() {
    return ChemoAttractants.getLastTraveral();
  }
  public boolean isActive() {
    return active;
  }
  public void performStep() {
    update();
    active = false;
  }
  public void resetOpt() {
    cells = new KDTree<Cell>();
    ChemoAttractants = new KDTree<ChemoAttractants>();
    // log = new Log();
    rand = new ExtendedRandom(System.currentTimeMillis());
    initialize();
    iterationCount = 0;
    dSum = 0;
  }
  public void resetOpt(Genome g) {
    cells = new KDTree<Cell>();
    ChemoAttractants = new KDTree<ChemoAttractants>();
    // log = new Log();
    rand = new ExtendedRandom(System.nanoTime());
    initialize(g);
    iterationCount = 0;
    dSum = 0;
    // stats = new Statistics(env, this, "trial 0");
  }
  public void setActive(boolean active) {
    this.active = active;
  }
  public boolean getActive() {
    return this.active;
  }
  public void updateCells() {
  }
  public Point createRandomMovement(Cell cell) {
    Point point;
    if (cell.getGenome().getRandomnessOfMovement() > rand.
        nextDouble()) {
      point = env.getRandomPoint(1);
      point.normalize();
      return point;
    }
    Point p = new Point(env.getEnvironmentSize().getDim());
    return p;
  }
  public double getAvg() {
    return dSum / (float) iterationCount;
  }
  public double getDeltaHeight() {
    return dDeltaHeight;
  }
```

```
public void setDeltaHeight(double deltaHeight) {
  dDeltaHeight = deltaHeight;
}
public double getMaxHeight() {
  return dMaxHeight;
}
public void setMaxHeight(double maxHeight) {
  dMaxHeight = maxHeight;
}
public double getMinHeight() {
  return dMinHeight;
}
public void setMinHeight(double minHeight) {
  dMinHeight = minHeight;
}
public int getIterationCount() {
  return iterationCount;
}
public void setIterationCount(int iterationCount) {
  this.iterationCount = iterationCount;
}
public boolean hasMoreIterations() {
  if (usingFixedIterations && maxIterations > iterationCount)
    return true;
  return false;
}
public int getMaxcells() {
  return maxcells;
}
public void setMaxcells(int maxcells) {
  this.maxcells = maxcells;
}
public boolean isMaximize() {
  return maximize;
}
public void setMaximize(boolean maximize) {
  this.maximize = maximize;
}
public int getMincells() {
  return mincells;
}
public void setMincells(int mincells) {
  this.mincells = mincells;
}
public boolean isUsingBirths() {
  return usingBirths;
}
```

```
public void setUsingBirths(boolean usingBirths) {
  this.usingBirths = usingBirths;
}
public boolean isUsingCellDensity() {
  return usingCellDensity;
}
public void setUsingCellDensity(boolean usingCellDensity) {
  this.usingCellDensity = usingCellDensity;
}
public boolean isUsingChemoAttractant() {
  return usingChemoAttractant;
}
public void setUsingChemoAttractant(boolean
    usingChemoAttractant) {
  this.usingChemoAttractant = usingChemoAttractant;
}
public boolean isUsingMutations() {
  return usingMutations;
}
public void setUsingMutations(boolean usingMutations) {
  this.usingMutations = usingMutations;
}
public Genome getInitialGenome() {
  return new Genome();
}
public boolean isUsingChemoAttractantAbsorbtion() {
  return usingChemoAttractantAbsorbtion;
}
public void setUsingChemoAttractantAbsorbtion(
    boolean usingChemoAttractantAbsorbtion) {
  this.usingChemoAttractantAbsorbtion =
      usingChemoAttractantAbsorbtion;
}
public int getCellCount() {
  return cells.size();
}
public int getChemoAttractantCount() {
  return ChemoAttractants.size();
}
public Point getPAvgPosition() {
  return pAvgPosition;
}
public void setPAvgPosition(Point avgPosition) {
  pAvgPosition = avgPosition;
}
public int getMaxIterations() {
  return maxIterations;
  }
  public void setMaxIterations(int maxIterations) {
    this.maxIterations = maxIterations;
  }
  public boolean isUsingFixedIterations() {
    return usingFixedIterations;
  }
  public void setUsingFixedIterations(boolean
      usingFixedIterations) {
    this.usingFixedIterations = usingFixedIterations;
  }
  public double harsh(double scale, double prob) {
    return Math.exp(scale * prob) / Math.exp(scale);
  }
  public double Invharsh(double scale, double prob) {
    return scale * Math.log(prob * 100) / Math.log(100);
  }
}
```

A.2.2 Environment Class

```
package ca.log2n.gav.asrs.optimizer;
import java.util.Iterator;
import java.util.Random;
import java.util.Scanner;
import java.util.regex.Pattern;
import ca.log2n.gav.ds.kdtree.KDNode;
import ca.log2n.gav.ds.kdtree.KDTree;
import ca.log2n.gav.maths.functions.Function;
import ca.log2n.gav.maths.geometry.Point;
public class Environment {
  Function fitnessFunction;
  double dMaxFound;
  double dMinFound;
  Point movementVector;
  Point translationVector;
  Point environmentSize;
  Point randomMovementVector;
  Point tempPoint;
  Random rand;
  KDTree<EnvironmentModifier> environmentModifiers;
  KDTree cells;
  double dSeverity;
  double dRandomnessOfEnv;
  double dRandomnessOfMovementVector;
  int iUpdateFrequency;
int count;
/**
 * default constructor
 *
 * @param environmentSize
 *              point with the dimensions of the env from the
   origin
 * @param FitnessFunction
 *              a function to evalutate any give location within
   the env.
 * @param cells
 *              a tree containing the cells in the environment.
   ?? NEEDED?
 */
public Environment(Point environmentSize, Function
   FitnessFunction /**
                                       * ,
                                       * KDTree
                                       * cells*
                                       */
) {
  this.environmentSize = environmentSize;
  this.fitnessFunction = FitnessFunction;
  movementVector = new Point(environmentSize.getDim());
  translationVector = new Point(environmentSize.getDim());
  randomMovementVector = new Point(environmentSize.getDim());
  dRandomnessOfMovementVector = 0.0;
  dRandomnessOfEnv = 0.01;
  environmentModifiers = new KDTree<EnvironmentModifier>();
  // this.cells = cells;
  dSeverity = 1;
  iUpdateFrequency = 0;
  count = 0;
  rand = new Random(System.currentTimeMillis());
  dMaxFound = -99999999999d;
  dMinFound = 999999999999d;
}
public double fitnessAt(Point point) {
  // calc fitness from raw env
  tempPoint = this.addPoints(point, translationVector);
  // add point to translationVector
  double fitness = fitnessFunction.evaluate(tempPoint);
  // scale env due to modifiers.
  // this is ugly and needs to be changed.
  if (environmentModifiers.size() > 0) {
    Iterator it = environmentModifiers.BreadthFirstEnumeration
       ();
```

```
      double effect = 0.0;
      while (it.hasNext()) {
        effect += ((EnvironmentModifier) (it.next()))
            .calculateEffect(point);
      }
      // return fitness + fitness*effect;
      fitness *= effect;
    }
    if (fitness > this.dMaxFound)
      dMaxFound = fitness;
    if (fitness < this.dMinFound)
      dMinFound = fitness;
    return fitness;
  }
  public void update() {
    if (count < iUpdateFrequency) {
      count++;
      return;
    } else {
      count = 0;
    }
    // update where we are moving the environment
    translationVector = addPoints(translationVector,
       movementVector);
    if (this.dRandomnessOfEnv > rand.nextFloat()) {
      if (rand.nextDouble() < this.dRandomnessOfEnv) {
        Point center = this.getRandomPoint(1);
        double radius = rand.nextDouble() / 100;
        double scale = 0.5;
        double severity = 3.2;
        this.environmentModifiers
            .insert(center, new EnvironmentModifier(center,
               radius,
                scale, severity));
      }
    }
    translationVector = addPoints(translationVector,
        this.randomMovementVector);
    KDTree tmpTree = new KDTree();
    Iterator it = this.environmentModifiers.
       BreadthFirstEnumeration();
    while (it.hasNext()) {
      // update point
      // add into new tree
      EnvironmentModifier m = (EnvironmentModifier) it.next();
      Point tmpPoint = m.center;
    tmpPoint = addPoints(tmpPoint, this.translationVector); //
       update
                                  // point
    m.center = tmpPoint;
    tmpTree.insert(tmpPoint, m); // insert into tree
  }
  environmentModifiers = tmpTree;
}
/**
 * Adds points together and ensures the result is within the
    range of the
 * search space
 *
 * @param point1
 * @param point2
 * @return
 */
public Point addPoints(Point point1, Point point2) {
  Point newPoint = point1.clone();
  newPoint.add(point2);
  for (int i = 0; i < environmentSize.getDim(); i++) {
    if (newPoint.getDiscriminant(i) > environmentSize
        .getDiscriminant(i))
      newPoint.setDiscriminant(i, newPoint.getDiscriminant(i)
          - environmentSize.getDiscriminant(i));
    else if (newPoint.getDiscriminant(i) < 0)
      newPoint.setDiscriminant(i, newPoint.getDiscriminant(i)
          + environmentSize.getDiscriminant(i));
  }
  return newPoint;
}
public void addEnvironmentModifier(Point center, double radius,
    double scale, double severity) {
  this.environmentModifiers.insert(center, new
     EnvironmentModifier(
      center, radius, scale, severity));
}
/**
 * sets the dimension of the environment.
 *
 * @param size
 */
public void setEnvironmentSize(Point size) {
  this.environmentSize = size;
  this.fitnessFunction.setDimensions(size);
}
/**
```

```
 * sets the dimension of the environment.
 *
 * @param size
 */
public void setEnvironmentSize(String text) {
  this.environmentSize.setDiscriminants(text);
}
public Point getEnvironmentSize() {
  return this.environmentSize;
}
public void setFitnessFunction(Function function) {
  this.fitnessFunction = function;
  this.environmentSize = function.getDimensions();
}
public Function getFitnessFunction() {
  return this.fitnessFunction;
}
/**
 * sets the amont of movement per update in the environment
 *
 * @param point
 *            the amount of movement
 */
public void setMovementVector(Point point) {
  this.movementVector = point;
}
/**
 * sets the amont of movement per update in the environment
 *
 * @param point
 *            the amount of movement
 */
public void setMovementVector(String text) {
  this.movementVector.setDiscriminants(text);
}
/**
 * gets the amount of movement per update to the environment.
 *
 * @return
 */
public Point getMovementVector() {
  return this.movementVector;
}
public void setTranslationVector(Point point) {
  this.translationVector = point;
}
public Point getTranslationVector() {
    return this.translationVector;
  }
  /*
   * double dRandomnessOfEnv; double dSeverity; double
   * dRandomessOfMovementVector;
   */
  public void setRandomnessOfEnv(double var) {
    dRandomnessOfEnv = var;
  }
  public double getdRandomnessOfEnv() {
    return dRandomnessOfEnv;
  }
  public void setSeverity(double var) {
    dSeverity = var;
  }
  public double getdSeverity() {
    return dSeverity;
  }
  public void setRandomessofMovementVector(double var) {
    dRandomnessOfMovementVector = var;
  }
  public double getdRandomessofMovementVector() {
    return dRandomnessOfMovementVector;
  }
  public int getUpdateFrequency() {
    return iUpdateFrequency;
  }
  public void setUpdateFrequency(int updateFrequency) {
    this.iUpdateFrequency = updateFrequency;
  }
  public Point getRandomPoint(double scale) {
    Point n = new Point(this.environmentSize.getDim());
    for (int i = 0; i < n.getDim(); i++) {
      n.setDiscriminant(i, environmentSize.getDiscriminant(i)
          * rand.nextDouble() * scale);
      if (rand.nextBoolean())
        n.setDiscriminant(i, environmentSize.getDiscriminant(i) *
            -1);
    }
    return n;
  }
  /**
   * resets the environment movement and translation, randomness
      to zero.
   *
   */
  public void resetEnv() {
    this.movementVector = new Point(0, 0);
    this.translationVector = new Point(0, 0);
    this.dRandomnessOfMovementVector = 0;
    this.environmentModifiers = new KDTree<EnvironmentModifier>()
        ;
  }
  public double getDMaxFound() {
    return dMaxFound;
  }
  public void setDMaxFound(double maxFound) {
    dMaxFound = maxFound;
  }
  public double getDMinFound() {
    return dMinFound;
  }
  public void setDMinFound(double minFound) {
    dMinFound = minFound;
  }
}
```

A.3 Swarm Package(ca.log2n.gav.asrs.swarm)

A.3.1 Cell Class

```
package ca.log2n.gav.asrs.swarm;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;
import ca.log2n.gav.maths.geometry.Point;
import ca.log2n.gav.maths.random.ExtendedRandom;
public class Cell {
  double dCurrFitness;
  Point pCurrLocation;
  Point pMovementVector;
  Point pOldMovementVector;
  Point pChemoAttractantVector;
  Point pOldVector;
  ExtendedRandom rand;
  public String name;
  double iLifeStepsRemaining;
  Genome genome;
  ArrayList ChemoAttractantList;
  /**
   * constructs default cell for 2d movement with a default
     genome.
   *
   */
  public Cell() {
    this.dCurrFitness = 0;
    this.pCurrLocation = new Point(0, 0);
    this.pMovementVector = new Point(0, 0);
    this.genome = new Genome();
    this.iLifeStepsRemaining = genome.getLifeSteps();
    this.rand = new ExtendedRandom(System.nanoTime());
    pOldMovementVector = new Point(0, 0);
    pChemoAttractantVector = new Point(0, 0);
  }
  /**
   * Constructor for a given point with a given Genome.
   *
   * @param currLocation
   * @param genome
   */
  public Cell(Point currLocation, Genome genome) {
    this.dCurrFitness = 0;
    this.pMovementVector = new Point(0, 0);
    pCurrLocation = currLocation;
    this.genome = genome;
    this.iLifeStepsRemaining = genome.getLifeSteps();
    this.rand = new ExtendedRandom(System.nanoTime());
    this.rand.setB(0.2);
    pChemoAttractantVector = new Point(0, 0);
  }
  public Cell(Point currLocation, Genome genome, ExtendedRandom
     rand) {
    this.dCurrFitness = 0;
    this.pMovementVector = new Point(0, 0);
    pCurrLocation = currLocation;
    this.genome = genome;
    this.iLifeStepsRemaining = genome.getLifeSteps();
    this.rand = rand;
    pChemoAttractantVector = new Point(0, 0);
  }
  /**
   * STORES A FITNESS VALUE... DOES NOT RE-EVALUTAT THE FITNESS
     FROM THE
   * ENVIROMNENT!!!!!!!
   *
   * @param val
   *            the currnet fitness
   */
  public void setCurrentFitness(double val) {
    this.dCurrFitness = val;
  }
  public double getCurrentFitness() {
    return this.dCurrFitness;
  }
  public void setCurrentLocation(Point val) {
    this.pCurrLocation = val;
  }
  public Point getCurrentLocation() {
    return this.pCurrLocation;
  }
  public void setMovementVector(Point val) {
    this.pMovementVector = val;
  }
  public Point getMovementVector() {
    return this.pMovementVector;
  }
  public void resetMovementVector() {
    this.pMovementVector.reset();
  }
  /**
   *
  * @return the cells genome.
  */
public Genome getGenome() {
  return genome;
}
public void setGenome(Genome genome) {
  this.genome = genome;
}
public ArrayList getChemoAttractantList() {
  return ChemoAttractantList;
}
public void setChemoAttractantList(ArrayList
   ChemoAttractantList) {
  this.ChemoAttractantList = ChemoAttractantList;
}
public void addChemoAttractant(ChemoAttractants p) {
  this.ChemoAttractantList.add(p);
}
public Iterator getChemoAttractantIterator() {
  return this.ChemoAttractantList.iterator();
}
public ChemoAttractants getChemoAttractantDrop() {
  return new ChemoAttractants(this.pCurrLocation.clone(),
     genome
       .getChemoAttractantRadius(), genome
       .getChemoAttractantDropRate(), genome
       .getChemoAttractantRemovalLimit(), genome
       .getChemoAttractantExpansionRate());
}
public double getLifeStepsRemaining() {
  return iLifeStepsRemaining;
}
public void setLifeStepsRemaining(double lifeStepsRemaining) {
  iLifeStepsRemaining = lifeStepsRemaining;
}
public void liveALittle() {
  this.iLifeStepsRemaining--;
}
public Point getOldMovementVector() {
  return pOldMovementVector;
}
public void setOldMovementVector(Point oldMovementVector) {
  pOldMovementVector = oldMovementVector;
}
public Cell clone() {
  return new Cell(this.pCurrLocation.clone(), genome.clone());
}
```

```
  /**
   * performs a movement according to a gaussian normal
       distrbution for each
   * dimension.
   *
   */
  public void moveGaussian() {
    moveGaussian(1.0);
  }
  /**
   * performs a movement according to a gaussian normal
       distrbution for each
   * dimension.
   *
   */
  public void moveGaussian(double scale) {
    for (int i = 0; i < this.pMovementVector.getDim(); i++)
      pMovementVector.setDiscriminant(i, pMovementVector
          .getDiscriminant(i)
          + rand.nextGaussian() * scale);
    pMovementVector.normalize();
  }
  public Point getPOldMovementVector() {
    return pOldMovementVector;
  }
  public void setPOldMovementVector(Point oldMovementVector) {
    pOldMovementVector = oldMovementVector;
  }
  public Point getPOldVector() {
    return pOldVector;
  }
  public void setPOldVector(Point oldVector) {
    pOldVector = oldVector;
  }
  public Point getPChemoAttractantVector() {
    return pChemoAttractantVector;
  }
  public void setPChemoAttractantVector(Point
      ChemoAttractantVector) {
    pChemoAttractantVector = ChemoAttractantVector;
  }
}
```

A.3.2 Genome Class

```
package ca.log2n.gav.asrs.swarm;
```

```
import java.util.Random;
import ca.log2n.gav.maths.functions.ProbabilityFunction;
import ca.log2n.gav.maths.geometry.Point;
import ca.log2n.gav.maths.random.ExtendedRandom;
public class Genome {
  // TO THE GENOME
  // ChemoAttractant SETTINGS
  /**
   * controls how much ChemoAttractant is to be dropped for in an
       optimal
   * condition
   */
  double dChemoAttractantDropRate;
  double dMutationRate_ChemoAttractantDropRate;
  /**
   * controls how fast our ChemoAttractant is expand.
   */
  double dChemoAttractantExpansionRate;
  double dMutationRate_ChemoAttractantExpansionRate;
  /**
   * controls the level when a ChemoAttractant should be removed
   */
  double dChemoAttractantRemovalLimit;
  double dMutationRate_ChemoAttractantRemovalLimit;
  /**
   * controls how much a cell will absob from its environment;
   */
  double dChemoAttractantAbsortionRate;
  double dMutationRate_ChemoAttractantAbsortionRate;
  /**
   * controls how much a cell will absob from its environment;
   */
  double dChemoAttractantRadius;
  double dMutationRate_ChemoAttractantRadius;
  /**
   * controls how sensitive the cell is to ChemoAttractants
      around it.
   */
  double dChemoAttractantSensitivity;
  double dMutationRate_ChemoAttractantSensitivity;
  // REPRODUCTION
  /**
   * controls how likely a cell is to reproduce given the cell
      density in its
   * viewing radius.
   */
  double dCellDensity_ReproductionRate;
double dMutationRate_CellDensity_ReproductionRate;
/**
 * controls how far a offspring can be born from its parent.
 */
double dBirthRadius;
double dMutationRate_BirthRadius;
// LIFE SPAN
/**
 * controls how long a cell is likely to live.. how many steps.
 */
double iLifeSteps;
double dMutationRate_iLifeSteps;
// MOVEMENT
/**
 * The Randomness of a Cells Movement
 */
double dRandomnessOfMovement;
double dMutationRate_RandomnessOfMovement;
ProbabilityFunction functionRandomMovement;
/**
 * The basic amount of a movement for any direction
 */
Point pMovementStep;
double dMutationRate_MovementStep;
/**
 * How far a cell can see.
 */
double dViewingRadius;
double dMutationRate_ViewingRadius;
public Genome() {
  dChemoAttractantDropRate = 10; // 0
  dMutationRate_ChemoAttractantDropRate = 0.00000000; // 1
  dChemoAttractantExpansionRate = 0.5; // 2
  dMutationRate_ChemoAttractantExpansionRate = 0.0;// 3
  dChemoAttractantRemovalLimit = 2.01; // 4
  dMutationRate_ChemoAttractantRemovalLimit = 0.00; // 5
  dChemoAttractantAbsortionRate = 0.00; // 6
  dChemoAttractantRadius = 1; // 7
  dMutationRate_ChemoAttractantRadius = 0.000000; // 8
  dMutationRate_ChemoAttractantAbsortionRate = 0.001;// 9
  dChemoAttractantSensitivity = 0.001; // 10
  dMutationRate_ChemoAttractantSensitivity = 0.0001; // 11
  dCellDensity_ReproductionRate = 8; // 12
  dMutationRate_CellDensity_ReproductionRate = 0.0;// 13
  dBirthRadius = 0.01; // 14
  dMutationRate_BirthRadius = 0.001; // 15
  iLifeSteps = 5; // 16
  dMutationRate_iLifeSteps = 0.001; // 17
  dRandomnessOfMovement = 0.01; // 18
  dMutationRate_RandomnessOfMovement = 0.001; // 19
  this.functionRandomMovement = new ProbabilityFunction();
  pMovementStep = new Point(0.020, 0.020); // 21
  dMutationRate_MovementStep = 0.01; // 22
  dViewingRadius = 0.10; // 23
  dMutationRate_ViewingRadius = 0.001; // 24
}
/**
 * @param ChemoAttractantDropRate
 * @param mutationRate_ChemoAttractantDropRate
 * @param ChemoAttractantExpansionRate
 * @param mutationRate_ChemoAttractantExpansionRate
 * @param ChemoAttractantRemovalLimit
 * @param mutationRate_ChemoAttractantRemovalLimit
 * @param ChemoAttractantAbsortionRate
 * @param mutationRate_ChemoAttractantAbsortionRate
 * @param ChemoAttractantRadius
 * @param mutationRate_ChemoAttractantRadius
 * @param ChemoAttractantSensitivity
 * @param mutationRate_ChemoAttractantSensitivity
 * @param cellDensity_ReproductionRate
 * @param mutationRate_CellDensity_ReproductionRate
 * @param birthRadius
 * @param mutationRate_BirthRadius
 * @param lifeSteps
 * @param mutationRate_iLifeSteps
 * @param randomnessOfMovement
 * @param mutationRate_RandomnessOfMovement
 * @param functionRandomMovement
 * @param movementStep
 * @param mutationRate_MovementStep
 * @param viewingRadius
 * @param mutationRate_ViewingRadius
 */
public Genome(double ChemoAttractantDropRate,
    double mutationRate_ChemoAttractantDropRate,
    double ChemoAttractantExpansionRate,
    double mutationRate_ChemoAttractantExpansionRate,
    double ChemoAttractantRemovalLimit,
    double mutationRate_ChemoAttractantRemovalLimit,
    double ChemoAttractantAbsortionRate,
    double mutationRate_ChemoAttractantAbsortionRate,
    double ChemoAttractantRadius,
    double mutationRate_ChemoAttractantRadius,
    double ChemoAttractantSensitivity,
            double mutationRate_ChemoAttractantSensitivity ,
            double cellDensity_ReproductionRate ,
            double mutationRate_CellDensity_ReproductionRate ,
            double birthRadius , double mutationRate_BirthRadius ,
            double lifeSteps , double mutationRate_iLifeSteps ,
            double randomnessOfMovement ,
            double mutationRate_RandomnessOfMovement ,
            ProbabilityFunction functionRandomMovement , Point
                movementStep ,
            double mutationRate_MovementStep , double viewingRadius ,
            double mutationRate_ViewingRadius) {
        super();
        dChemoAttractantDropRate = ChemoAttractantDropRate;
        dMutationRate_ChemoAttractantDropRate =
            mutationRate_ChemoAttractantDropRate;
        dChemoAttractantExpansionRate = ChemoAttractantExpansionRate;
        dMutationRate_ChemoAttractantExpansionRate =
            mutationRate_ChemoAttractantExpansionRate;
        dChemoAttractantRemovalLimit = ChemoAttractantRemovalLimit;
        dMutationRate_ChemoAttractantRemovalLimit =
            mutationRate_ChemoAttractantRemovalLimit;
        dChemoAttractantAbsortionRate = ChemoAttractantAbsortionRate;
        dMutationRate_ChemoAttractantAbsortionRate =
            mutationRate_ChemoAttractantAbsortionRate;
        dChemoAttractantRadius = ChemoAttractantRadius;
        dMutationRate_ChemoAttractantRadius =
            mutationRate_ChemoAttractantRadius;
        dChemoAttractantSensitivity = ChemoAttractantSensitivity;
        dMutationRate_ChemoAttractantSensitivity =
            mutationRate_ChemoAttractantSensitivity;
        dCellDensity_ReproductionRate = cellDensity_ReproductionRate;
        dMutationRate_CellDensity_ReproductionRate =
            mutationRate_CellDensity_ReproductionRate;
        dBirthRadius = birthRadius;
        dMutationRate_BirthRadius = mutationRate_BirthRadius;
        iLifeSteps = lifeSteps;
        dMutationRate_iLifeSteps = mutationRate_iLifeSteps;
        dRandomnessOfMovement = randomnessOfMovement;
        dMutationRate_RandomnessOfMovement =
            mutationRate_RandomnessOfMovement;
        this.functionRandomMovement = functionRandomMovement;
        pMovementStep = movementStep;
        dMutationRate_MovementStep = mutationRate_MovementStep;
        dViewingRadius = viewingRadius;
        dMutationRate_ViewingRadius = mutationRate_ViewingRadius;
    }
    public void mutate() {
```

```
    ExtendedRandom rand = new ExtendedRandom(System.nanoTime());
    if (rand.nextDouble() < this.dChemoAttractantDropRate)
      setChemoAttractantDropRate(dChemoAttractantDropRate + .01
          * rand.nextCauchyZeroOne());
    if (rand.nextDouble() < this.
        dMutationRate_ChemoAttractantSensitivity)
      setChemoAttractantSensitivity(this.
          dChemoAttractantSensitivity
          + 0.001 * rand.nextCauchyZeroOne());
    if (rand.nextDouble() < this.
        dMutationRate_CellDensity_ReproductionRate)
      setCellDensity_ReproductionRate(this.
          dCellDensity_ReproductionRate
          + this.dCellDensity_ReproductionRate
          * rand.nextCauchyZeroOne());
    if (rand.nextDouble() < this.dMutationRate_BirthRadius)
      setDBirthRadius(this.dBirthRadius + 0.01 * rand.
          nextCauchyZeroOne());
    if (rand.nextDouble() < this.dMutationRate_iLifeSteps)
      setLifeSteps(this.iLifeSteps + this.iLifeSteps
          * rand.nextCauchyZeroOne());
    if (rand.nextDouble() < this.
        dMutationRate_RandomnessOfMovement)
      setRandomnessOfMovement(Math.abs(this.dRandomnessOfMovement
          + .01
          * rand.nextCauchyZeroOne()));
    if (rand.nextDouble() < this.dMutationRate_ViewingRadius)
      setViewingRadius(this.dViewingRadius + this.dViewingRadius
          * rand.nextCauchyZeroOne());
    for (int i = 0; i < pMovementStep.getDim(); i++)
      if (rand.nextDouble() < dMutationRate_MovementStep) {
        double r = pMovementStep.getDiscriminant(i) + .1
            * rand.nextCauchyZeroOne();
        if (r < 0.005)
          r = 0.005;
        else if (r > .05)
          r = .05;
        getMovementStep().setDiscriminant(i, r);
      }
  }
  public double getCellDensity_ReproductionRate() {
    return dCellDensity_ReproductionRate;
  }
  public void setCellDensity_ReproductionRate(
      double cellDensity_ReproductionRate) {
    dCellDensity_ReproductionRate = cellDensity_ReproductionRate;
  }
```

```
public double getMutationRate_CellDensity_ReproductionRate() {
  return dMutationRate_CellDensity_ReproductionRate;
}
public void setMutationRate_CellDensity_ReproductionRate(
    double mutationRate_CellDensity_ReproductionRate) {
  dMutationRate_CellDensity_ReproductionRate =
     mutationRate_CellDensity_ReproductionRate;
}
public double getMutationRate_iLifeSteps() {
  return dMutationRate_iLifeSteps;
}
public void setMutationRate_iLifeSteps(double
   mutationRate_iLifeSteps) {
  dMutationRate_iLifeSteps = mutationRate_iLifeSteps;
}
public double getMutationRate_MovementStep() {
  return dMutationRate_MovementStep;
}
public void setMutationRate_MovementStep(double
   mutationRate_MovementStep) {
  dMutationRate_MovementStep = mutationRate_MovementStep;
}
public double getMutationRate_ChemoAttractantAbsortionRate() {
  return dMutationRate_ChemoAttractantAbsortionRate;
}
public void setMutationRate_ChemoAttractantAbsortionRate(
    double mutationRate_ChemoAttractantAbsortionRate) {
  dMutationRate_ChemoAttractantAbsortionRate =
     mutationRate_ChemoAttractantAbsortionRate;
}
public double getMutationRate_ChemoAttractantDropRate() {
  return dMutationRate_ChemoAttractantDropRate;
}
public void setMutationRate_ChemoAttractantDropRate(
    double mutationRate_ChemoAttractantDropRate) {
  dMutationRate_ChemoAttractantDropRate =
     mutationRate_ChemoAttractantDropRate;
}
public double getMutationRate_ChemoAttractantExpansionRate() {
  return dMutationRate_ChemoAttractantExpansionRate;
}
public void setMutationRate_ChemoAttractantExpansionRate(
    double mutationRate_ChemoAttractantExpansionRate) {
  dMutationRate_ChemoAttractantExpansionRate =
     mutationRate_ChemoAttractantExpansionRate;
}
public double getMutationRate_ChemoAttractantRemovalLimit() {
  return dMutationRate_ChemoAttractantRemovalLimit;
}
public void setMutationRate_ChemoAttractantRemovalLimit(
    double mutationRate_ChemoAttractantRemovalLimit) {
  dMutationRate_ChemoAttractantRemovalLimit =
      mutationRate_ChemoAttractantRemovalLimit;
}
public double getMutationRate_RandomnessOfMovement() {
  return dMutationRate_RandomnessOfMovement;
}
public void setMutationRate_RandomnessOfMovement(
    double mutationRate_RandomnessOfMovement) {
  dMutationRate_RandomnessOfMovement =
      mutationRate_RandomnessOfMovement;
}
public double getMutationRate_ViewingRadius() {
  return dMutationRate_ViewingRadius;
}
public void setMutationRate_ViewingRadius(double
   mutationRate_ViewingRadius) {
  dMutationRate_ViewingRadius = mutationRate_ViewingRadius;
}
public double getChemoAttractantAbsortionRate() {
  return dChemoAttractantAbsortionRate;
}
public void setChemoAttractantAbsortionRate(
    double ChemoAttractantAbsortionRate) {
  dChemoAttractantAbsortionRate = ChemoAttractantAbsortionRate;
}
public double getChemoAttractantDropRate() {
  return dChemoAttractantDropRate;
}
public void setChemoAttractantDropRate(double
   ChemoAttractantDropRate) {
  dChemoAttractantDropRate = ChemoAttractantDropRate;
}
public double getChemoAttractantExpansionRate() {
  return dChemoAttractantExpansionRate;
}
public void setChemoAttractantExpansionRate(
    double ChemoAttractantExpansionRate) {
  dChemoAttractantExpansionRate = ChemoAttractantExpansionRate;
}
public double getChemoAttractantRemovalLimit() {
  return dChemoAttractantRemovalLimit;
}
public void setChemoAttractantRemovalLimit(
    double ChemoAttractantRemovalLimit) {
  dChemoAttractantRemovalLimit = ChemoAttractantRemovalLimit;
}
public double getRandomnessOfMovement() {
  return dRandomnessOfMovement;
}
public void setRandomnessOfMovement(double randomnessOfMovement
   ) {
  dRandomnessOfMovement = randomnessOfMovement;
}
public double getViewingRadius() {
  return dViewingRadius;
}
public void setViewingRadius(double viewingRadius) {
  dViewingRadius = viewingRadius;
}
public ProbabilityFunction getFunctionRandomMovement() {
  return functionRandomMovement;
}
public void setFunctionRandomMovement(
    ProbabilityFunction functionRandomMovement) {
  this.functionRandomMovement = functionRandomMovement;
}
public double getDBirthRadius() {
  return dBirthRadius;
}
public void setDBirthRadius(double birthRadius) {
  dBirthRadius = birthRadius;
}
public double getDMutationRate_BirthRadius() {
  return dMutationRate_BirthRadius;
}
public void setDMutationRate_BirthRadius(double
   mutationRate_BirthRadius) {
  dMutationRate_BirthRadius = mutationRate_BirthRadius;
}
public double getLifeSteps() {
  return iLifeSteps;
}
public void setLifeSteps(double lifeSteps) {
  iLifeSteps = lifeSteps;
}
public Point getMovementStep() {
  return pMovementStep;
}
public void setMovementStep(Point movementStep) {
  pMovementStep = movementStep;
```

```
}
public double getChemoAttractantRadius() {
  return dChemoAttractantRadius;
}
public void setChemoAttractantRadius(double
   ChemoAttractantRadius) {
  dChemoAttractantRadius = ChemoAttractantRadius;
}
public double getMutationRate_ChemoAttractantRadius() {
  return dMutationRate_ChemoAttractantRadius;
}
public void setMutationRate_ChemoAttractantRadius(
    double mutationRate_ChemoAttractantRadius) {
  dMutationRate_ChemoAttractantRadius =
     mutationRate_ChemoAttractantRadius;
}
public double getMutationRate_ChemoAttractantSensitivity() {
  return dMutationRate_ChemoAttractantSensitivity;
}
public void setMutationRate_ChemoAttractantSensitivity(
    double mutationRate_ChemoAttractantSensitivity) {
  dMutationRate_ChemoAttractantSensitivity =
     mutationRate_ChemoAttractantSensitivity;
}
public double getChemoAttractantSensitivity() {
  return dChemoAttractantSensitivity;
}
public void setChemoAttractantSensitivity(double
   ChemoAttractantSensitivity) {
  dChemoAttractantSensitivity = ChemoAttractantSensitivity;
}
public Genome clone() {
  Point movement = pMovementStep.clone();
  return new Genome(dChemoAttractantDropRate,
      dMutationRate_ChemoAttractantDropRate,
      dChemoAttractantExpansionRate,
      dMutationRate_ChemoAttractantExpansionRate,
      dChemoAttractantRemovalLimit,
      dMutationRate_ChemoAttractantRemovalLimit,
      dChemoAttractantAbsortionRate,
      dMutationRate_ChemoAttractantAbsortionRate,
      dChemoAttractantRadius,
         dMutationRate_ChemoAttractantRadius,
      dChemoAttractantSensitivity,
      dMutationRate_ChemoAttractantSensitivity,
      dCellDensity_ReproductionRate,
      dMutationRate_CellDensity_ReproductionRate, dBirthRadius,
      dMutationRate_BirthRadius , iLifeSteps ,
      dMutationRate_iLifeSteps , dRandomnessOfMovement,
      dMutationRate_RandomnessOfMovement ,
         functionRandomMovement ,
      movement, dMutationRate_MovementStep , dViewingRadius ,
      dMutationRate_ViewingRadius ) ;
}
public Genome PointToGenome(Point p, ProbabilityFunction
   function) {
  Genome a = new Genome(p.getDiscriminant(0), p.getDiscriminant
     (1), p
      .getDiscriminant(2), p.getDiscriminant(3),
      p.getDiscriminant(4), p.getDiscriminant(5), p
          .getDiscriminant(6), p.getDiscriminant(7), p
          .getDiscriminant(8), p.getDiscriminant(9), p
          .getDiscriminant(10), p.getDiscriminant(11), p
          .getDiscriminant(12), p.getDiscriminant(13), p
          .getDiscriminant(14), p.getDiscriminant(15), (int) p
          .getDiscriminant(16), p.getDiscriminant(17), p
          .getDiscriminant(18), p.getDiscriminant(19), function
             ,
      new Point(p.getDiscriminant(21), p.getDiscriminant(21)),
         p
          .getDiscriminant(22), p.getDiscriminant(23), p
          .getDiscriminant(24));
  return a;
}
public double createMutatedAllele(Random rand, double var) {
  double variation = rand.nextDouble() - rand.nextGaussian();
  return var;
}
public String toString() {
  return dChemoAttractantDropRate + ","
      + dMutationRate_ChemoAttractantDropRate + ","
      + dChemoAttractantExpansionRate + ","
      + dMutationRate_ChemoAttractantExpansionRate + ","
      + dChemoAttractantRemovalLimit + ","
      + dMutationRate_ChemoAttractantRemovalLimit + ","
      + dChemoAttractantAbsortionRate + ","
      + dMutationRate_ChemoAttractantAbsortionRate + ","
      + dChemoAttractantRadius + ","
      + dMutationRate_ChemoAttractantRadius + ","
      + dChemoAttractantSensitivity + ","
      + dMutationRate_ChemoAttractantSensitivity + ","
      + dCellDensity_ReproductionRate + ","
      + dMutationRate_CellDensity_ReproductionRate + ","
      + dBirthRadius + "," + dMutationRate_BirthRadius + ","
```

```
            + iLifeSteps + "," + dMutationRate_iLifeSteps + ","
            + dRandomnessOfMovement + ","
            + dMutationRate_RandomnessOfMovement + ","
            + functionRandomMovement + "," + pMovementStep + ","
            + dMutationRate_MovementStep + "," + dViewingRadius + ","
            + dMutationRate_ViewingRadius;
    }
}
```

A.3.3 ChemoAttractants Class

```
package ca.log2n.gav.asrs.swarm;
import ca.log2n.gav.maths.geometry.Point;
public class ChemoAttractants {
  Point pLocation;
  double dRadius;
  double dValue;
  double dMinStrength;
  double dRadiusStep;
  boolean bAlive;
  double dCurrValue;
  /**
   * Creates a Pheromone that mimics an expanding ring of cAMP
      whose value at
   * a point is given by the area of the cAMP. NOTE** ASSUMES
      DIMENSION OF
   * PHEROMONE IS EQUAL IN ALL DIRECTIONS.
   *
   * @param location
   *            where the pheromone is released from
   * @param radius
   *            the initial radius
   * @param value
   *            the Value/Strengh of the pheromone. Note that
      this is not
   *            effected by updates.
   * @param minStrength
   * @param radiusStep
   */
  public ChemoAttractants(Point location, double radius, double
     initialValue,
      double minStrength, double radiusStep) {
    pLocation = location;
    dRadius = radius;
    dValue = initialValue;
    dMinStrength = minStrength;
    dRadiusStep = radiusStep;
    bAlive = true;
    dCurrValue = dValue / (dRadius * dRadius);
  }
  /**
   * updates the value of the pheromone
   *
   */
  public void update() {
    if (isAlive()) {
      dRadius += dRadiusStep;
      dCurrValue = dValue / (dRadius * dRadius);
    }
  }
  public boolean isAlive() {
    return dCurrValue > this.dMinStrength;
  }
  public double getRadius() {
    return dRadius;
  }
  public void setRadius(double radius) {
    dRadius = radius;
  }
  public double getValueLimit() {
    return dMinStrength;
  }
  public void setMinStrength(double radiusLimit) {
    dMinStrength = radiusLimit;
  }
  public double getRadiusStep() {
    return dRadiusStep;
  }
  public void setRadiusStep(double radiusStep) {
    dRadiusStep = radiusStep;
  }
  public double getInitialValue() {
    return dValue;
  }
  public void setInitialValue(double value) {
    dValue = value;
  }
  public Point getLocation() {
    return pLocation;
  }
  public void setLocation(Point location) {
    pLocation = location;
  }
  public double getCurrValue() {
    return dCurrValue;
  }
  public void setCurrValue(double currValue) {
    dCurrValue = currValue;
  }
}
```

A.4 Maths Package(ca.log2n.gav.maths)

A.4.1 Function Interface Class

```
package ca.log2n.gav.maths.functions;
import ca.log2n.gav.maths.geometry.Point;
public interface Function {
  public double evaluate(Point point);
  public void setDimensions(Point point);
  public Point getDimensions();
}
```

A.4.2 Ackley Function Class

```
package ca.log2n.gav.maths.functions;
import ca.log2n.gav.maths.geometry.Point;
public class Ackley extends FitnessFunction {
  double a = 0;
  double b = 1;
  double xfactor = 0.50;
  double yfactor = 0.50;
  int n;
  double max;
  double min;
  double base;
  /**
   * default constructor.
   *
   */
  public Ackley() {
    super();
    dimensions = new Point(1, 1);
    double x = 1;
    double y = 1;
    min = 0;
  }
  public Ackley(Point dimensions) {
    super(dimensions);
    this.dimensions = dimensions;
  }
  /**
   * @param pointN
   *          a point value between zero and 1.
   * @return a value for the function between zero and 1.
   */
  public double evaluate(Point point) {
    if (point.getDim() > 2) {
      // throw exception
    }
    double x = point.getDiscriminant(0);
    x = ScaleDouble(x, 0, 1, -2, 2);
    double y = point.getDiscriminant(1);
    y = ScaleDouble(y, 0, 1, -2, 2);
    double z;
    z = -20
        * Math.exp(-0.2
            * Math.sqrt(0.5 * (Math.pow(x, 2) + Math.pow(y, 2))))
        - Math.exp(0.5 * (Math.cos((x) * 2 * Math.PI) + Math.cos
            ((y)
            * 2 * Math.PI))) + 20 + 2.71828;
    return (z);
  }
  private double ScaleDouble(double val, double fromMin, double
     fromMax,
      double toMin, double toMax) {
    double distanceFrom = fromMax - fromMin;
    double distanceTo = toMax - toMin;
    double ratio = distanceTo / distanceFrom;
    return toMin + ratio * val;
  }
  public void setA(double a) {
    this.a = a;
  }
  public void setB(double b) {
    this.b = b;
  }
}
```

A.4.3 Point Class

```
package ca.log2n.gav.maths.geometry;
import java.text.NumberFormat;
import java.util.Scanner;
import java.util.regex.Pattern;
public class Point implements Comparable {
  private double[] n;
  public Point(int dimension) {
    n = new double[dimension];
  }
  public Point(Point aVect) {
    n = new double[aVect.getDim()];
    for (int i = 0; i < aVect.getDim(); i++)
```

```
      n[i] = aVect.getDiscriminant(i);
  }
  public Point(double x, double y) {
    n = new double[2];
    n[0] = x;
    n[1] = y;
  }
  public Point(double x, double y, double z) {
    n = new double[3];
    n[0] = x;
    n[1] = y;
    n[2] = z;
  }
  public Point(double[] n) {
    this.n = n;
  }
  /**
   *
   * @returns the dimension of the Vector.
   */
  public int getDim() {
    return n.length;
  }
  public double getDiscriminant(int dimension) {
    if (dimension > -1 && dimension < getDim()) {
      return n[dimension];
    } else {
      // throw IndexOutOfBoundsException;
      return -1;
    }
  }
  public void setDiscriminant(int dimension, double value) {
    if (dimension > -1 && dimension < getDim()) {
      n[dimension] = value;
    } else {
      // throw IndexOutOfBoundsException;
      return;
    }
  }
  /**
   * sets the dimension of the environment through a space or
       coma delimited
   * string
   *
   * @param size
   */
  public boolean setDiscriminants(String text) {
```

```
  Scanner scan = new Scanner(text);
  scan.useDelimiter(Pattern.compile(",* |" + scan.delimiter()))
      ;
  int dimcount = 0;
  while (scan.hasNextDouble()) {
    if (dimcount < this.getDim()) {
      this.setDiscriminant(dimcount, scan.nextDouble());
    } else
      return false;
    dimcount++;
  }
  return true;
}
public String getDiscriminantsString() {
  String s = "";
  for (int i = 0; i < n.length - 1; i++) {
    s += n[i] + ", ";
  }
  if (n.length > 0)
    s += n[n.length - 1];
  return s;
}
/**
 * adds this vector to vector passed through and returns new
    Vector.
 *
 * @param aVect
 * @return
 */
public void add(Point aVect) {
  for (int i = 0; i < n.length; i++)
    n[i] += aVect.getDiscriminant(i);
}
public void sub(Point aVect) {
  for (int i = 0; i < n.length; i++)
    n[i] -= aVect.getDiscriminant(i);
}
public double distanceFrom(Point aVect) {
  double sum = 0.0;
  double dif;
  for (int i = 0; i < aVect.getDim(); i++) {
    dif = n[i] - aVect.getDiscriminant(i);
    sum += dif * dif;
  }
  return Math.sqrt(sum);
}
public void normalize() {
    double sizeSq = 0;
    for (int i = 0; i < getDim(); i++)
      sizeSq += n[i] * n[i];
    // if zero vector, set alignment to 0
    if (sizeSq == 0) {
      for (int i = 0; i < getDim(); i++)
        n[i] += 0;
      return;
    }
    // scale vector to be a unit vector
    double scaleFactor = 1.0f / Math.sqrt(sizeSq);
    for (int i = 0; i < getDim(); i++)
      n[i] *= scaleFactor;
  }
  public double magnitude() {
    double sum = 0;
    for (int i = 0; i < getDim(); i++)
      sum += n[i] * n[i];
    return Math.sqrt(sum);
  }
  public void scale(double ratio) {
    for (int i = 0; i < getDim(); i++)
      n[i] *= ratio;
  }
  public void scale(Point ratio) {
    for (int i = 0; i < getDim(); i++)
      n[i] *= ratio.getDiscriminant(i);
  }
  public void setMagnitude(double size) {
    this.normalize();
    this.scale(size);
  }
  /**
   * sets the point to the origin
   *
   */
  public void reset() {
    for (int i = 0; i < n.length; i++) {
      n[i] = 0;
    }
  }
  public int compareTo(Object vectorn) {
    Point a = (Point) vectorn;
    if (getDim() != a.getDim())
      ;
    {
      // throw exception;
    }
    boolean equal = false;
    for (int i = 0; i < getDim(); i++) {
      if (n[i] > a.getDiscriminant(i)) {
        if (n[i] == a.getDiscriminant(i))
          equal = true;
        else
          return -1;
      }
    }
    if (equal)
      return 0;
    else
      return 1;
  }
  public boolean equals(Object aVect) {
    Point a = (Point) aVect;
    for (int i = 0; i < a.getDim(); i++)
      if (n[i] != a.getDiscriminant(i))
        return false;
    return true;
  }
  public String toString() {
    NumberFormat form = NumberFormat.getInstance();
    form.setMaximumFractionDigits(3);
    form.setMinimumFractionDigits(3);
    String string = "";
    for (int i = 0; i < n.length; i++) {
      string += ("x" + i + ": " + form.format(n[i]) + " ");
    }
    return "Point" + n.length + ": " + this.hashCode() + " (" +
       string
        + ")";
  }
  public Point clone() {
    double array[] = new double[n.length];
    for (int i = 0; i < n.length; i++)
      array[i] = n[i];
    return new Point(array);
  }
}
```

A.4.4 Extended Random Class

```
package ca.log2n.gav.maths.random;
import java.util.Random;
```

```
public class ExtendedRandom extends Random {
  /** use serialVersionUID from JDK 1.1 for interoperability */
  static final long serialVersionUID = 7905348978240129619L;
  private boolean haveNextNextCauchy = false;
  private double b;
  public ExtendedRandom() {
    super();
    b = 1;
  }
  public ExtendedRandom(long seed) {
    super(seed);
    b = 1;
  }
  /**
   * returns a Cauchy Random Var as implemented through the
      geometric
   * description described of the Cauchy function at
   * http://mathworld.wolfram.com/CauchyDistribution.html.
   *
   * @return double cauchy based on the current b value.
   */
  synchronized public double nextCauchy() {
    long l = ((long) (next(26)) << 27) + next(27);
    double ranDouble = l / (double) (1L << 53);
    double rad = ScaleDouble(ranDouble, 0, 1, -1.57, 1.56); //
        get random
                                        // double
                                        // between -PI/2
                                        // and PI/2
    return Math.tan(rad) * b; // returns it cauchy Val.
  }
  /**
   *
   * @param theta
   *            a radian angle.
   * @return
   */
  synchronized public double nextCauchy(double theta) {
    return Math.tan(theta) * b; // returns it cauchy Val.
  }
  synchronized public double nextCauchyZeroOne() {
    return nextCauchy() / nextCauchy(1.56);
  }
  synchronized private double ScaleDouble(double val, double
     fromMin,
      double fromMax, double toMin, double toMax) {
    double distanceFrom = fromMax - fromMin;
    double distanceTo = toMax - toMin;
    double ratio = distanceTo / distanceFrom;
    return toMin + ratio * val;
  }
  public double getB() {
    return b;
  }
  public void setB(double b) {
    this.b = b;
  }
}
```

A.5 Utils Package(ca.log2n.gav.maths)

A.5.1 Statistics Class

```
package ca.log2n.gav.utils.log;
import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;
import java.util.regex.Pattern;
import ca.log2n.gav.asrs.optimizer.Environment;
import ca.log2n.gav.asrs.optimizer.Optimizer;
import ca.log2n.gav.asrs.swarm.Cell;
import ca.log2n.gav.asrs.swarm.Genome;
import ca.log2n.gav.maths.geometry.Point;
import ca.log2n.gav.maths.utils.MathUtils;
public class Statistics {
  Iterator<Cell> cells;
  Log l;
  Environment env;
  Optimizer opt;
  public Statistics(Environment env, Optimizer opt, String output
     ) {
    this.env = env;
    this.opt = opt;
    File out = new File(output);
    while (true) {
      if (out.exists()) {
        Scanner s = new Scanner(out.getName());
        String name = s.next();
        int id = s.nextInt();
        output = name + " " + (id + 1);
        out = new File(output);
      } else {
        break;
      }
    }
    l = new Log(new File(output), l.APPEND);
    String header = " Function: " + env.getFitnessFunction().
        getClass()
        + ", Objective: "
        + (opt.isMaximize() ? "maximize" : "minimize") + ",
           Severity: "
        + env.getdSeverity() + ", Randomness: "
        + env.getdRandomnessOfEnv() + ", Update Frequency"
        + env.getUpdateFrequency() + ", Max Iteration"
        + opt.getMaxIterations() + ", Max Cells: " + opt.
            getMaxcells()
        + ", Min Cells: " + opt.getMincells() + ", Reproduction:
            "
        + opt.isUsingBirths() + ", ChemoAttractants: "
        + opt.isUsingChemoAttractant() + ", ChemoRepellents: "
        + opt.isUsingCellDensity() + ", Mutation:  "
        + opt.isUsingMutations();
    l.log(header);
    l.log("Stats");
    String columns = "iteration,";
    // avg position, var
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      columns = addToString(columns, "avg_pos x" + i);
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      columns = addToString(columns, " var_pos x" + i);
    columns = addToString(columns, "avg height");
    columns = addToString(columns, "var height");
    columns = addToString(columns, "std dev height");
    columns = addToString(columns, "max height");
    columns = addToString(columns, "min height");
    columns = addToString(columns, "% Optimized");
    // avg position, var
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      columns = addToString(columns, " avg_step x" + i);
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      columns = addToString(columns, " var_step x" + i);
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      columns = addToString(columns, " stddev_step x" + i);
    columns = addToString(columns, "avg random movement");
    columns = addToString(columns, "var random movement");
    columns = addToString(columns, "std dev random movement");
    columns = addToString(columns, "avg sensitiviy");
    columns = addToString(columns, "var sensitiviy");
    columns = addToString(columns, "std dev sensitiviy");
    columns = addToString(columns, "avg distance from the genome"
        );
    l.log(columns);
  }
  // optimized
  double[] avg_position;
  double[] var_position;
  double avg_height;
  double var_height;
  // genome
  double avg_ranmovement;
  double var_ranmovement;
```

```
double[] avg_stepsize;
double[] var_stepsize;
double avg_senstivity;
double var_senstivity;
double avg_life;
double var_life;
public void collectStats() {
  if (!opt.hasMoreIterations()) {
    l.close();
    return;
  }
  Iterator<Cell> cells = opt.getCells();
  ArrayList<double[]> positions = new ArrayList<double[]>();
  for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
    positions.add(new double[opt.getCellCount()]);
  double[] height = new double[opt.getCellCount()];
  double[] movementran = new double[opt.getCellCount()];
  double[] sensitivity = new double[opt.getCellCount()];
  double max = -999999999999d;
  double min = 9999999999999d;
  ArrayList<double[]> stepsize = new ArrayList<double[]>();
  for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
    stepsize.add(new double[opt.getCellCount()]);
  // //////////////////////////////
  // collect data
  // /////////////////////////////
  int count = 0;
  while (cells.hasNext()) {
    Cell currCell = cells.next();
    Point position = currCell.getCurrentLocation();
    for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
        {
      // add positions into array
      positions.get(i)[count] = position.getDiscriminant(i);
    }
    // get height
    height[count] = currCell.getCurrentFitness();
    if (height[count] > max)
      max = height[count];
    if (height[count] < min)
      min = height[count];
    // collect genome stats
    Genome g = currCell.getGenome();
    movementran[count] = g.getRandomnessOfMovement();
    sensitivity[count] = g.getChemoAttractantSensitivity();
    Point step = g.getMovementStep();
  for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
      {
    // add positions into array
    stepsize.get(i)[count] = step.getDiscriminant(i);
  }
  count++;
}// end collecting data about cells.
// info collected now calculated avg, var and std dev
// optimized
// get avg position
double sum[] = new double[env.getEnvironmentSize().getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  for (int j = 0; j < count; j++) {
    // add positions into array
    double array[] = (double[]) positions.get(i);
    sum[i] += positions.get(i)[j];
  }
}
avg_position = new double[env.getEnvironmentSize().getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  avg_position[i] = sum[i] / (double) (count + 1.0d);
}
var_position = new double[env.getEnvironmentSize().getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  double array[] = (double[]) positions.get(i);
  var_position[i] = MathUtils.variance(array);
}
// contruct string
String out = "";
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
  out += avg_position[i] + ",";
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
  out += var_position[i] + ",";
// HEIGHT
this.avg_height = MathUtils.avg(height);
out = addToString(out, avg_height);
out = addToString(out, MathUtils.variance(height));
out = addToString(out, MathUtils.standard_deviation(height));
out = addToString(out, max);
out = addToString(out, min);
double percent = opt.isMaximize() ? max / env.getDMaxFound()
    : min
    / env.getDMinFound();
out = addToString(out, percent);
// / GENOME FEATURES
// get average step
sum = new double[env.getEnvironmentSize().getDim()];
```

```
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  for (int j = 0; j < count; j++) {
    sum[i] += stepsize.get(i)[j];
  }
}
avg_stepsize = var_position = new double[env.
    getEnvironmentSize()
      .getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  avg_stepsize[i] = sum[i] / (count + 1.0d);
}
var_stepsize = new double[env.getEnvironmentSize().getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  double array[] = (double[]) positions.get(i);
  var_stepsize[i] = MathUtils.variance(array);
}
double[] dev_stepsize = new double[env.getEnvironmentSize().
    getDim()];
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++) {
  dev_stepsize[i] = Math.sqrt(var_stepsize[i]);
}
// contruct string
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
  out += avg_stepsize[i] + ",";
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
  out += var_stepsize[i] + ",";
for (int i = 0; i < env.getEnvironmentSize().getDim(); i++)
  out += dev_stepsize[i] + ",";
// random movement, sens and life
avg_ranmovement = MathUtils.avg(movementran);
out = addToString(out, avg_ranmovement);
var_ranmovement = MathUtils.variance(movementran);
out = addToString(out, var_ranmovement);
out = addToString(out, MathUtils.standard_deviation(
    movementran));
avg_senstivity = MathUtils.avg(sensitivity);
;
out = addToString(out, avg_senstivity);
var_senstivity = MathUtils.variance(sensitivity);
out = addToString(out, var_senstivity);
out = addToString(out, MathUtils.standard_deviation(
    sensitivity));
Genome g = opt.getInitialGenome();
double ran_dif = avg_ranmovement - g.getRandomnessOfMovement
    ();
double sens_dif = avg_senstivity - g.
    getChemoAttractantSensitivity();
    double x_dif = avg_stepsize[0] - g.getMovementStep().
        getDiscriminant(0);
    double y_dif = avg_stepsize[1] - g.getMovementStep().
        getDiscriminant(1);
    // double distance_from_genome = DistanceGenome.magnitude();
    out = addToString(out, Math.sqrt(ran_dif * ran_dif + sens_dif
         * sens_dif + x_dif * x_dif + y_dif * y_dif));
    l.log(opt.getIterationCount() + "," + out);
    for (int i = 0; i < movementran.length; i++) {
      // System.out.println(stepsize.get(0)[i]+","+ stepsize.get
          (1)[i]);
    }
    // double var_avgposition;
    // double avg_height;
    // double var_height;
    //
    // // genome
    // double avg_ranmovement;
    // double var_ranmovement;
    // double avg_stepsize;
    // double var_stepsize;
    // double avg_senstivity;
    // double var_senstivity;
    // double avg_life;
    // double var_life;
  }
  public String addToString(String string, String addition) {
    string += addition + ", ";
    return string;
  }
  public String addToString(String string, double addition) {
    string += addition + ",";
    return string;
  }
}
```