
Masters Theses

Student Theses and Dissertations

Spring 2010

Dynamic ant colony optimization for globally optimizing consumer preferences

Pavitra Dhruvanarayana

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses



Part of the [Computer Sciences Commons](#)

Department:

Recommended Citation

Dhruvanarayana, Pavitra, "Dynamic ant colony optimization for globally optimizing consumer preferences" (2010). *Masters Theses*. 5126.

https://scholarsmine.mst.edu/masters_theses/5126

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

DYNAMIC ANT COLONY OPTIMIZATION FOR GLOBALLY OPTIMIZING
CONSUMER PREFERENCES

by

PAVITRA

A THESIS

Presented to the Faculty of the Graduate School of the
MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2010

Approved by

Seth Orsborn, Co-Advisor
Frank Liu, Advisor
Sanjay Madria

© 2010

Pavitra

All Rights Reserved

ABSTRACT

Consumer preference for any product or product feature can be expressed in the form of a utility function. Many such utility functions form a part of a preference map, where each of these are expressed in terms of the attributes defining the product or the product feature. In order to optimize the design, it is required to optimize the overall utility function obtained by a mathematical combination of individual utility functions defined in the preference map. The objective of this research is to devise and implement an algorithm to optimize all the individual utility functions comprised in a preference map for a product or product feature. Executed together, this will optimize the overall utility function, $U(x)$. So, an algorithm is needed to compute the optimal values for each attribute forming the individual utility functions by efficiently and thoroughly testing the entire allowed range of values in the function domain, i.e. the global optimum. The challenges faced in this include the presence of a complex space created by interactions between the various attributes in the preference map. This makes it prohibitive to solve using traditional algorithms. Thus, software agents aid in the computation as two or more software agents can collaborate on the task of optimization, enabling every single software agent to cater to a single attribute. Thus, any number of software agents can be employed to run synchronously so that all the concerned attributes can be efficiently optimized.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Seth Orsborn, for providing me the opportunity to work under him on this excellent research topic. I am extremely grateful to him for his excellent guidance, motivation and continued patience throughout my research, and the course of my graduate program and during the writing of this thesis. I would also like to express my gratitude towards Dr. Frank Liu and Dr. Sanjay Madria for their valuable suggestions, discussions and comments.

Hannah Turner and Stephen Mues from my lab team have played a significant role in my research work and I would like to thank both of them for the same. I would like to convey my special thanks to my parents and brother for their love and support all these years and during the course of my graduate studies. Their support and faith in me have always encouraged me to perform well in all my endeavors. Last, but not the least, I would like to thank my friend Vikas who provided valuable insight into certain areas of my research.

TABLE OF CONTENTS

	Page
ABSTRACT.....	iii
ACKNOWLEDGMENTS	iv
LIST OF TABLES	viii
1. INTRODUCTION.....	1
1.1. OPTIMIZATION ALGORITHMS.....	2
1.2. APPLICATION OF OPTIMIZATION ALGORITHM IN CONSUMER PREFERENCE.....	2
1.3. RESEARCH OBJECTIVE	4
1.4. TOOLS AND SOFTWARE	5
1.5. OUTLINE OF THESIS	5
2. BACKGROUND.....	7
3. ALGORITHM	11
3.1. ALGORITHM FOR OPTIMIZATION	11
3.2 COMPONENTS OF THE SYSTEM.....	11
3.3 DYNAMIC ANT COLONY OPTIMIZATION.....	14
3.4 DYNAMIC ACO FOR COLOR PREFERENCE OPTIMIZATION.....	19
4. IMPLEMENTATION DETAILS.....	21
5. EXPERIMENTS AND RESULTS	29
5.1 EXPERIMENT 1	29
5.2 EXPERIMENT 2	36
5.3 DISCUSSION OF RESULTS	39
6. CONCLUSIONS AND FUTURE WORK	41

6.1 CONCLUSIONS 41

6.2 FUTURE WORK..... 42

BIBLIOGRAPHY..... 44

VITA..... 46

LIST OF ILLUSTRATIONS

	Page
Figure 3.1 Flowchart of the System.....	15
Figure 4.1 Overall Class Diagram of the System	21
Figure 4.2 Class Diagram of DesignSpace Class.....	22
Figure 4.3 Class Diagram of Direction Class	23
Figure 4.4 Class Diagram of AntDelegate Class	24
Figure 4.5 Class Diagram of Point Class	25
Figure 4.6 Class Diagram of Ant Class	26
Figure 4.7 Class Diagram of AntColonyOptimizer Class	27

LIST OF TABLES

	Page
Table 5.1 : Accuracy for Maximization with One Run	31
Table 5.2 : Accuracy for Maximization with Multiple Runs.....	32
Table 5.3 : Effect of Change in Search Radius.....	33
Table 5.4: Number of Generations	34
Table 5.5: Accuracy for Minimization with Single Run	35
Table 5.6 : Accuracy for Minimization with Multiple Runs	35
Table 5.7: Number of Generations	36
Table 5.8: Comparison of Survey Results with Algorithm Results for Most Preferred Color	37
Table 5.9: Comparison of Survey Results with Algorithm Results for Least Preferred Color	38

1. INTRODUCTION

The goal of new product development is to create products that match consumer preferences and thus will succeed in the marketplace. These consumer preferences for products can be summarized in a utility function [1]. This utility function can then be optimized to find a set of best possible designs for a single consumer. Consumer preference is the key to finding the best possible design. Optimal consumer preference leads to optimal design [2]. Optimization of consumer preference required in new product development [2] can be achieved in two steps. The first step involves optimizing a utility function for each product attribute and the second step is to evaluate the main utility function, which mathematically combines the individual utility functions corresponding to every product attribute. The complexity of evaluating the main utility function is a direct result of factors such as the number of attributes that make up the product and the degree of the utility functions involved. The main utility function can be evaluated with considerable ease when the number of product attributes and the degree of each utility function are both significantly less. However, when either or both of these increase, the computational complexity of the tasks of optimizing individual and overall utility functions in the preference map increases tremendously, causing a direct impact on computation time and resources utilized. One technique of efficiently optimizing individual utility functions and evaluating the overall utility function has been previously proposed [2]. This paper proposes the idea of assigning multiple synchronous software agents to optimize all utility functions and a main software agent to evaluate the main utility function. A software agent would be able to optimize a utility function using an optimization algorithm that it implements. This overall technique of optimizing a utility

function and evaluating the overall utility function has been algorithmically defined in this thesis.

1.1. OPTIMIZATION ALGORITHMS

An optimization algorithm is the technique of discovering an element x from a range of values defined by some constraints, where x is such that when substituted into an objective function, it yields the maximum or minimum output. Sometimes optimization problems are expressed by complex algorithms. Global optimization is optimization with the goal to discover the best possible elements, i.e. more than one element that could constitute the optimal solution set. Thus, global optimization is comprised of all techniques that can be used to find the best elements x^* , in a set X with respect to criteria $f \in F$ [3]. In order to ensure that the optimum obtained is global, the range of values used to solve the problem must cover the entire problem space or problem domain. The range must also preferably contain real numbers.

1.2. APPLICATION OF OPTIMIZATION ALGORITHM IN CONSUMER PREFERENCE

Consumer preference is an economic term that defines an option that has the greatest anticipated value to a consumer amongst a number of other options [3]. Thus consumer preference includes the options which may be neutrally acceptable to the consumer but signify an optimal choice to the consumer owing to the other options which are less preferable. Consumer preference can be summarized and formally expressed with mathematical functions called utility functions [2]. Various other optimization techniques which could be used in this scenario such as Simulated Annealing and Genetic

Algorithms have been compared and contrasted before [5]. Utility functions can be created with various feedback mechanisms and surveys which form a part of consumer preference tests. Having studied consumer preference, the statistical data obtained can be formally expressed in the form of utility functions. Thus for every product, there can be a utility function pertaining to a certain product feature. Every product feature can be a function of a certain number of attributes. Multiple such utility functions, which signify consumer preference for different product features, can be mathematically combined to obtain the main utility function which can be said to denote the consumer preference for the entire product.

In the research presented by Orsborn *et al.* [2], a new approach to automatically generate product designs has been presented. It discusses the use of collaborative software agents as opposed to cooperative software agents to concurrently generate product designs that match consumer preference. In their research, software agents play the crucial roles of utility function optimizer and design generator. There are some key characteristics of software agents that encourage their use [2]. They are autonomous, adaptive, intelligent, efficient and portable. Since they are autonomous, they do not need human intervention. The system thus makes use of this feature of software agents to dynamically handle design generations, by continuously running agents and adding or removing them from the system as and when needed. Since this system is intended to be quick in its operation, a multi-agent platform employing collaborative software agents has been proposed in [2] to efficiently process a number of designs in the least possible time. With the aid of software agents, the computational complexity can be reduced considerably, making the process even faster.

1.3. RESEARCH OBJECTIVE

The objective of this research is to devise an algorithm which could efficiently and accurately compute the global optimum for any utility function represented by a continuous function. The second goal is to combine the use of software agents with this algorithm to make the system autonomous, continuous and performance friendly. The most prominent aspect of the algorithm is its ability to obtain the global optimum of any continuous utility function, with any number of attributes. Thus irrespective of the number of attributes, the complexity of the utility function and the size of the problem domain, the algorithm can deduce the global optimum in the most efficient manner with the aid of agents designed for the system.

In this research, the swarm intelligence method of Continuous Ant Colony Optimization has been modified and adapted in order to obtain the best results in optimizing any kind of continuous function of any form, with any number of variable parameters. This technique has been implemented on utility functions generating color preferences of various users. In this thesis, the algorithm dynamic ant colony optimization has been obtained from CACO based algorithm proposed by L. Kuhn [6]. The utility functions indicating the color preferences of various users were optimized with the dynamic Continuous Ant Colony Optimization methodology, to explore the color choices efficiently and compute the most suitable color choice of every user. This was compared with the survey results obtained from the users to analyze the correctness of color preferences deduced by the algorithm.

1.4. TOOLS AND SOFTWARE

A part of the research was to successfully simulate the algorithm on the test data obtained from users. The programming was done in Java. Agents were programmed to represent ants and various techniques were used to optimize the performance of the simulation. The system was designed to dynamically load the data set from spreadsheets containing the survey results of users. The specifications of the programming language, compiler and software tools used in the program have been listed below.

- Eclipse IDE
- Java Development Kit (JDK 5)
- Java Runtime Environment (JRE 1.5)
- Java Expression Parser (JEP 3.3.0)
- JADE for Software Agents

The specifications of hardware on which the program was run have been listed below.

- Computer Type: PC
- RAM: 3 GB
- System Type: 32-bit (Windows)
- Processor: Intel Core 2 Duo 2 GHz

1.5. OUTLINE OF THESIS

The next section of the thesis describes the literature review. Section 3 explains the algorithm in detail. Section 4 explains the implementation details of the system development. It also explains the design diagram in detail. In Section 5 various experiments and their results have been presented. It also presents a discussion of results

from the experiments. The conclusions from the experiments and the research are presented in Section 6. This section also explains some areas for future work in this research, with a brief explanation of how the future work can benefit the system developed in this research.

2. BACKGROUND

Some preliminary background in the area of global optimization techniques found in the literature will be discussed in this section. A lot of work has been done in the area of discrete function optimization, which gives insight into the various methodologies used for global optimization of discrete functions. Some of the common discrete optimization problems are the traveling salesman problem, vehicle routing problem and minimum spanning tree problem. The most common discrete optimization problem is a linear programming problem. Various methods such as evolutionary algorithms, genetic algorithms and simulated annealing have been widely researched and applied to optimization problems. However very limited options are available for using an algorithm to globally optimize a non-discrete, continuous function. Amongst the algorithms that can be used for global optimization of continuous functions are evolutionary algorithms and Swarm based optimization algorithms. The evolutionary algorithms are comprised of methods such as the genetic algorithm which uses a structure similar to chromosomes and iteratively applies some operators on the structure to present a solution to a potential problem [4]. The swarm based optimization algorithms mimic components of nature which form swarms, like flock of birds and ants. Their social behavior forms an integral part of the algorithm. Swarm based optimization algorithms are computationally inexpensive and fast. They use direct search methods which depend on evaluation of objective function, unlike most other techniques which use derivatives. Every swarm based optimization algorithm employs agents which can synchronously scan the search space to test different positions. Ants act as the search agents in case of ant colony optimization. The ant colony optimization is a relatively new field of study and has not

been extensively used to solve conventional optimization problems. It has been found to be well suited for only certain categories of optimization problems, such as in the areas of combinatorial optimization, scheduling and networks [4]. In order to apply ant colony optimization to an optimization problem, the problem needs to be modeled as a directed graph. Ant colony optimization is characterized by the use of the collective behavior of ants. In this technique, the best solutions provided by ants from previous iterations are used to base the movement of ants in the future iterations [6]. The optimization of consumer preference requires an optimization algorithm that utilizes software agents concurrently to evaluate solutions of design problems. In this context, the ant colony optimization proves to be the most appropriate technique.

The most unique adaptation of the ant colony optimization is the ant colony optimization for continuous spaces [6] which uses the movement of ants in order to direct them in multiple directions to solve continuous functions. It is modeled on the behavior of real ants but the movement of ants is designed to work for continuous functions. While ant colony optimization is intended to simply detect an optimum path within a graph, the continuous ant colony optimization (CACO) repeatedly computes the optimum path within small regions of the search space and uses the optimum path to relocate ants in further search iterations. CACO has been successfully tested on a variety of functions involving single variables [6]. It is far more adaptable to dynamically changing design spaces than most other methods [7]. Also, the computational complexity of this algorithm is less than most other leading algorithms such as Simulated Annealing, Monte Carlo simulation and Genetic Algorithms [7]. Methods such as Simulated Annealing are based on stochastic search methods. In Simulated Annealing, randomly generated moves which

satisfy a global acceptance criterion are executed. This method involves complex function evaluations in an iterative manner causing an impact on the computational complexity of the algorithm. Similarly, Monte Carlo simulation is another stochastic search based method that randomly evaluates samples from the search space to generate further iterations in the algorithm. Stochastic search methods coupled with higher computational complexity make these methods less preferable than ant colony optimization.

Ant colony optimization has been modified in numerous ways. One such version is the modified ant colony optimization (MACO) [8]. It proposes the idea of using a solution vector for each ant, which is constantly updated by making comparisons to outcomes from previous iterations. MACO was designed to work on functions involving up to two variables. This was inadequate to consider since this research requires application on multiple-variable functions.

Apart from the optimization techniques discussed above, some background research on software agents will now be presented in this section. Software agents are a relatively new concept and not much research has been done on them. The most common software agent frameworks that are being used in software agent development are the Java-based agent framework for multi-agent systems (JAFMAS) [11], Java Agent Development Framework JADE [12], Java Agent Template Lite (JATLite) [13] and Gaia Methodology [14]. Out of these, JADE was found to have the most robust architecture. It has also been widely used in developing software agents. JAFMAS is a relatively new framework and its communication mechanism is not centralized. Gaia methodology presents a methodology to create frameworks for software agent development; however it

is not a framework in itself. The main requirement from the framework for this research is the ability to create multiple collaborative agents which can be remotely deployed. This implies that it is essential that Remote Method Invocation (RMI) be a part of the framework. So, JADE proved to be the most appropriate framework for this research.

3. ALGORITHM

This section explains CACO and the modifications made to it in order to adapt it to the requirements of this research. It further expounds the steps of the algorithm, which highlight the ability of this technique to reach global optima. This section begins with an introduction to the algorithm and further defines the basic terminology used in DACO.

3.1. ALGORITHM FOR OPTIMIZATION

The algorithm used to optimize continuous functions is based on the iterative steps of sending out ants in various directions of the design space and tracing a path that would take the ants towards the global optimum. The objective of this research is to utilize the algorithm to search the problem domain and return all possible global optima. This requires some modifications to the original continuous ant colony optimization so that it is possible to return all best solutions, not just a single solution, from the design space for an n -variable function of any degree. This modified continuous ant colony optimization is explained in steps in this section.

3.2 COMPONENTS OF THE SYSTEM

Before understanding the algorithm, some background of ant colony optimization and design spaces is needed which is explained below.

The algorithm has been devised with the intention of optimizing utility functions which represent the consumer preferences for a product. The variables in the utility function signify the attributes of the product or product feature. For example, let the product that is being designed be a box and the system is expected to optimize its volume in order to optimize its design. The attributes of the box are length, breadth and height.

Utility functions for each attribute would be $u(l)$, $u(b)$ and $u(h)$, indicating optimal consumer preference for length, breadth and height, respectively. The main utility function would be another function $U = u(l) * u(b) * u(h)$, since the main utility function represents optimal volume. Thus main utility function is the product of optimal attributes of the box. This main utility function is a direct reflection of the preference of consumers for a particular design of the box.

In ant colony optimization, a colony of ants is prepared, where each ant is a thread of program holding some relevant information. In real life, ants follow the trace of other ants which have crossed their path. The ants trace other ants that have passed along the same path by sensing the presence of pheromone which is a chemical substance left behind by ants following a track towards the food source. In the virtual world application, the ants are threads of program, the food source is the global optimum and the pheromone is the best possible solution on the way. The ants constantly update the best possible solution. Thus there are some basic components of this optimization system based on CACO, which need to be understood before proceeding with the algorithm. They are as follows:

1. Step size: It indicates the least possible value the optimized variable can take.

Since, the application of this system is in the field of optimizing design features for a product, the least value that can be assumed by the attributes of the product is restricted. In color preference optimization, the RGB components of color are optimized. The RGB components can change in steps of 1, there cannot be a color 235.5, there can be only 235 or 236. Thus step size in this case becomes 1. This

step size can be used to round off the final optimum obtained into a preferred precision that suits the product.

2. Ant : Ant is a thread of program that is performing a search on a certain region of the design space. The ant also updates the best solutions on the way so that other ants can use it to trace the path towards global optimum.
3. Pheromone: For real ants, pheromone is the chemical substance left behind by ants to guide other ants coming along the same path. The pheromone trail helps ants find the path that most other ants have used in the past [6]. This path is more likely to be followed by the other ants which come upon the same trail. Also, in cases where ants run into obstacles, the pheromone trail is used by ants to converge back on their previous path. In this system, the pheromone is a variable that holds the information of all the best solutions that have come up in a certain direction.
4. Direction Vector: It is the set of directions which can be taken in the design space to search for prospective solutions. Each direction in this vector is unique and indicates a certain direction along which ants can move to find the best solution in that direction.
5. Iteration: Iteration in the algorithm indicates completion of one round of search in the current region of interest in a particular direction.
6. Generation: If one iteration has been completed in all possible directions, then a generation of search is said to have been completed. After every generation of search ends, a new generation starts with the first direction and an updated region of interest to search in.

7. Search radius: At a time, the ants search only for values in a region of interest. This region of interest is specified by a search radius which is considerably large in the beginning of the algorithm to evaluate and checks a large range of values and reduces towards the end to converge on a local optimum. At a point in time, in a certain iteration, the ants search for values only within the search radius.
8. Nest: The centre point of the search radius is the nest. The entire idea of leading the ants towards finding the optimum rests on updating the nest to the coordinates of the best known solution, after each iteration completes.
9. Design Space: The design space is the plot of the function to be optimized. In the function to be optimized, each variable corresponds to an axis in the space and substituting various values to each variable gives some points on the graph. These points constitute the entire plot representing the function. Each variable of the function is an attribute of the entire product. In case of the color preference optimization, if the function $R=u(r)$ indicates the utility function of red component for a particular user, then r indicates the number of survey responses corresponding to that level of redness in the color and R indicates the effect of that redness on the user's utility [9].

3.3 DYNAMIC ANT COLONY OPTIMIZATION

The algorithm makes use of the concepts mentioned in the previous section. Since this algorithm executes search on dynamically changing domains, it has been named the Dynamic Ant Colony Optimization (DACO). The algorithm in detail is explained in this section. The flow chart in Figure 3.1 summarizes the process, representing each step in it.

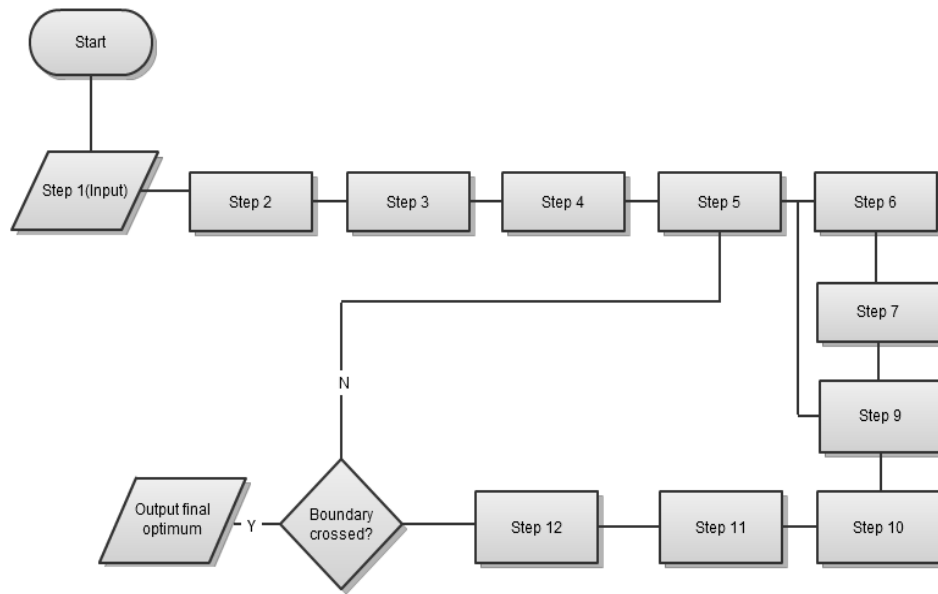


Figure 3.1 Flowchart of the System

1. First, the system inputs the details of the design space which are needed to understand the structure of the plot. These inputs are the function to be optimized, the step size, the bounds of each axis and the option to maximize or minimize.
2. Evaluate the number of directions: The number of directions is equal to the number of quadrants as per the structure of the design space, $n = \sum_i b_i$ where $i=1$ to N , where N is total number of axes, $b_i = 2$ if bounds spread over positive and negative axes and $b_i = 1$ if bounds are either both positive or both negative. The original CACO algorithm uses the idea of dividing the space into a certain number of directions say 8 or 12, but does not give a dynamic way of doing so. If the number of directions is made a function of the number of axes, then it is a

feasible option as long as the number of axes is not excessively large. Since the application of DACO is to optimize consumer preferences, the number of attributes can be unlimitedly large, in hundreds or even thousands. Perceiving the plot of such a design space is not possible. In such cases if the number of directions is a function of the number of axes, then it becomes exceedingly large. Moreover the number of points covering each direction becomes limited. This causes unnecessary computational complexity in the system. The same effect can be achieved by dividing the space into n number of directions where n is the number of quadrants. For each axis, the number of directions covering that axis will be 2 if the bounds in the axis range from positive to negative. The number of directions is 1 if bounds are either both positive or both negative.

3. Evaluate search radius: Search radius is a portion of the maximum Euclidean distance within the search space. In CACO, the initial search radius has been taken as $1/10^{\text{th}}$ of the maximum Euclidean distance in the space, but in this research it has been modified to various fractions of the maximum Euclidean distance to observe its impact on the final optimum. Out of these $1/9^{\text{th}}$ of the maximum Euclidean distance was found to give the best results for color preference optimization.
4. Evaluate number of ants required per direction: This is a function of the maximum search radius reached throughout the algorithm, since at a time in any iteration the number of ants needed to evaluate the function will not exceed the number of points within the search radius. Thus it was observed with experiments that creating k number of ants per direction, where k is the initial search radius,

gave the best results. This is because search radius is the maximum in the beginning of the algorithm and reduces as it progresses.

5. Initialize the nest to origin of the search space for the first generation: In CACO this nest remains the same throughout the algorithm. CACO uses nest to simply assign a start point for the ants to travel out into a certain direction. However, in this paper this step was modified to assign a nest to every direction following the first generation of search, so that ants could continue searching from the best position of the previous generation, which the nest is updated to. This step will be explained in detail in the next section.
6. Randomly select a direction from the direction vector, to start the evaluation process. Initialize the nest to origin and create ants for the direction. Each ant is assigned a point within the search radius. The ant substitutes the coordinates of the point into the function to evaluate the output. All outputs are compared to find the best point, depending on Maximization or Minimization option that has been chosen. The nest is updated to this best point for the selected direction.
7. Pheromone for the direction is updated to the current pheromone summed with the best output of the iteration. This way pheromone indicates potential best values for each direction.
8. Step 6 and Step 7 are repeated for all other directions. This completes one generation of search.
9. Before the next generation begins, all directions are ranked and ants are redistributed to assign more ants to the best direction i.e., the direction with the optimum pheromone. This means that if the algorithm is being run for

maximization, more ants are assigned to the direction which has maximum pheromone and lesser ants to the direction of lesser ranks. Similarly in case of minimization, the directions are ranked in the reverse order with the direction yielding minimum output ranked on top. This is done so that more ants are assigned to look for optima in the direction where the probability of finding the optima is the highest. Thus with one generation of search completed, the top ranked direction gets more share of ants from the total pool of ants in all directions. In order to select the best output from a direction, all ants' outputs are sorted using bubble sort and the coordinates yielding optimum are converged upon by selecting the coordinates giving maximum or minimum depending on the optimization option.

10. After every generation of search following initial threshold number of searches, the search radius of each direction is updated to a fraction of the original search radius which is given by $\text{search radius} = \text{search radius} * m$, where $0 < m < 1$ and preferably as small as possible, say 0.1. This constantly reduces the search radius without reaching 0 and thus exhaustively searches all possible regions of the search space. The search radius begins reducing after the number of generations has exceeded t , where t is a threshold value taken as an input parameter.
11. If an ant evaluates the same coordinate for multiple successive iterations, it is marked as starved and made to update its coordinates to the point outside the search radius. The number of iterations evaluating same coordinate can be set as a parameter in the beginning of the algorithm.

12. The feature which markedly makes DACO different from CACO is the method of storing peaks. The algorithm records the coordinates which are peak positions in the plot of the function. The peak positions are positive peaks in case of maximization and negative peaks in case of minimization. All the saved peak positions are sorted to give the coordinates of the most maximum or most minimum peak. This is needed because the application of this algorithm is in mainly in consumer preference. In such areas or other similar areas that require the choices to be Pareto optimal [10]. There could be plots that come from constant functions and absolutely flat. In such cases all the values yielding the constant output should be returned as optimal choices. Thus the output from the algorithm is any coordinate which gives the optimum utility from the utility function.
13. The algorithm exits when the constraints are violated or coordinate points outside the bounds of the plot are assigned to ants to evaluate. In this case the peaks which have been stored so far are sorted. The sorting is done according to optimizing option, which could be maximization or minimization. These peaks represent the various function outputs in the graph and the most optimal output is chosen to be returned. All coordinate points which resulted in this optimal output will be returned by the algorithm. These will represent the most desirable choice of attributes to be chosen for the product.

3.4 DYNAMIC ACO FOR COLOR PREFERENCE OPTIMIZATION

In case of color preference, the product at hand is color and its attributes are the red, green and blue (RGB) components of color. The utility functions for the attributes of

color were obtained from the surveys presented in [9]. Every utility function represents the preference of a color component to a single user. With the dynamic ant colony optimization, all such utility functions of all users for every color component are optimized. The resultant outputs indicate the optimal color component preference for a particular user whose utility function was optimized. If the optimal color components are combined using the RGB color model, then it would present the most preferred color for the user. This entire process of optimization can be carried out using Dynamic Ant Colony Optimization to test the complexity, processing time and accuracy of outputs for the algorithm. Here, the main utility function is $U(R,G,B)$ which combines the red, green and blue components to generate a color that is probably the user's most preferred color.

4. IMPLEMENTATION DETAILS

The Dynamic Ant Colony Optimization was implemented in Java 5.0 using the Eclipse IDE. The open source Java library JEP (Java Expression Parser) was used to evaluate the test functions for different ranges of values within the bounds. This section explains the design decisions and details of coding the algorithm. Unified Modeling Language (UML) was used to design the system.

The system was divided into various functional components required for the algorithm. The different classes, their class diagrams and functionalities are explained in detail below. Figure 4.1 gives the overall class diagram for the entire system. It shows the relationship between the classes defined in this section. The main class is the AntColonyOptimizer that refers to an instance of the DesignSpace class and a collection of Direction class objects. Each object of the Direction class in turn refers to an object of the AntDelegate class which refers to a collection of Ant objects. Each Ant object can access the JEP library that is used for function evaluation.

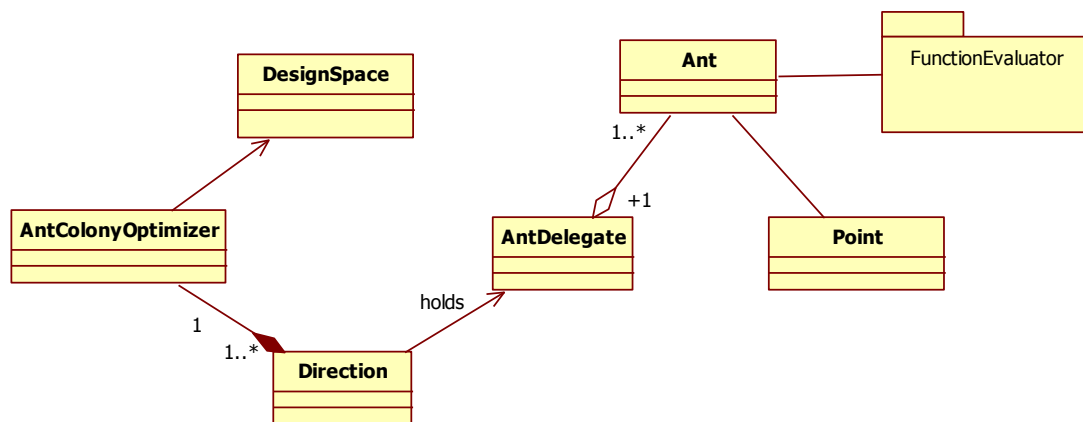


Figure 4.1 Overall Class Diagram of the System

- **DesignSpace class:** An instance of this class is initialized in the beginning of the program to define all the details regarding the function space. These details can be the utility function that represents the design space, the list of attributes representing the design properties, the bounds of the function and the least value that the attribute can take, i.e. the step size. Figure 4.2 presents the class diagram for this class.

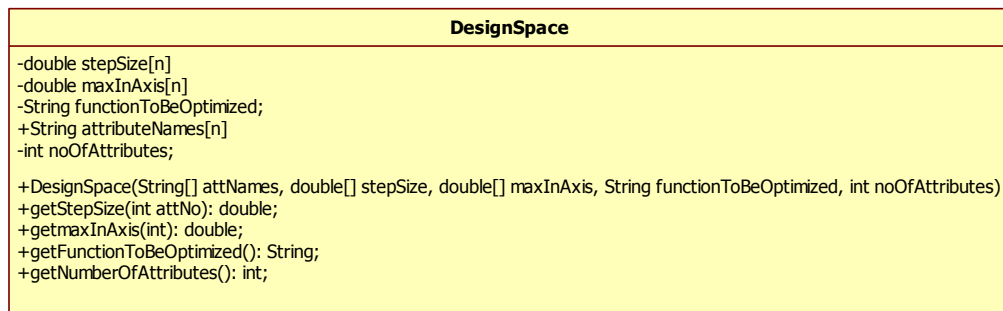


Figure 4.2 Class Diagram of DesignSpace Class

- **Direction Class:** An object of the direction class represents a unique direction in space. All ants are divided into n sets, where n is the number of directions. All ants that belong to a certain direction can move only in that direction. Each direction object has details regarding that direction such as a unique number to identify that direction, a metric given by the variable *pheromone*, that indicates how good the direction is proving to be in terms of optimal attribute search, the number of ants moving in that direction, the nest for that direction around which the ants are searching the optimum in a certain iteration, the search radius for

every iteration and the best coordinate obtained after every iteration of search. The metric *pheromone* is constantly updated to the best optimum of an iteration summed up with the previous *pheromone*. This is done after every iteration of search ends. This way it indicates the probability of finding the optimum in that direction. The direction class is extremely important since it holds the best coordinates obtained from every iteration. The nest in a direction is updated at the end of each iteration to the coordinates giving the current optimum output. Figure 4.3 presents the class diagram for this class.

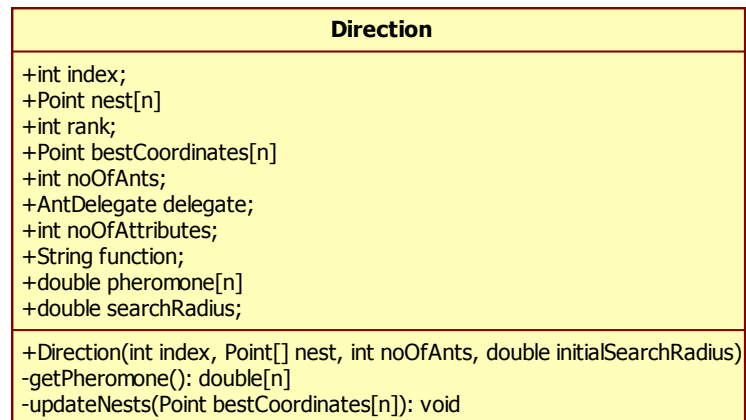


Figure 4.3 Class Diagram of Direction Class

- **AntDelegate:** The AntDelegate class acts as an interface between the ants and the direction that the ants are moving in. The delegate has a reference to the set of ants moving in a particular direction. It also handles the starvation of ants. It marks the ants *starved* if they are evaluating the same set of coordinates iteratively. The number of iterations in which the same ant repeatedly evaluates

the same coordinate before being marked as *starved* can be taken as an input parameter. Typically it is preferred to be in the range to 5 to 10, in order to quickly mark such ants as starved and make them move outside the search radius. Quickly moving them out of the search radius reduces execution time. Each direction object has a reference to an AntDelegate object which in turn maintains a set of ants which it delegates the task of searching for optimum coordinates around the nest of the concerned direction. Figure 4.4 presents the class diagram for this class.

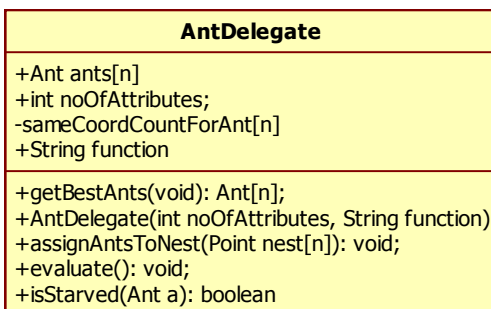


Figure 4.4 Class Diagram of AntDelegate Class

- **Point:** Each object of the Point Class is indicative of a coordinate in space. Since, the point in space is a function of the number of attributes that make up the utility function, so it needs to be able to dynamically change to a set of any number of coordinates to make up one point. This number changes from one utility function to the other. If the utility function is $f(x,y)$ then the object of Point class is represented by the $\langle x,y \rangle$ coordinates in space. However if the function is $f(x,y,z)$

then every point in space is represented by $\langle x,y,z \rangle$. Also, every point needs to be represented using the exact name of the attribute as it is used in the utility function. So if the utility function is $f(r,g,b)$ then the point in space would be $\langle r,g,b \rangle$. The Point class takes this information of the attribute names from the design space. The object of Point class is used extensively in the system since every point that an ant evaluates is represented by the object of Point class and is understood by the ant through that representation. Similarly, the nest around which the ants need to search in an iteration and the coordinates yielding the best output are both represented by objects of the Point class. Figure 4.5 presents the class diagram for this class.

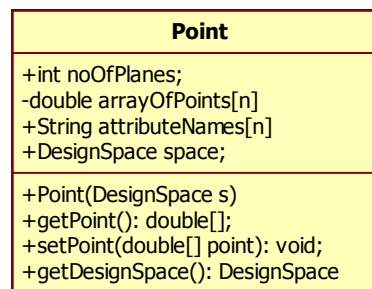


Figure 4.5 Class Diagram of Point Class

- **Ant:** The Ant class is used to represent one ant. All the details an ant needs and returns are stored in this class. Ant class thus has instances of the output generated, the coordinate evaluated which is an object of the Point class, the nest around which the evaluation was done which is again an object of the Point class

etc. The ant is the least correlated and most independent entity of the system. Each ant object is what eventually performs the evaluation of the function for a certain coordinate value. This evaluation is done by the JEP library. Thus the instance of the JEP library is used within the Ant class. The evaluate() method in the class takes in a string which is the objective function and passes it to the JEP library to evaluate the function and return the output. Figure 4.6 presents the class diagram for this class.

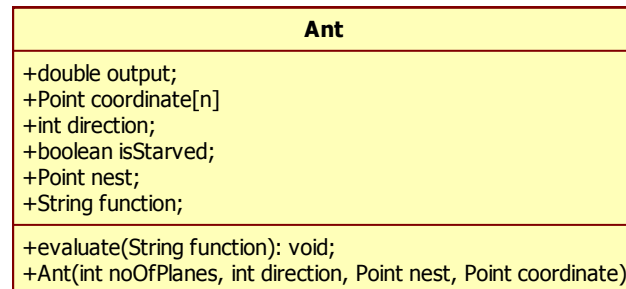


Figure 4.6 Class Diagram of Ant Class

- **AntColonyOptimizer:** This class is the main class where the program starts running. The instances of the DesignSpace and the direction vectors are created in this class. Thus this class is used for parameter setting within the DesignSpace object which can later be referred to by the other classes. This class also holds the utility function to be optimized. This can be explicitly mentioned in the main class or read from text files. All methods which pertain to updating data or continuing or stopping the algorithm, are implemented in this class. The method evaluateNoOfAnts() evaluates the number of ants needed for the algorithm. This

is a function of the size of the domain. The method `evaluateNoOfDirections()` evaluates the number of directions that is obtained by computing the number of quadrants in the domain. It also has the methods `startACO()` and `stopACO()` which start and stop the algorithm. The variable `finalOptimum` indicates the optimum obtained at the end of the algorithm. This variable can be directly output from the system and saved on to output text files or a database for further reference. All such outputs are referred to by the main utility function evaluator that would sum up or multiply the individual optimal attributes. Figure 4.7 presents the class diagram for this class.

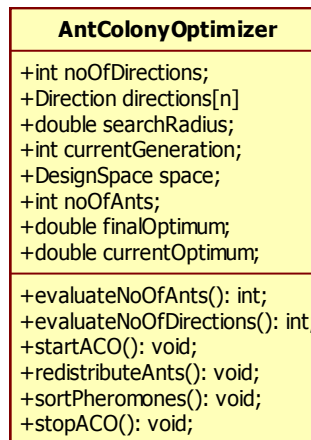


Figure 4.7 Class Diagram of AntColonyOptimizer Class

The software agents developed to act as the main agent to combine the individual utility functions were developed using the framework JADE which is an open source framework for software agent development.

Some design patterns were applied in designing this system. The expert pattern was used in the Ant Class. Every Ant object is an expert since it alone accesses the JEP Library, so it alone knows how to evaluate a function. Similarly, creator pattern was used in the AntDelegate Class. This class is responsible for creating sets of ants meant to search in a certain direction. Also design patterns such as high cohesion and low coupling were used. All classes are independent in their functionalities and hence have low coupling with each other. Also each class is responsible for providing a unique functionality not handled by another class, making all the classes highly cohesive.

5. EXPERIMENTS AND RESULTS

5.1 EXPERIMENT 1

DACO has been developed to optimize a utility function of any degree containing any number of variables. However to run the algorithm, a set of utility functions to be optimized is needed. Utility functions are typically obtained from consumer surveys comprising of choice-based questions. The utility functions for the experiments on DACO were collected from the results of choice-based surveys conducted in the research conducted by Turner, et al. [10]. These utility functions are cubic in nature and contain a single variable defining a level of color-component that makes the color. In these surveys, the color of product was based on the RGB color model and hence was broken into red, green and blue components, creating 3 utility functions per consumer.

The dynamic ant colony optimization was run on these functions. Every function in this set represents color preference of a certain user for a certain color component: red, green or blue. Thus all functions were divided into sets of 3, each set corresponding to one user and 3 color components. Utility functions were generated from 40 users taking the survey. The algorithm was run on these functions to maximize and minimize them. All the functions were stored in an input text file. A parser was written to generate functions in case only the attribute coefficients were known. A batch-processing program was written so as to pick the functions from the input file. The outputs obtained were saved in to another set of text files. In order to verify the outputs from DACO, the algorithm was compared with another source that can compute maximum and minimum of functions. The numerical computing language Matlab was chosen. All the utility functions optimized by DACO were plotted in Matlab; Matlab commands were used to

maximize and minimize the functions. Their outputs were written into separate text files. These were later merged with the files obtained from the algorithm so that they could be compared. The deviations of the outputs from DACO were recorded in terms of various degrees such as 0, ± 1 , ± 2 and ± 3 . The experiments done with maximization are shown in Tables 5.1, 5.2 and 5.3.

Table 5.1 shows the deviation of maximization outputs returned by the algorithm from the outputs returned from Matlab batch files. This experiment was conducted with search radius fixed to the maximum Euclidean distance in the search space multiplied by a multiplication factor of 1/8. The algorithm was run once in this case. The execution time for all of the 120 functions was 4.1 seconds, on the PC described in Section 1.4. This execution time indicates that DACO can compute the optimum of a single function in approximately 30ms with a standard processor. Since DACO has been implemented to be applied on continuous quick design generation processes, the execution time for every utility function being less than a second is a desirable feature.

It can be observed that there was no deviation observed between the output from algorithm and the output from Matlab in most functions. There was a deviation of ± 1 encountered in most other functions. Very few functions resulted in a difference of more than 2. These variations of ± 2 and ± 3 were caused due to two major factors. It can be concluded from the observed results that Matlab's internal setting of using index 1 for storing the start index of any matrix could be one factor causing deviations of ± 1 . Since all values are internally stored as matrices in Matlab, this may lead to an inconsistency for cases where the correct output is one of the boundary values. This comes from the observation that almost all deviations of ± 1 have been caused in cases of optimal outputs

being boundary values. The other factor that could have affected the accuracy of output is the use of random numbers in positioning ants within the search radius. The number of ants used within a search radius decreases with every passing iteration. In some cases, the number of ants becomes lesser than the number of coordinates within the search radius. Since, random numbers are used to generate a unique location for every ant within search radius, so in such cases, due to lack of ants, there are some locations which do not get generated. There is a probability that this missed location leads to the most accurate output. In such cases the value that gets returned by the algorithm is in the range of ± 1 or ± 2 .

The solution to the problem caused by this factor is to be able to dynamically manage the number of ants so that every location in the search radius is covered by an ant. If the number of ants is always maintained to be a number larger than the number of coordinates in the search radius from previous iteration, then there could be more than one ant in the search radius being assigned the same coordinate for evaluation. This may lead to multiple ants referring to the same coordinate, leading to some logical issues. Thus, ants have to be constantly managed and increased and decreased in number. This could affect the performance of the system due to the continuous increase or decrease in the number of ants in every iteration.

Table 5.1 : Accuracy for Maximization with One Run

Deviation from Matlab	Occurrences	Percentage
Deviation of 0	72	60.0%
Deviation of ± 1	33	27.5%
Deviation of ± 2	9	7.5%
Deviation of ± 3	6	5.0%

It was observed that these deviations can be further reduced by the repeated running of the algorithm and returning the average of all the outputs, as the final output. The algorithm by itself exits upon reaching coordinates at the boundary for evaluation. This cannot be changed but the entire algorithm can be repeated from start for the same function to obtain an average final optimum from multiple runs. This experiment eliminated the deviations to a large extent as can be seen from Table 5.2. The number of runs on the algorithm was varied from 5 to 13 and the accuracy of the returned output was tabulated. The execution time was also recorded for these, since an increase in the number of runs caused a direct impact on the execution time.

The repeated runs totally eliminated deviations of ± 3 . Only certain occurrences of deviations of ± 2 were observed with most runs. There was a maximum of up to 6 deviations caused with 7 runs and a minimum of 0 deviations with 10 runs. Thus with 10 runs the outputs were perfectly matching the outputs from Matlab. The execution time ranged from 19 seconds for 5 runs to up to a minute with 15 runs.

Table 5.2 : Accuracy for Maximization with Multiple Runs

Number of runs	Deviation of ± 2	Execution time(seconds)
5	5	19
7	6	27
9	2	36
10	0	41
13	2	54
15	2	58

The parameters which are set for every optimization problem are the search radius, function to be optimized, and the bounds. Out of these the parameters, search radius was varied by changing the multiplication factor for the maximum Euclidean distance in the search space. A change in search radius has significant effects on the final output. It was observed that with too large a radius, potential optimum values lying in a range of values less than the search radius could get missed. With too small a radius, the ants could get repeatedly starved, making the algorithm run in an incremental fashion, evaluating almost every coordinate in the search space.

It can be seen from Table 5.3 that multiplying the search radius with a factor of 1/8 yielded the least number of deviations, whereas a factor of 1/9 was the least accurate. The deviations here are in the range of ± 2 which also includes ± 1 .

Table 5.3 : Effect of Change in Search Radius

Multiplication Factor for Search Radius	Deviation of ± 2
1/6	10
1/7	10
1/8	7
1/9	19
1/10	9

The number of generations was also studied. A generation is one round of evaluations in all directions in the search space. The number of generations is a metric which can indicate what percentage of search space was evaluated to converge on the final optimum. The performance of the system in terms of its speed is inversely proportional to the number of generations. If all the coordinates in the search space are

evaluated then the approach gets close to brute force approach and the selectivity of ants cannot be considered be well utilized in such a case.

In the experiments run on maximization of the functions, the range of number of generations was from a minimum of 68 to a maximum of 110. This range was for experiments run with the search radius $r = 1/9 * (\text{Maximum Euclidean Distance})$.

Table 5.4 shows the range of the number of generations that the algorithm completed before the final maximization output was obtained.

Table 5.4: Number of Generations

Number of generations (n)	Number of functions completing n generations
60-70	6
70-80	24
80-90	47
90-100	35
100-110	8
Range: 60-110	Total: 120 functions

This indicates that even though there were 255 coordinates in all, searches were centered around 60 to 110 coordinates which were marked as best coordinates on the way. This was possible by choosing the best coordinates as nests on the way.

All the experiments were also performed for minimization. Table 5.5 shows the accuracy of the algorithm with minimization of the functions using one run. It can be observed from Table 5.5 that the number of results matching perfectly with Matlab is almost same as that with maximization.

Table 5.5: Accuracy for Minimization with Single Run

Deviation from Matlab d	Number of functions with deviation d	Percentage
Deviation of 0	69	57.5%
Deviation of ± 1	40	33.3%
Deviation of ± 2	7	5.83%
Deviation of ± 3	4	3.33%

Table 5.6 shows the effect of number of runs of the algorithm on the accuracy of the results. As compared to maximization, minimization gives a very accurate result with a much lesser number of iterations. This requires less execution time as well. However 10 runs show the same effect of perfect accuracy with both maximization and minimization and have an execution time that is not very expensive in terms of performance.

Table 5.6 : Accuracy for Minimization with Multiple Runs

Number of runs r	Number of results giving Deviation of ± 2 , with r runs	Execution time(seconds)
5	0	20
7	3	31
9	1	36
10	0	43
13	0	54
15	2	58

Table 5.7 shows the number of generations covered for the entire algorithm using one run, before arriving at the optimal output. The number of generations is in the same range as needed for maximization.

Table 5.7: Number of Generations

Number of generations n	Number of functions completing n generations
60-70	4
70-80	19
80-90	44
90-100	43
100-110	10

The maximization and minimization runs were performed on a function with two variables and a function with no variable i.e. a constant function. In case of the constant function, all values are expected to be returned as equally optimal. This desired result was obtained.

5.2 EXPERIMENT 2

One more experiment that was conducted was to match the output from main utility function with the results of follow-up survey conducted on the same consumers. The follow up surveys offered choice based questions containing the optimal color obtained by optimizing the cubic equations representing color components and combining them to yield a color choice, using standard derivative-based optimization. The survey also presented to the consumers colors that were computed to be of minimal preference and average preference to the consumer. The consumers' choice of optimal color from the survey matched the optimal color from the algorithm in most cases, indicating that combining the optimal color components indeed yields the optimal color choice for the concerned consumer.

For more than half the total number of users, the colors were perfectly matching for both maximization and minimization. However, in some cases the colors obtained from maximizing the function using the algorithm matched up with the average preference of the consumer. Some sample output images from the experiment have been presented below.

Table 5.8: Comparison of Survey Results with Algorithm Results for Most Preferred Color










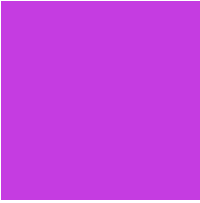

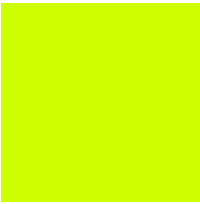

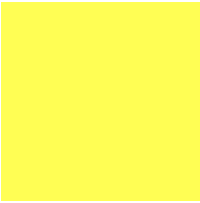


Most preferred Color Choice from	Maximized color Choice from Algorithm
	
	
	
	

Table 5.9: Comparison of Survey Results with Algorithm Results for Least Preferred Color

Least preferred color choice from survey	Minimized color choice from Algorithm
	
	
	
	

5.3 DISCUSSION OF RESULTS

It may be noted that the experiments were conducted on a scale much smaller than one found in a typical design generation environment, in terms of the number of attributes and size of the design space. For example, there could be hundreds, thousands or even greater number of attributes in a typical automobile design preference scenario. The bounds, design space, degree of utility function would all be varying from one attribute to the other. The system is not expected to assume any parameter in any aspect of the computation. Each run of the algorithm would have different parameters from the next run. The system is expected to consistently output optimum attributes for each utility function, irrespective of any of the above mentioned factors.

DACO has been developed keeping this scenario in mind. Thus, none of the parameters that vary according to design criteria are fixed to a certain value in the algorithm. These varying parameters are the number of attributes, the range of design space, the bounds, the mathematical signs of the bounds of each attribute, the optimization option (maximization or minimization), the minimum value that can be assumed by an attribute (step size) and the degree of the utility function. So, if the first utility function optimized by DACO is a quadratic function with 3 variables and the next utility function is a 5th degree function with 5 variables, then both would be independently optimized using different suitable parameters, by two instances of DACO running in the system.

This algorithm was tested on a set of 120 utility functions, each of degree 3 and having 1 variable. Since none of the parameters was assumed by the system and it dynamically computed the parameters required for this kind of utility function, it would do the same for a utility function of n degree, with k variables where n and k could be as

large or as small as possible. In the color preference research discussed, 3 utility functions formed a part of the main utility function. All 3 utility functions were of the same degree and same number of variables. DACO has been designed to create the main utility function with any number of utility functions of little or no similarity in terms of degree and variables, as long as the number of utility functions making up the product is specified as an input to the system. Thus DACO is dynamic to the extent of evaluating the optimum utility of each product attribute and combining all such utilities with the aid of input parameters set according to the product being designed.

6. CONCLUSIONS AND FUTURE WORK

The algorithm DACO was developed to optimize utility functions obtained from consumer surveys, in order to understand the most preferred product attributes that can be combined into a complete product. This section explains the conclusions that can be drawn from the algorithm regarding its application, performance and other observations. This section further details the areas in which the algorithm could be further worked upon, to make it more dynamic, robust and performance friendly.

6.1 CONCLUSIONS

The goal of this research was to devise an optimization algorithm to correctly recognize the global optimum of utility functions derived from consumer inputs, which could comprise of a large number of attributes, making the functions computationally complex and dynamically changing. This required a dynamic algorithm which could identify the pattern of the function on the run and globally optimize it. The function evaluator used in the implementation of the algorithm is an open source expression parser library. It handles evaluation of a function of any pattern. But the library needed attribute names and values to be substituted in them, as inputs, which was handled dynamically by the algorithm. The other goal of this research was to use software agents to mathematically combine the outcomes of various utility functions to synchronously yield a comprehensive output representing user preference. These goals were achieved by adapting the Continuous Ant Colony Optimization (CACO) to create a dynamic algorithm which can globally optimize any function based on the inputs fed to the system. The algorithm is adaptable to any kind of utility function, with a design space of

any size. However, it was mainly tested on a simple test space of a considerably small size, with cubic utility functions containing one variable. These were obtained from studying consumer preference for color. The algorithm accurately optimized the utility functions generated from all users' survey results. The best results were achieved for both maximization and minimization performed on these functions using multiple iterations. Results were compared to maximum and minimum values of variables returned by Matlab for the same functions. A software agent was created to run any referenced program, so that it could generate colors out of the optimal values of R, G and B components returned by the algorithm.

6.2 FUTURE WORK

The robustness of the algorithm could be enhanced by creating a dynamic way of setting its parameters like the most desirable search radius and the most desirable number of runs. This could make the system more dynamic, involving no human intervention whatsoever, as desired in the system proposed in [2]. There is also a need to make the number of ants change dynamically so that in each iteration it changes to the number of coordinates available within the search radius in that iteration, so that every point in the search radius gets evaluated, increasing the accuracy of the outputs. Currently, the number of ants is initialized to a number computed in the beginning of the algorithm. This number is based on the size of the design space. However, as the algorithm progresses, if the number of ants reduces then there is no provision to increase it again. This may lead to a dearth of ants in some regions of search, causing inaccurate results. The design of the system could be changed to automate parameter settings for the algorithm. These parameters are the portion of maximum Euclidean distance that needs to

be taken as search radius initially, the number of iterations leading to starvation of ants, the number of attributes and the attribute names needed by the JEP library. Currently these details are manually entered into the system as inputs. Automating these inputs could also increase the performance of the system in terms of execution time and computational complexity. The long term effort would be to automate the entire system to handle any kind of function without requiring any change to the parameters such as the initial search radius or the number of ants. When these classes get referenced by software agents that are continuously active, then it is ideal to keep all such parameters dynamically handled. Also, the class which is referred to the main agent, which is a software agent, needs to be designed to understand the mathematical combination required for the main utility function. Currently the main agent class has been explicitly written to handle combination of colors. So the current main utility function can be assumed to be a summation function of individual utility functions. But in some cases, the main utility function may need to compute the product or some other mathematical combination of individual utility functions. The algorithm also needs to dynamically know the number of individual utility functions being optimized, so as to enable it to combine all those functions. In color preference optimization, the number of individual utility functions is 3. But this has been currently fixed in the system and needs to be dynamic in a generic utility function optimization system. These could be some areas of future research in enhancing this algorithm.

BIBLIOGRAPHY

- [1] Orsborn, S., Cagan, J. and Boatwright, P., 2009, "Quantifying Aesthetic Form Preference in a Utility Function," ASME Journal of Mechanical Design, 131(6), 061001.
- [2] Orsborn, S. and Cagan, J., 2009, "Multi-Agent Shape Grammar Implementation: Automatically Generating Form Concepts According to a Preference Function," ASME Journal of Mechanical Design, 131(12), 121007.
- [3] C.F-Schaw, K.Telay, I.Vloerbergh, J.Chenoweth, G.Morrison, C.Lundéhn, Measuring customer preferences for drinking water services, TECHNEAU 2006.
- [4] Global Optimization Algorithm- Theory and Application, Second Edition, Thomas Weise, Chapter 1, pp 21-22.
- [5] Sachdev, S., 1998, Explorations in Asynchronous Teams, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, P.A.
- [6] L.Kuhn, Ant Colony Optimization for Continuous Spaces, Thesis submitted to The Department of Information Technology and Electrical Engineering The University of Queensland.
- [7] Gambardella L.M., Engineering Complex Systems: Ant Colony Optimization to Model and to Solve Complex Dynamic Problems, SELF-STAR 2004, Self-* Properties in Complex Information Systems, Italy, 31-May - 2 June 2004.
- [8] M. Duran Toksarı, A heuristic approach to find the global optimum of function, Journal of Computational and Applied Mathematics 209 (2007) 160 – 166.
- [9] H. Turner, S. Orsborn, K. H. Lough, Quantifying product color preference in a utility function, ASEM Conference, 2009
- [10] <http://faculty.washington.edu/wtalbott/phil466/hdpo.htm>, Last Accessed March 14, 2010.
- [11] D. Chauhan. "Developing coherent multiagent systems using jafmas," International Conference on Multi Agent Systems, ICMAS98, Cite des Sciences - La Villette, Paris, France, July 1998.
- [12] F. Bellifemine, G. Rimassa, and A. Poggi. JADE – A FIPA-compliant agent framework, 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99), pages 97–108, London, UK, December 1999.

- [13] H. Jeon, C. Petrie, M.R. Cutkosky, JATLite: A Java Agent Infrastructure with Message Routing, “<http://www-cdr.stanford.edu/~petrie/agents/jatlite.pdf>,” Last Accessed March 22 2010
- [14] F. Zambonelli, N. Jennings, and M. Wooldridge, “Developing Multiagent Systems: The Gaia Methodology,” *ACM Trans. Software Eng. and Methodology*, vol. 12, no. 3, pp. 417-470, July 2003.

VITA

Pavitra received her Bachelor of Engineering (B.E.) from the Ramrao Adik Institute of Technology (RAIT), Mumbai, India in the field of Information Technology in June 2006.

She worked as Programmer Analyst at Kale Consultants Ltd, Mumbai, India from July 2006 to May 2008. She joined the Missouri University of Science and Technology (Missouri S&T) in January 2009. She later interned at Cerner Corporation from May 2009 to July 2009. She received her Master's degree in Computer Science in May 2010. Her areas of interests include distributed computing, image processing, mobile sensors and network security.