MULTIPLE VEHICLE POSITIONING SIMULATION AND OPTIMIZATION

By

ERICA FRANCES ZAWODNY

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

ACKNOWLEDGMENTS

I would like to thank Dr. Carl Crane for being the chairperson of my committee and for providing advice and encouragement throughout. In addition, I would like to thank Dr. John Schueller and Dr. Gloria Wiens for serving on my committee and offering their expertise. I would especially like to thank Donald MacArthur for his undying support and encouragement throughout this experience. I thank my parents, Joe and Yvonne, and my brother, Nikolas, for all of their love, support, and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Master of Science

MULTIPLE VEHICLE POSTIONING SIMULATION AND OPTIMIZATION

By

Erica Frances Zawodny

May 2003

Chair: Carl D. Crane, III
Major Department: Mechanical and Aerospace Engineering

There is a wealth of research being performed in the area of autonomous vehicles. Much of the research involves development of platforms and integration of computers and various sensors to create an intelligent robot. With the development of multiple robotic agents comes an interest in multiple robot interaction. There are two limiting factors with this type of research: robot cost and robot size. Therefore, much of the research is performed in simulation initially and later implemented on actual systems.

In order to optimally place robots throughout a region an algorithm should first be developed and tested in simulation. Related work has been done on optimal placement of sonar beacons throughout a region for a beacon based navigation system.

There are several issues that need to be addressed when positioning multiple robots. The number of robots to be placed, the size of the region, the obstacle locations within the region, and sensor limitations are all important issues that are addressed in the solution of this problem. In addition, the optimization technique chosen will have

significant effect on the results and the time it takes to determine a solution. This raises the issue of how to determine where the robots should be positioned for a given environment.

This particular problem is an optimal placement type of problem. Problems of this nature and search problems can be solved using genetic algorithms. Genetic algorithms are a type of directed, randomized search technique that is conceptually analogous to the process of natural selection as seen in nature.

This thesis utilizes genetic algorithms to solve this multiple robot placement problem. The software developed can be applied to any map containing known obstacles and a user-specified number of robots. The program will try to minimize the combined shadow area (area that is not visible to any of the robots) for a specified number of robots by placing them at different locations until an acceptable solution is found.

The result of this research is software that can take a two-dimensional map with known obstacle locations and determine where to position multiple robots such that maximum coverage is ensured and line-of-sight communication among the robots is maintained. A graphical user interface allows ease of use and displays results to a window.

This work can be extended to the three-dimensional case where the terrain and elevation are taken into account. In addition, it could later be implemented on a multiple agent ground vehicle system or air-ground vehicle system.

CHAPTER 1
INTRODUCTION

A variety of research is being done in the area of autonomous mobile robots. Much of this research involves the development of platforms and integration of computers and various sensors to create an intelligent robot. With the development of multiple robotic agents comes an interest in multiple robot interaction. Researchers in this area explore how robots interact with one another and their surroundings. Different experiments have been performed in simulation and have been implemented on multiple robot systems [5]. Actual implementation has been on a smaller scale due to cost constraints. Therefore, simulation plays an invaluable and important role in this type of research. Different algorithms may be tested in simulation before implementation on a multi-agent system.

Multiple Robot Positioning and Communication

This research focuses on a scenario where a mobile robot will operate in an obstacle-strewn environment and must maintain line-of-sight communication with the operator control unit. This line-of-sight communication will be facilitated by using mobile communication repeater robots to chain together a total line-of-sight path from the mobile robot to the operator control unit. The objective of this research is to determine where to best position the communication repeater robots to maximize the area of the region in which the mobile robot can operate while maintaining communication to the operator control unit. Positioning and placement of multiple communications repeater robots depends on several factors: the number of robots to be positioned, the size of the

search space (map) and location of obstacles within the map, and communication. While maximum coverage of the map is desired it is also necessary that the communication repeater robots maintain line-of-sight communication among one another. Optimal positioning of all robots requires maximizing coverage while maintaining line-of-sight communication. When positioning the robots they must be able to communicate to at least one additional robot. Lastly, the scope of this research is limited to a two-dimensional case where the region of operation and the obstacles can be represented by polygons.

One area of concern involving this type of simulation problem is the need to have a general approach such that it can apply to different maps, a different number of robots and different types (shapes) of obstacles. This problem is addressed by modeling the map using a grid-based approach. The map is discretized into grid points and the obstacles within the map are represented by a series of grid points. The robots can be placed at any of these grid points as long as they do not coincide with an obstacle. This method eliminates the need for calculating complex areas and provides a simplified technique for representing the desired map.

Optimization

Another area of concern is the how to determine the position of all robots such that they satisfy the stated requirements. This is an optimal placement problem and can be solved using many different techniques and an overview of these techniques is presented in Chapter 3. There are several variables in this problem: number of robots, size of the search space (map), and number of obstacles in the map. Due to this variability, genetic algorithms will be used to determine optimal robot positioning.

Genetic algorithms are directed, randomized search techniques that are conceptually analogous to the process of natural selection in nature. A genetic algorithm library, GAlib [12], will be implemented to perform reproduction, crossover, and mutation. The initial population will be selected randomly and will consist of the desired number of robots and their corresponding positions. Possible solutions are represented by fixed-length strings of bits. The best-fit individuals are determined from a fitness function and they are selected for reproduction to form the next generation. The fitness function measures how well the map is covered by the current robot configuration. Mutation is introduced by flipping a bit in a randomly chosen string. This process is repeated in succession until an acceptable solution is obtained.

This research will use a genetic algorithm to determine the optimal robot positions for a given map. It should be noted that while genetic algorithms provide a method to finding a solution to a particular problem, it might not always be the globally 'optimal' solution. The benefit of using genetic algorithms over other optimization techniques is that the algorithm will always converge to a solution albeit not always optimal, and will converge in a relatively short time for a fairly large search space. Traditional optimization techniques require a well-defined function that can be optimized and convergence can take time, but in most instances the optimal solution will be found.

This software will allow the user to select the number of robots and a map. The program interprets the map and then proceeds to minimize the combined shadow area or unseen area. This area is the cumulative area that cannot be seen by any of the robots for the current robot configuration. Minimization of the shadow area allows for the maximum map coverage. The algorithm will terminate when it obtains the optimal

solution, a shadow area of zero, or after it cycles through a specified number of generations. A graphical user interface will be used to display the solution and the map along with the shadow area, if any. Additional statistics are saved to a data file for analysis.

CHAPTER 2
MODELING THE PROBLEM

There were several areas that were addressed before programming the simulation commenced. It was decided that the Visual C++ programming package would be used for all programs and graphics. The next step was to determine how to represent a map and obstacles contained within a computer program. Another element of the problem was determining if line-of-sight is maintained between at least two robots. Finally, there is the issue of determining the combined robot shadow areas, i.e. the region where line-of-site communication is not maintained. All of these issues will be addressed in detail throughout this chapter.

It was desired to be able to represent this problem in the most general manner. The generality will allow the algorithm to scale easily to account for larger maps or a greater number of robots.

A Map of Points

A significant amount of effort has been put into how to represent a map and the obstacles contained within. Assuming that the boundary edges of the map are known and the obstacles are known, the initial step was calculating the area of the obstacles and later, the shadow regions.

Polygon Triangulation Approach

The initial approach that was considered was to determine the exact area of each obstacle and the corresponding shadow area using polygon triangulation. This involves breaking a polygon into triangular sections, determining the area of each triangle, and then summing the areas [9]. This method is straightforward when the polygon is convex (see figure 2.1). When dealing with a non-convex polygon a negative area must be calculated and then subtracted from the surrounding boundary (see figure 2.2). This method provides a very reliable estimation of the areas but became computationally intensive when calculating the shadow areas.

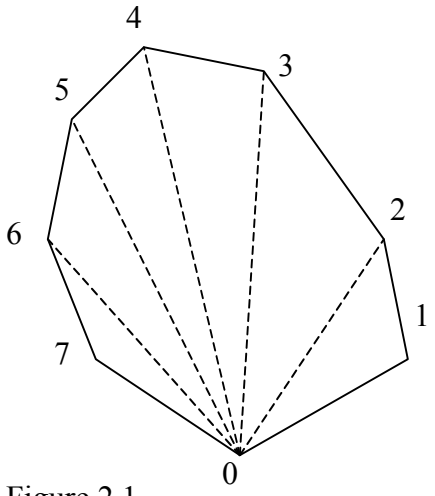

Figure 2.1

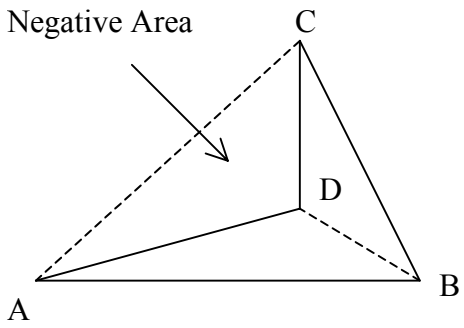

Figure 2.2

Figure 2.1: Triangulation of a convex polygon

Figure 2.2: Triangulation of a non-convex polygon. The area ADC is negative and is subtracted from area ABC.

Grid Based Approach

This method simplifies the map to a grid of M × N points. The obstacles contained within the map can be modeled by a matrix of 1's and 0's. All the grid points that make up an obstacle are represented by a '1' while empty space is indicated as '0'.

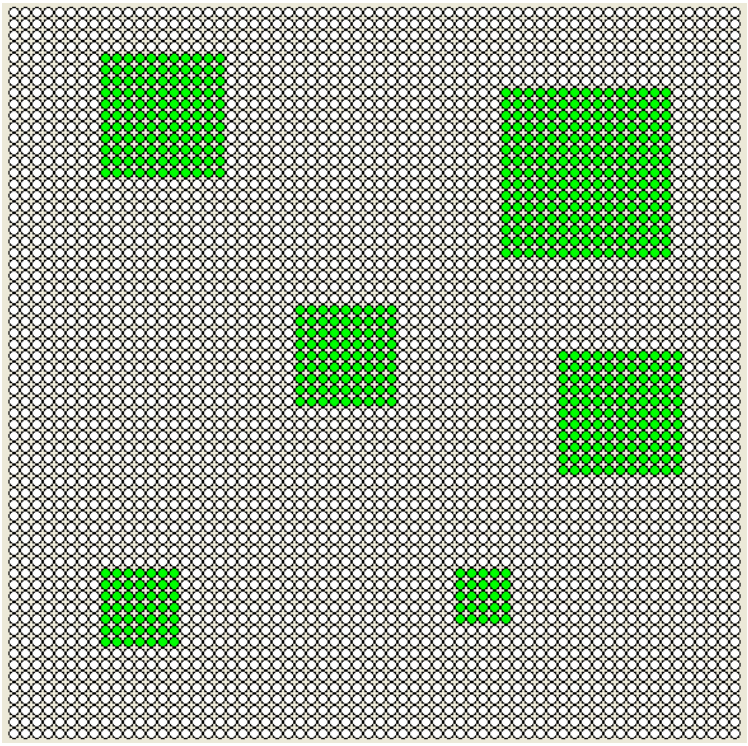

Figure 2.3: A sample 64×64 resolution map

This simplification greatly reduces area calculations and eliminates the need to differentiate between convex and non-convex polygons. A sample map is shown in figure 2.3. The map information is stored in a file that contains the points (x,y) that make up the obstacles. The program reads this file and is able to display the map to the screen graphically. An added benefit this method provides is the ease in displaying the graphics to the application window. A matrix can represent the map in figure 2.3 and the robots can be placed at points within the matrix, other than those occupied by obstacles.

Initial Line-of-Sight Method

Robot communication is an area that is essential to robot development. Whether robots are communicating with a human operator or amongst themselves, there are limitations. Much of the time the limiting factor is that the robot must maintain line-of-sight to be able to communicate. Line-of-site communications is often a mission

requirement for security of command and control and for this reason recent advances in communications such as wireless Ethernet that are typically omni-directional are not considered.

This work is based on the premise that the robots must maintain line-of-sight in order to communicate with a base station or another robotic entity. Therefore, initial steps were taken to develop an algorithm that determines if line-of-sight exists between two robots.

Line of sight is determined for two robots whose positions are known. Simply put, it is necessary to determine if a line segment drawn between the two robots intersects any of the known obstacles. With the grid-based approach, this can be done rapidly by starting at robot 1's position and then moving to the next adjacent grid position that moves closer to robot 2's position. A start angle (q_{start}) is calculated based on the (x,y) positions of the two robots. The current angle ($q_{current}$) is initialized to the start angle and the current position (x_c,y_c) is initialized to the position of robot 1. The value of the

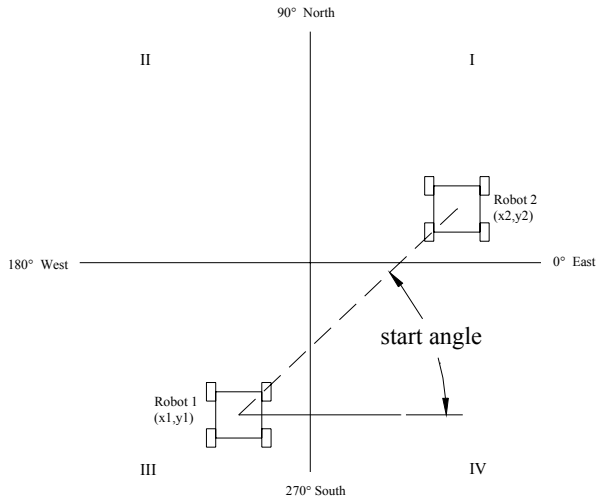

Figure 2.4: Illustration of line-of-sight method

starting angle will determine which direction to traverse, north, south, east or west (see figure 2.4). This method will determine if there is an unobstructed path between the two robots.

The algorithm divides the map into four quadrants. For example, if $q_{current}$ is between 0° and 90° (quadrant I) and less than q_{start} then x_c will be incremented by one, indicating a move due east. Otherwise, y_c is incremented by one, indicating a move due north. The algorithm proceeds in a similar fashion when $q_{current}$ is in a different quadrant of the map. The algorithm is terminated when x_c or y_c is equal to the x or y position of robot two.

Another matrix is created that is the same size as the map that contains the line-of-sight information. A '1' indicates the position of robot 1 and robot 2 in the matrix. If line-of-sight exists all intermediate points have a value of '1' in the matrix as well. The rest of the points have a value of '0'.

This method is beneficial because it minimizes the number of calculations. Instead of having to scan all of the points in the map, it is only necessary to check the points between the two robots in question. A preliminary graphical user interface was created to demonstrate the line-of-sight algorithm (see figure 2.5). The user is able to enter the locations of robot 1 and robot 2. The squares indicate the robot positions and the collection of points represent obstacles. The points connecting the two robots are filled in to indicate the line-of-sight path. This method was initially tested on a simple 20×20 grid but is easily applicable to larger areas.

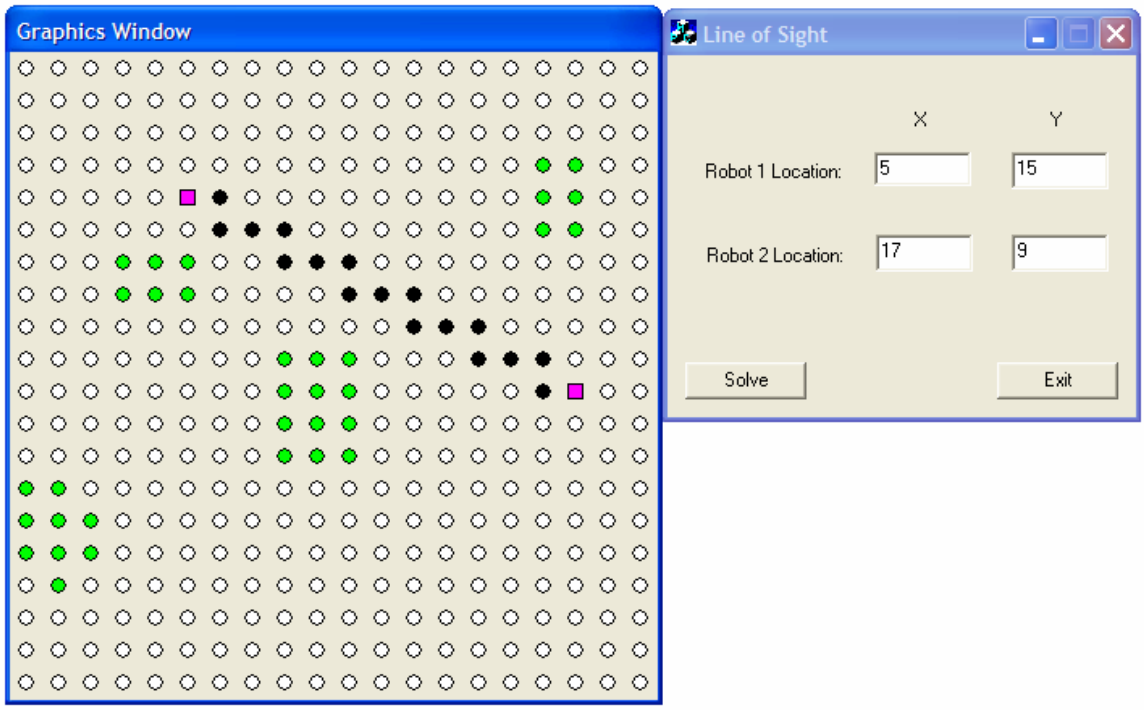

Figure 2.5: Preliminary implementation of the line-of-sight algorithm

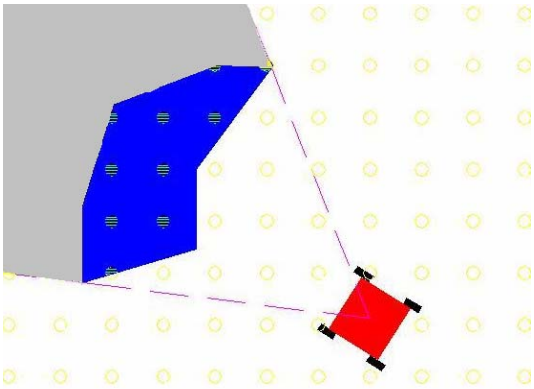

Figure 2.6: The shadow area is the grey area behind the obstacle.

Initial Shadow Area Calculation

Determining the shadow areas posed a difficult problem because the area *behind* an obstacle is being calculated. The first issue was trying to formulate the area to be calculated (see figure 2.6). The polygon triangulation method would add complexity to the problem because a polygon would first have to be created using projected lines and the boundary of the map. In contrast, the grid method allows for easy area calculation.

Once the shadow area has been determined the grid points designated as ‘shadow’ are simply summed together.

The technique used to determine the shadow areas is borrowed from the line-of-sight method discussed earlier. A robot is placed at a specified grid point and the remaining grid points are checked to see if line-of-sight exists. If line-of-sight does not exist the grid point is in a shadow region. Once again, a matrix is developed that contains all of the shadow points for each robot.

The overall shadow area can be determined by combining the data for each robot into a single matrix. This is done using the Boolean logic operator ‘AND’ on the matrices. This results in a single matrix that contains the shadow area information for a specific map and robot configuration. This method has been applied to a two-robot system in a 20×20 grid (see figure 2.7). More results are presented in Chapter 5.

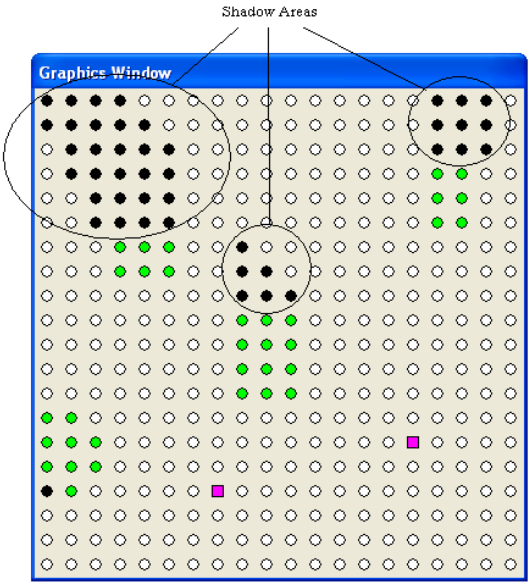

Figure 2.7: Resulting shadow areas for a simple two robot system (20×20 grid)

This method was applied to a single robot on a 50×50 grid in order to visualize where the maximum shadow areas occur. The robot was placed at every point in the matrix and

the corresponding total shadow areas were calculated at each point. The shadow areas were saved to a data file and later analyzed using MATLAB. Three- dimensional plots were created (see figure 2.8) that contain the x and y locations of the robot and the corresponding total shadow area. These plots are used to visualize where the maximum and minimum shadow areas occur. Points occupied by an obstacle have a maximum shadow area, which assumes the entire map is a shadow region.

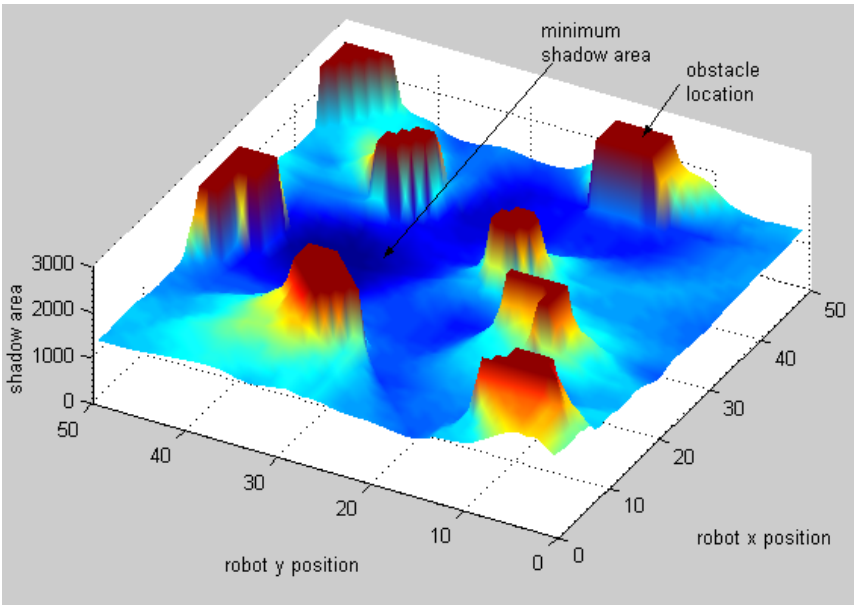

Figure 2.8: A plot of the shadow areas for a single robot placed at every point on the map (except obstacle locations)

Although this method provides a reasonable estimate of the shadow areas, it takes approximately seven minutes to determine a solution for a single robot. Optimization of this method is imperative once multiple robots are introduced.

Line-of-Sight and Shadow Area Method Revisited

After experimenting with the line-of-sight method mentioned earlier, it was decided that this method proved to be entirely too slow for this application. Due to the nature in which the line-of-sight function is used, the speed to determine a solution is of utmost

importance especially when various robots are considered. The line-of-sight method and thus, the shadow area method had to be revisited for further optimization.

Line Segment Intersection Method

A method was devised that accounts for the line segments that comprise the obstacles contained within the map. Therefore, a file containing all of the obstacle line segments was created. Using the obstacle line segments, the current robot position, and another arbitrary point or robot position it could be determined if there was an unobstructed path between the robot and the point in question. At this point, it was necessary to write a function that could determine if two lines intersect.

Further research revealed a function written by Mukesh Prasad [1] that determines if two line segments intersect. A total of four (x,y) pairs are passed into the lines_intersect function. The first two pairs (x_1, y_1, x_2, y_2) represent the first line segment and the second two pairs represent the second line segment (x_3, y_3, x_4, y_4). The following line equation is used in the function:

$$a_1 x + b_1 y + c_1 = 0 \tag{2.1}$$

The function proceeds to calculate the coefficients of the line equation, a_1, b_1, and c_1.

$$a_1 = y_2 - y_1 \tag{2.2}$$

$$b_1 = x_1 - x_2 \tag{2.3}$$

$$c_1 = x_2 \times y_1 - x_1 \times y_2 \tag{2.4}$$

The function then performs a calculation that determines if the endpoints of the second line segment, (x_3,y_3) and (x_4,y_4), lie on the same side of the first line segment. If this is true, the line segments do not intersect. If this is false, a calculation is performed which determines if the endpoints of the first line segment, (x_1,y_1) and (x_2,y_2), lie on the same

side of the second line segment. If this is true, the line segments do not intersect and if this is false, the line segments do intersect. This is achieved with a simple sign check. The x and y values of point three (x_3,y_3) are placed into the line equation (eqn. 2.1) joining points one and two. The same is done for the fourth point. The equations below are used:

$$a_1 x_3 + b_1 y_3 + c_1 = r_3 \qquad (2.5)$$

$$a_1 x_4 + b_1 y_4 + c_1 = r_4 \qquad (2.6)$$

For example, if the signs of r_3 and r_4 are the same then the two line segments in question do not intersect. This means that point three and point four lie on the same side of the first line segment. The original function returns the actual point of intersection. Modifications were made to the function to return a value of '1' if the segments do intersect and a '0' if they do not intersect.

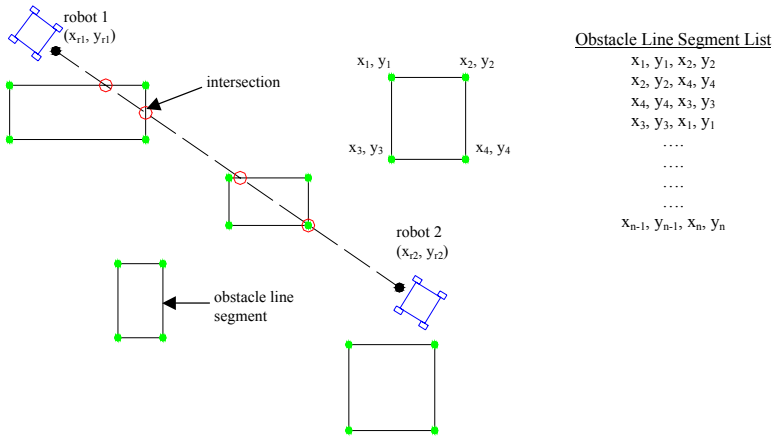

Figure 2.9: Illustration of the 'lines_intersect' method - A file contains a list of all line segments that make up each obstacle in the map.

Line-of-Sight Determination

For this particular problem the line-of-sight criteria is defined as a single robot being able to see *at least* one other robot. Line-of-sight among a pair of robots can be determined using the lines_intersect function explained previously. Assuming that the

positions for two robots are known, it can be determined if the path between the two robots is obstructed. This is accomplished by checking for intersection between the robot path and all obstacle line segments (see figure 2.9).

Shadow Area Calculation

The shadow area is calculated by calling the lines_intersect function to check for possible intersection between the robot path and all obstacle line segments. The shadow area function checks a single robot and an arbitrary point for possible intersection with the obstacle line segments. If intersection exists, a matrix containing the combined shadow area is set to '1' at that arbitrary point. This is done for all robots and the combined shadow area is calculated simply by adding all of the cells set to '1' in the shadow area matrix.

The final algorithm proceeds to place the robots at grid points in the map and then calculate the corresponding shadow area. However, a problem arises when a grid point that lies within an obstacle is chosen. If a grid point contained within an obstacle is selected, the algorithm will return the maximum shadow area for that particular robot position, i.e. the total number of grid points for that particular map.

Preliminary results

These methods have been applied to the single robot case in which a robot is placed at a point and the shadow area is calculated for that particular position. After implementing the lines_intersect function for the single robot case all possible shadow areas are calculated for a 50×50 grid in approximately 18 seconds. This is a vast improvement from the previous method, which took seven minutes.

An exhaustive search technique was applied to the two-robot case. Every possible two-robot configuration was checked for the line-of-sight criteria and the corresponding

combined shadow area was calculated, again for a 50×50 grid. This entire process took approximately 11 hours to complete. From this preliminary experiment it can be learned that the exhaustive search technique is not ideal for this type of problem. The next chapter will discuss how a genetic algorithm search approach can be used to significantly decrease the time required to obtain an optimal placement for multiple communication repeater robots to minimize the resulting shadow area.

CHAPTER 3
AN OVERVIEW OF OPTIMIZATION TECHNIQUES

This goal of this chapter is to discuss different types of optimization techniques and to provide examples and applications of the techniques. A general overview of these techniques provides insight into the nature of this particular problem and the thought process involved to arrive at the final solution. This chapter is by no means inclusive of all optimization techniques but provides a general discussion of some of the techniques used in industry and academia.

Background

Optimization techniques are used throughout industry and applications include designing aircraft wings, economic analyses, airline scheduling, and internet search engines to name a few. Much of the optimization techniques in use today can trace their origins back to World War II. During this time any technique that increased the efficiency of the war effort was extremely important especially when faced with a limited number of people, machines, and supplies [2]. Due to this need, large-scale optimization techniques were developed to deal with the massive logistical problems raised when having to organize millions of troops and their associated gear and weaponry.

During this time all calculations were done manually. With the invention of the digital computer, the speed at which calculations are performed and the amount of calculations possible increased dramatically. The digital computer revolutionized the field of practical optimization.

Golden Section Method

The golden section method can be used for estimating the maximum, minimum, or zero of a one-variable function. The function is assumed to uni-modal, but does not necessarily have to have continuous derivatives. In contrast to polynomial curve fitting techniques, the rate of convergence for the golden section method can be determined. This method is reliable for poorly conditioned problems but a well-conditioned problem will not converge any more rapidly.

The golden section recurs in nature as an aesthetic ratio and throughout history as a number to which mysterious properties were attributed [11]. This ratio is the ratio of the base to the height of the Great Pyramid. Leonardo da Vinci added to the popularity with his studies on proportions of the human body. He found that this ratio was the distance from the navel to the ground divided by the distance from the top of the head to the navel of the "optimum" human body (see figure 3.1).

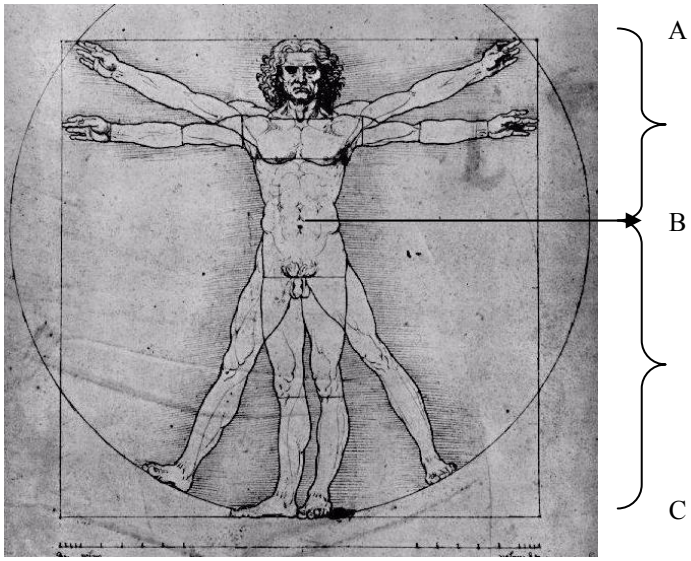

Figure 3.1: BC/AB = Golden Section Ratio

A function, F, is a function of the independent variable x. The value of x that will minimize F is to be determined. There are lower and upper bounds that bracket the minimum, X_L and X_U.

The function F has been evaluated at X_L and X_U and therefore the corresponding values of F_L and F_U are known.

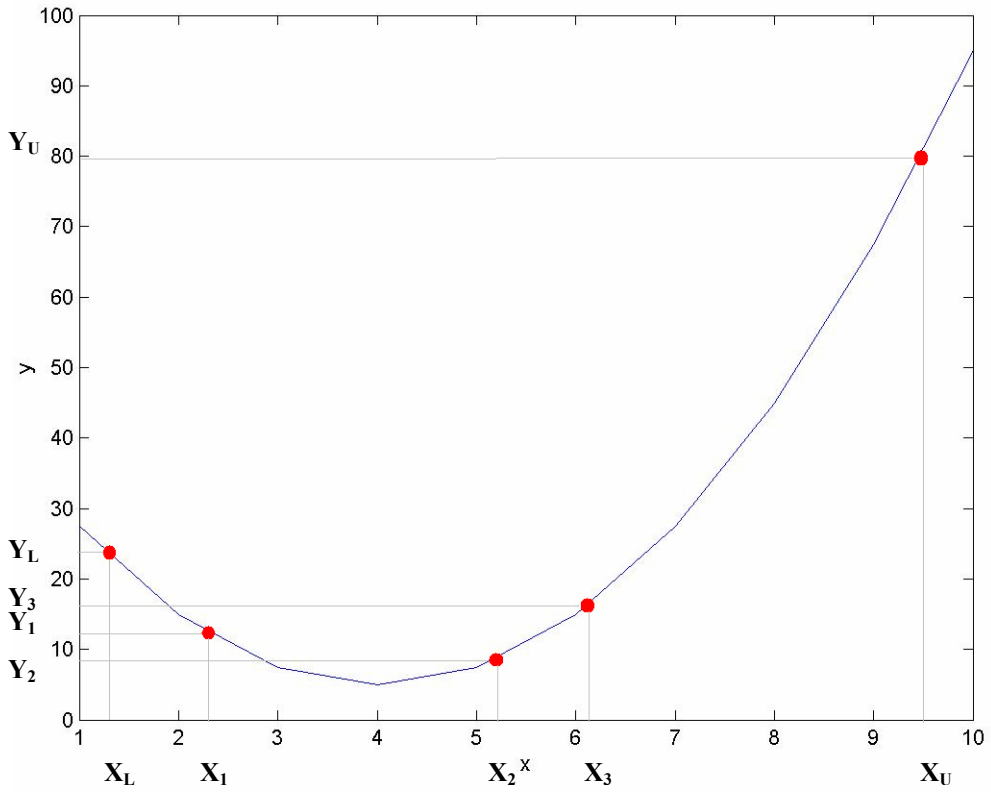

Figure 3.2: Illustration of the Golden Section Method

The golden section method proceeds by choosing two intermediate points between X_L and X_U (see figure 3.2). As stated earlier, the function is assumed to be uni-modal, therefore X_1 or X_2 will form a new bound on the minimum. Because F_1 is greater than F_2 it forms a new lower bound, which produces a new set of bounds X_1 and X_U. X_1 is the new lower bound and a new point X_3 is chosen and evaluated for F_3. When comparing F_2

and F_3, F_3 is greater and X_3 replaces X_U as the upper bound. This process continues iteratively until the bounds are narrowed to a desired tolerance.

For this particular method the golden section is used as a sequence for dividing the interval of the interior points to find the minimum value of the function, F, with as few function evaluations as possible. The golden section ratio is .38197 or 38%. This iterative procedure can be applied once a convergence criterion has been determined; this criterion will indicate when the process has converged to an acceptable solution.

An initial interval of uncertainty has been defined, as X_U-X_L and it may be desired to reduce the interval by a fraction, ε, of the initial interval or by a magnitude Δx. ε is referred to as a local tolerance and Δx as an absolute tolerance because it is independent of the initial interval. The relative tolerance can be defined as follows:

$$\varepsilon = \frac{\Delta x}{X_U - X_L} \qquad (3.1)$$

This tolerance, ε, can be converted to a maximum number of function evaluations in addition to the three required to evaluate F_L, F_1, and F_U. This can be done because the interval is reduced by the golden ratio, τ (38%), for each function evaluation. Therefore, for a specified tolerance, ε:

$$\varepsilon = (1-\tau)^{(N-3)} \qquad (3.2)$$

where N is the total number of evaluations. Solving the equation for N yields:

$$N = \frac{\ln \varepsilon}{\ln(1-\tau)} + 3 = -2.078 * \ln \varepsilon + 3 \qquad (3.3)$$

Choosing the value of ε is left to the programmer. For example, if it is desired to reduce the interval to 2% of the initial interval (ε=0.02) then N = 8.13+3 = 11.13$\rightarrow$12 function

evaluations. *N* can now be used as a convergence criterion. The iteration process is terminated when a total of *N* function evaluations have been performed. Note that *N* may take on fractional values and should be rounded up to the next higher integer.

Although the golden section algorithm has been developed to determine the minimum of a one-variable function, the maximum of a function can be determined simply by minimizing the negative of the function.

Gradient Techniques

The rate of convergence of an optimization technique for a particular problem is dependent upon the initial parameters selected. If poor initial parameters are chosen, the optimization technique converges very slowly and is rendered useless. Depending on the technique chosen the initial parameters will dictate the speed at which a solution is determined as well as the validity of the final solution.

A combination of optimization techniques may be applied initially. A slow algorithm may be tried and once approaching the minimum, switch to a technique that is much faster. Initially, a decision is made as to which direction to proceed. Then, a decision is made on how far to proceed in that direction. The latter, is determined by a set of rules and steepest descent is an example of this.

Steepest Descent

For any optimization procedure an initial guess $\mathbf{x}$ must be made for the parameter value. This guess can then be modified to $\mathbf{x}+\delta\mathbf{x}$ in order to reduce an error function, $E(\mathbf{x})$. Here $\mathbf{x}$ is a vector that contains the n design parameters and the objective is to find the values of these parameters that minimizes the objective function $F(\mathbf{x})$. The steepest descent technique chooses $\delta\mathbf{x}$ so as to be traveling in a "downhill" direction (i.e. direction

which reduces the error function), approaching a minimum. Due to the fact that $\delta\mathbf{x}$ is a vector it is described by two quantities: a magnitude and a direction. In order to choose the direction of $\delta\mathbf{x}$ the dot product is applied.

Definition of the dot product:

$$\boldsymbol{a} = (a_1, a_2, \ldots, a_n) \qquad \boldsymbol{b} = (b_1, b_2, \ldots, b_n) \tag{3.4}$$

$$\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}|\,|\boldsymbol{b}| \cos\theta \tag{3.5}$$

Where $|\boldsymbol{a}|$ and $|\boldsymbol{b}|$ is the magnitude of $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively, and θ is the angle between the two vectors. The dot product may be applied to the following relation:

$$E(\mathbf{x}+\delta\mathbf{x}) \approx E(\mathbf{x}) + \nabla E^T \delta\mathbf{x} \tag{3.6}$$

The last term in equation 3.6 may be written as:

$$\nabla E^T \delta\mathbf{x} = \frac{\partial E}{\partial x_1}\delta x_1 + \frac{\partial E}{\partial x_2}\delta x_2 + \ldots + \frac{\partial E}{\partial x_n}\delta x_n \tag{3.7}$$

This is the dot product of the gradient ∇E and the parameter change $\delta\mathbf{x}$. Therefore, a unit vector $\delta\mathbf{x}$ will produce a maximum change if it points in the direction of the gradient and a minimum (negative change) if it points in the negative direction of the gradient. The steepest descent method uses this information by choosing the $\delta\mathbf{x}$ to point in the negative gradient direction. The steepest descent technique proceeds as follows:

1. An initial set of parameters $\mathbf{x}$ is chosen, which results in an error $E(\mathbf{x})$.
2. The gradient ∇E is calculated at point $\mathbf{x}$. The parameter direction changes are chosen to be in the direction of the negative gradient:
   $\delta\mathbf{x} = -\alpha\nabla E$ (α is a positive constant) (3.8)
3. This process is iterated until the desired minimum is reached.

In an optimization problem it is usually desired to find a global minimum and avoid getting trapped in a local minimum. Failing to find the global minimum is not just a

difficulty associated with steepest descent; any optimization technique can be plagued by this problem [4].

Monte Carlo Integer Programming

Exhaustive search methods for solving optimization problems will produce the optimal solution, however, as the search space increases it is doubtful that the solution will be determined in a reasonable amount of time. A more efficient method might require checking only a sample of possible solutions and then selecting the best solution out of the sample population. Therefore, random samples of feasible solutions are chosen and the corresponding (maximum or minimum) best solution from the random sample is determined. This method is known as Monte Carlo integer programming. There is some concern regarding the 'goodness' of the answer obtained using this method mainly attributed to the random nature in which the samples are selected [3]. However, based on the statistics of the objective function, the random samples follow the probability distribution of the possible solutions for the integer-programming problem.

Suppose the following function is to be maximized:

$$P = x_1^{\,2} + x_2^{\,2} + 10x_3^{\,2} + 5x_4^{\,2} + 6x_5^{\,2} - 3x_1 - x_2 + 3x_3 - 2x_4 + x_5 \qquad (3.9)$$

where $0 \le x_i \le 99$ (i=1…5)

There are five variables in the above equation with values ranging from 0 to 99. This means there are a total of 100^5 possible points to check (10,000,000,000). Although computers are becoming faster, some computers may be burdened by this many calculations. The Monte Carlo technique will check a random sample of the 100^5 points for a feasible solution. For example, a random sample of one million points is evaluated and the point with the maximum value will be deemed the solution to the problem.

For the line-of-sight communications problem, the evaluation of the objective function is a time consuming process. For example, if n communicator repeater robots are to be positioned, there are 2n design variables, i.e. the x and y coordinates for each of the repeater vehicles. Evaluating the objective function for each possible set of robot positions requires calculating the total shadow area for this particular configuration, which is a lengthy process. Since for this application the partial derivatives of the objective function with respect to the design variables cannot be determined analytically, many optimization approaches such as the steepest descent method must be ruled out. For practical purposes, a random search algorithm must be implemented such as a random search direction method, the Monte Carlo search method, or a genetic algorithm approach method. This latter method was used in this problem and the next chapter will describe the genetic algorithm approach.

This randomized approach will produce a near optimal solution. Although the optimal solution is not necessarily obtained, the speed at which a solution is determined is greatly reduced.

CHAPTER 4
GENETIC ALGORITMS

Genetic algorithms are search algorithms based on the mechanics of natural selection and genetics. They efficiently use previous information to determine new search points with improved performance. John Holland and his colleagues developed this technique at the University of Michigan in 1975. The theme of much of the genetic algorithm research has been robustness and the balance between efficiency and the strength necessary for survival in a variety of environments. In sophisticated artificial systems, features common in biological systems such as self-repair, reproduction, and self-guidance barely exist. The following quote by John Holland captures the plight of many in the field of computer and design optimization:

> Living organisms are consummate problem solvers. They exhibit a versatility that puts the best computer programs to shame. This observation is especially galling for computer scientists, who may spend months or years of intellectual effort on an algorithm, whereas organisms who come by their abilities through the apparently undirected mechanism of evolution and natural selection. [7, p. 1]

Traditional Methods versus Genetic Algorithms

Calculus-based methods have been studied extensively and can be divided into two main classes: indirect and direct. The indirect methods seek local extrema by solving a set of (usually) non-linear equations resulting from setting the derivative or gradient of the objective function equal to zero. This method is extremely useful for a single-peak function as shown in figure 4.1.

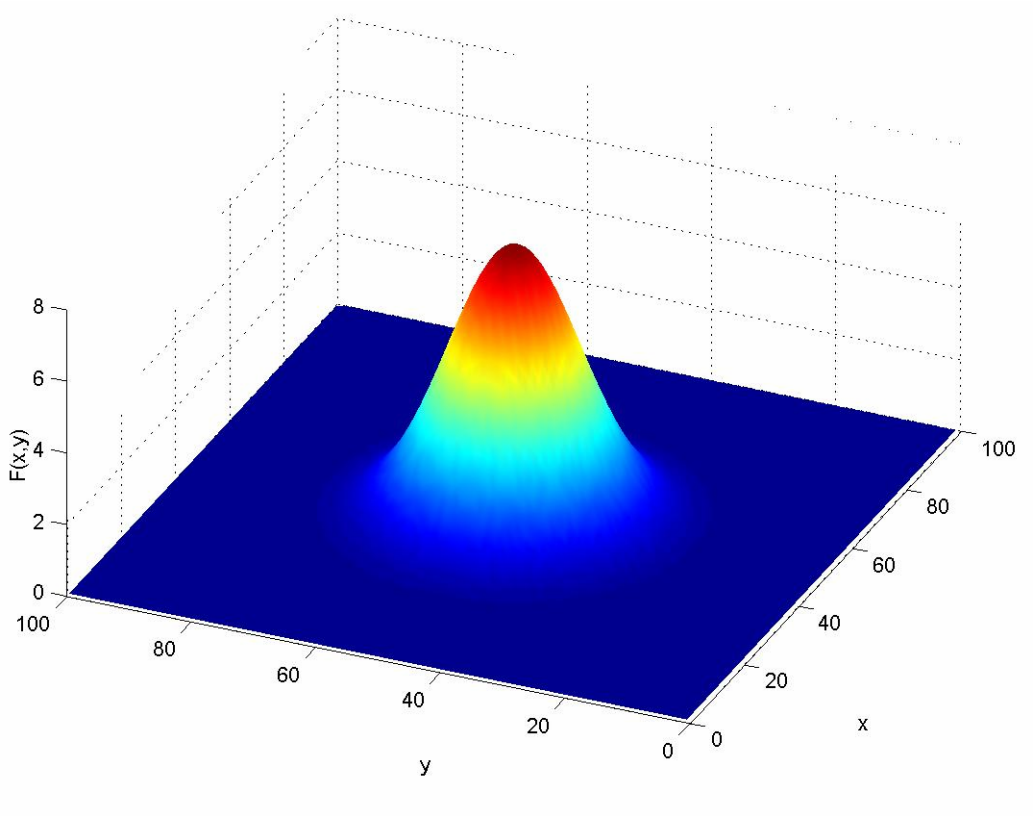

Figure 4.1: Single-peak function

Direct methods seek local optima by moving in a direction related to the local gradient. This is similar to the method of steepest-descent discussed in Chapter 3. Both methods are local in scope, that is, the optimum is the best in a neighborhood of the current point. In terms of the direct search methods the multiple peak function shown in figure 4.2 proves to be a bit of a challenge. In this case it would be difficult to determine which peak to climb and once an alleged solution is found it is not known whether it is a local or a global optimum. In addition, search and optimization problems can be multi-modal, noisy search spaces that can be rife with discontinuities. The calculus-based methods are practical for certain types of problems.

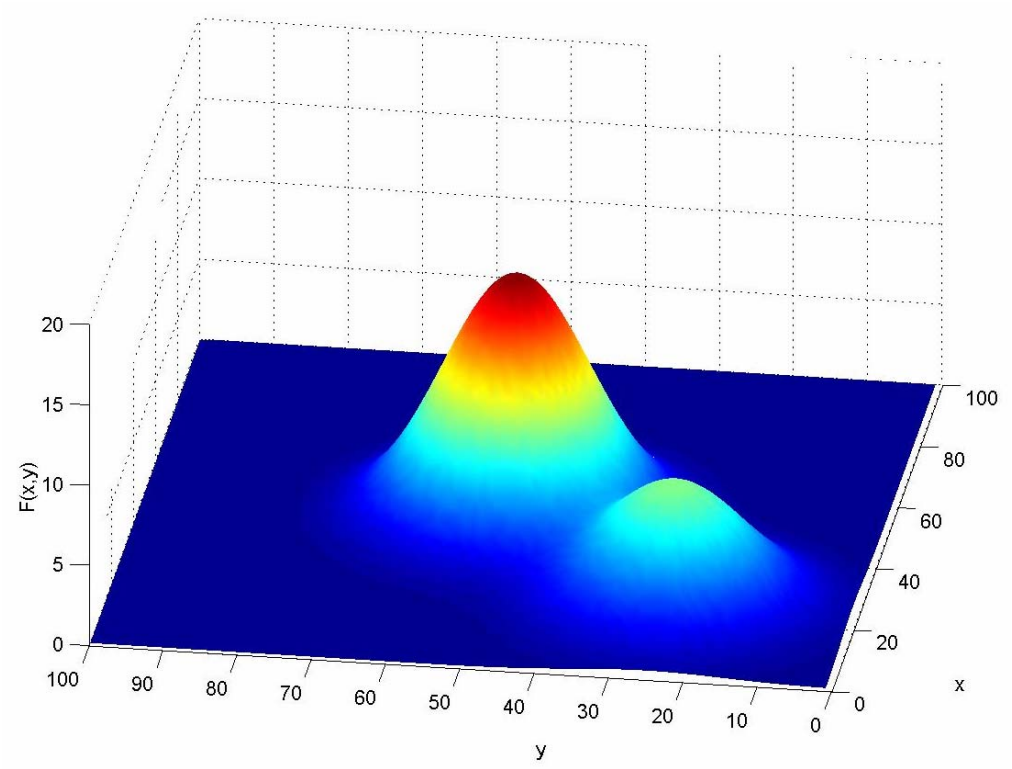

Figure 4.2: Double-peak function

Genetic algorithms require that the parameter set of the problem be coded as a finite length string. In addition, an objective function is necessary to measure how fit a current solution is compared to the rest of a population. The search is guided using probability and random choice. As the generations pass, strings associated with improved performance will predominate and as the mating process progresses strings are combined in new ways generating more sophisticated solutions [6].

Simple Genetic Algorithm

A simple genetic algorithm involves complex copying of strings and swapping of partial strings. An initial population is chosen, usually at random. It is often necessary to define a set of simple operations that take this initial population and generate successive populations that improve over time.

Reproduction

Reproduction is the first step of the genetic algorithm. During this process individual strings are copied according to their objective function values, or fitness values. Copying strings according to this measure means that strings with higher fitness values have a greater probability of propagating through the next generation. This is an artificial version of "natural selection." In terms of an algorithm, reproduction may be implemented in several ways. Goldberg creates a biased roulette wheel where each current string in the population has a slot proportional to its fitness [6]. An illustration of this idea is shown in figure 4.3.

String #	String	Fitness	% of Total
1	11011	385	45.9
2	10001	141	16.8
3	00110	62	7.4
4	10101	251	29.9
Total		839	100

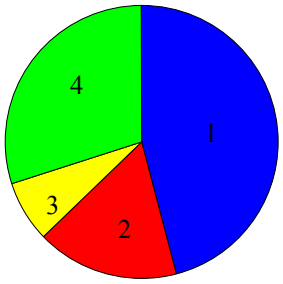

Figure 4.3: Sample string types and biased roulette wheel

A simple spin of the weighted roulette wheel yields the reproduction candidate, which is solely based on fitness values and the overall probabilities. Using this method, more highly fit strings have a greater number of offspring in the succeeding generation. Once a string has been selected for reproduction an exact replica of the string is created and it is then entered into the mating pool, resulting in a tentative new population.

Crossover

After reproduction, simple crossover proceeds in two steps. Members of the new population are mated at random. Next, each pair of strings undergoes crossover: a position *i* along the string is selected at random between 1 and then string length (L)

minus one [1, L-1]. Swapping all characters between *i*+1 and L forms two new strings. For example:

S_1 = 1 1 0 | 1 1

S_2 = 1 0 0 | 0 1

Note: The | character indicates the crossover site.

i is chosen at random to be 3. After crossover two new strings are formed which are part of the new generation. The new strings are as follows:

S_1' = 1 1 0 0 1

S_2' = 1 0 0 1 1

Mutation

Mutation is necessary because, even though reproduction and crossover effectively search and recombine, there are occasions when there can be a loss of genetic material. In addition, mutation adds extra variability to the existing population. "Mutation alone does not generally advance the search for a solution, but it does provide insurance against the development of a uniform population incapable of further evolution" [7, p. 3].

In the simple genetic algorithm, mutation usually has a very small probability and simply corresponds to a random alteration of the value of a string position. Because mutation rates are low, it can be considered a secondary mechanism of genetic algorithm adaptation [6].

Genetic Algorithm Implementation

A significant portion of this research was devoted to developing a method to measure how well the current robot configuration covered the region to minimize the shadow area while making sure that every communications repeater robot was within line-of-sight of another. In order to gauge the results obtained from the genetic algorithm it was necessary to develop an efficient and robust objective function. In particular, the

combined shadow area - area that is not visible to any of the robots - was measured for a robot configuration using this objective function. The objective function not only takes into account the shadow area of each robot but it also has to incorporate the line-of-sight criteria. It was decided that line-of-sight communication was maintained as long as there was an unobstructed path between the robot in question and at least one other robot.

After a solid objective function was developed, a string representation of the robot positions was developed. It can be noted that the map resolution correlates directly to the number of bits used in the binary string. For example, for the 64×64 map, there are a total of 4096 data points. Therefore, in order to represent all of the corresponding x and y robot positions using a binary string there had to be a total of 12 bits ($2^{12} = 4096$). The most significant six bits represent the robot's x position and least significant six bits represent the robot's y position (see figure 4.4). The position of each repeater robot is represented using the binary string representation.

X_1	X_2	X_3	X_4	X_5	X_6	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6

Figure 4.4: Binary String Representation of Robot's (x,y) Position

A series of maps were generated with greater resolution in order to demonstrate the expandability of the method and the versatility of the genetic algorithm. These maps had a resolution of 256×256, which corresponds to 65,536 data points – a considerably larger search space than the aforementioned. Due to the increased number of data points, the number of bits in the string was increased to 16 bits ($2^{16} = 65,536$). A similar binary string as that in figure 4.4 was used with the addition of two more bits to both the x and y component of the string.

The software for this work used the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology [12]. This library required an

objective function and a solution that could be represented in a single data structure. The library is broken into two primary classes: a genome and a genetic algorithm. Each genome represents a single solution to the problem. The actual genetic algorithm defines how the evolution should take place. The genetic algorithm uses the objective function to determine how 'fit' each genome is for survival. Genome operators, which are built into the genome, and selection strategies, which are built into the genetic algorithm, are used to generate new individuals.

This library incorporates several different types of genetic algorithms such as: simple, steady state, and incremental. The simple genetic algorithm proceeds to create an entirely new population of individuals every generation and automatically advances the best individual to the next generation. The steady state genetic algorithm uses an overlapping population. That is, the user is allowed to specify how much of the population may be replaced in the next generation. When using an incremental genetic algorithm, the next generation consists of only one or two children. In addition, methods are defined for how the new generation should be integrated into the overall population.

For purposes of this research, the simple genetic algorithm was chosen because previous work could easily be adapted to the format required by the library. The diagram in figure 4.5 illustrates the overall program flow in terms of the genetic algorithm. The next chapter will present the results of applying this algorithm to the problem at hand. Several test cases will be examined and discussed.

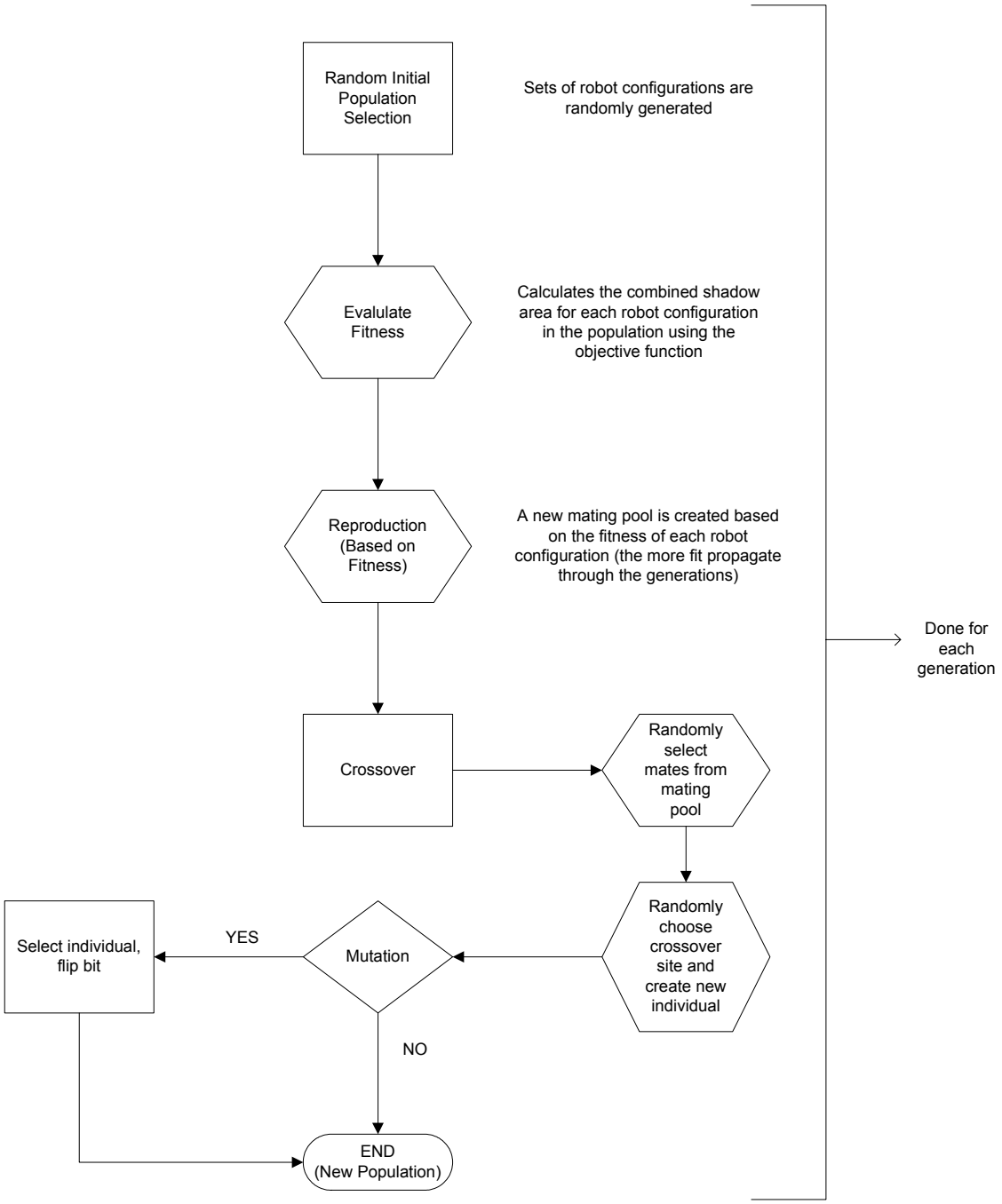

Figure 4.5: Genetic Algorithm Flow Diagram

CHAPTER 5
EXPERIMENTS AND RESULTS

This chapter describes the experiments that were run to test the feasibility of the multiple vehicle positioning simulation program. A series of maps were created at two different resolutions. There were a total of twelve maps, half of which were of 64×64 resolution and the other half were of 256×256 resolution. For both types of maps the algorithm was run for a two, three, four, and five communications repeater robot configuration. The results were saved to a file and displayed visually. This chapter has a section for each of the different resolutions and subsections within containing results from the individual maps. In addition, there is a section that compares the genetic algorithm technique to an exhaustive search technique and presents the corresponding results. All maps were generated using a map program that was written in C. All measurements are in units of pixels and can be adjusted to fit the needs of the user..

Graphical User Interface

The graphical user interface (GUI) was created using the Visual C++ programming package. It should be noted that the GUI program only displays the 64×64 resolution maps. It was necessary to create a GUI that was visually appealing as well as user-friendly. Some other features of the program are that the user is able to choose from six different maps, enter the desired number of robots, view the corresponding robot positions, and visualize the resulting shadow area. The total processing time is displayed as well. A sample of the GUI is shown in figure 5.1.

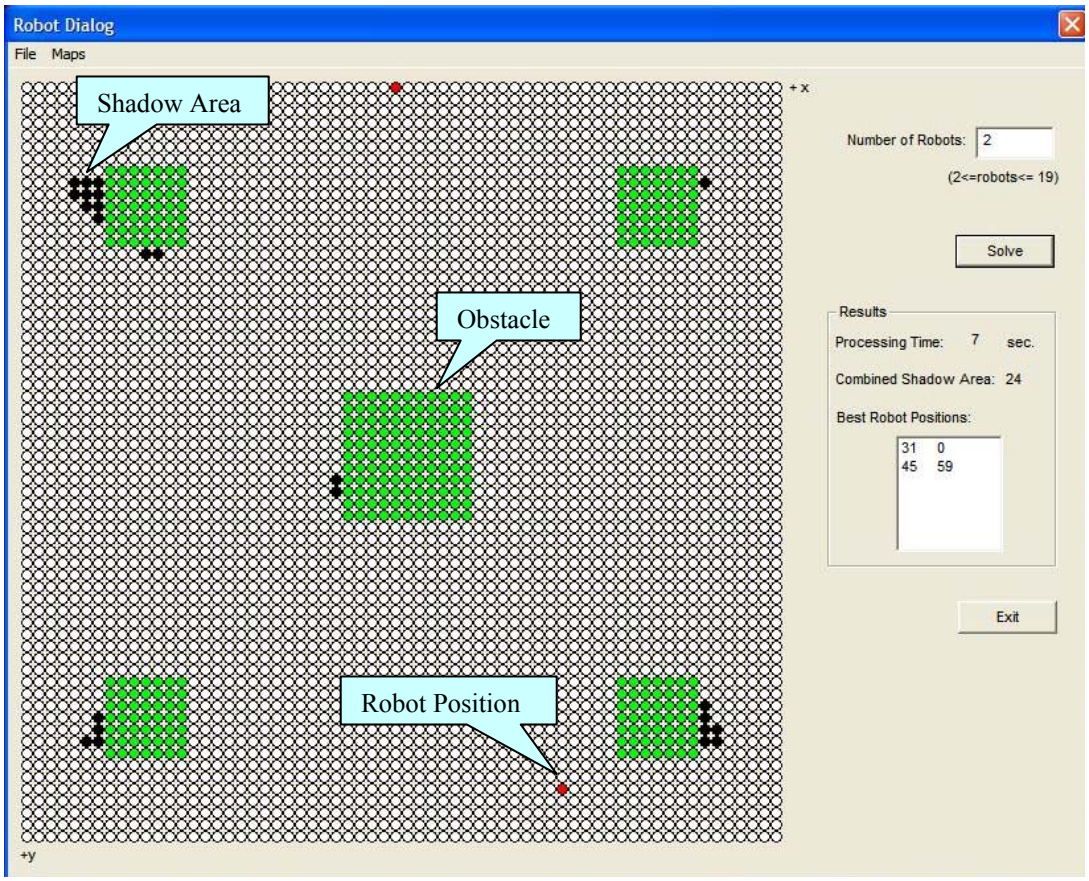

Figure 5.1: Sample GUI written in VC++

64×64 Resolution Maps

Map One

The first map that was analyzed was somewhat simplistic and was initially used for testing the algorithm. This map was created manually, simply by creating a file that contained all of the points that make up each obstacle. This file is then read into the optimization program and positioning of the desired number of robots proceeds using this map. For all maps of 64×64 resolution the convergence criterion was specified to be the best solution after 100 generations.

The genetic algorithm is constantly trying to improve the results obtained from the objective function by combining the most fit offspring of each generation. For this map, as more robots are added to the solution, the better the overall coverage of the map.

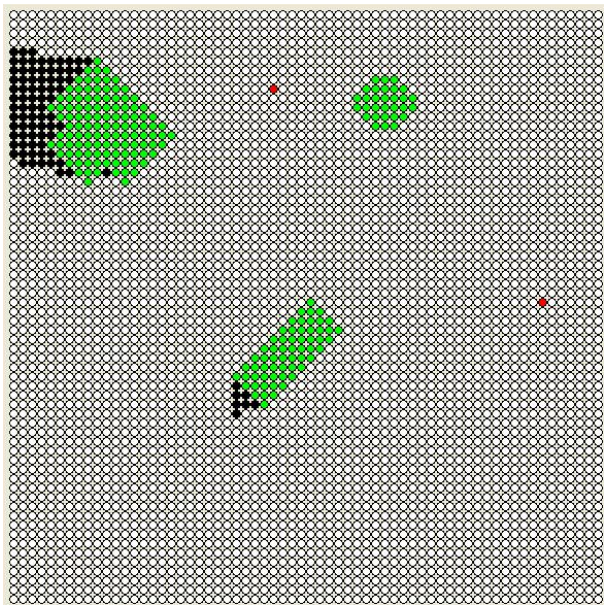

Figure 5.2a: Map 1, 2 Robots,
Shadow Area = 82

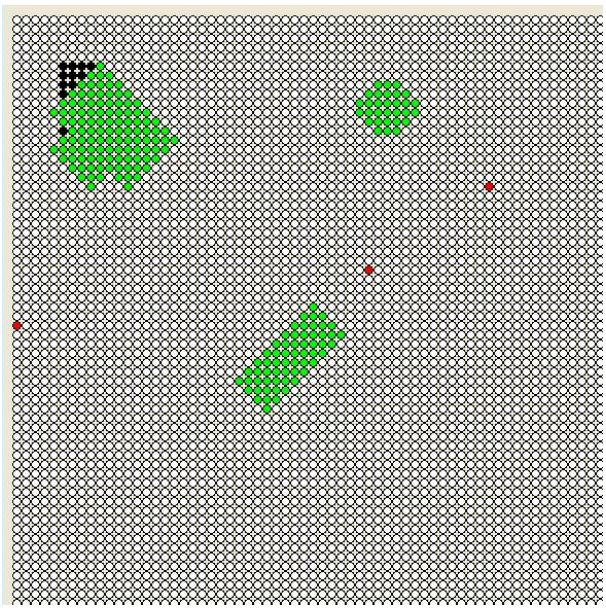

Figure 5.2b: Map 1, 3 Robots,
Shadow Area = 11

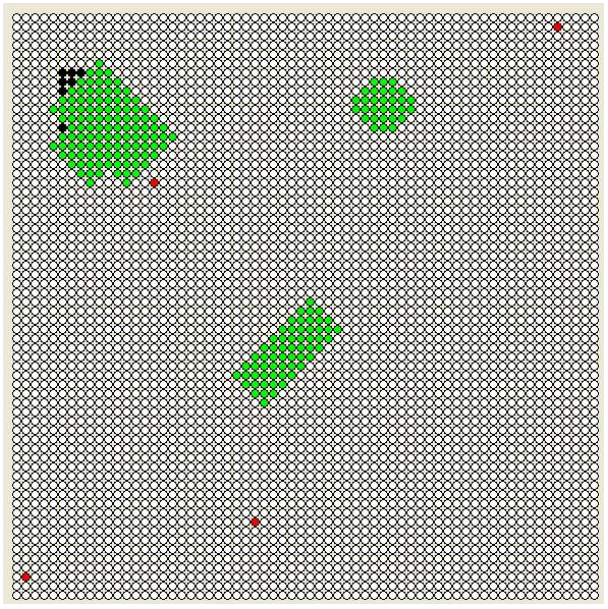

Figure 5.2c: Map 1, 4 Robots,
Shadow Area = 7

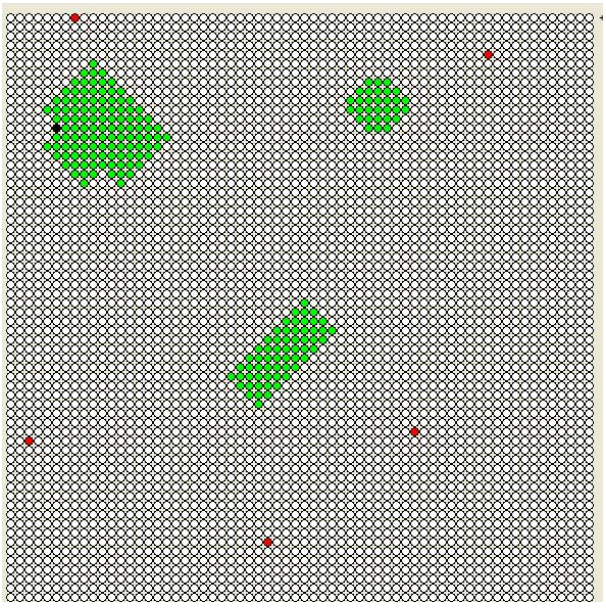

Figure 5.2d: Map 1, 5 Robots,
Shadow Area = 1

Figure 5.2: Results for Map1

The results from map one are relatively encouraging. As more robots are added to the solution, there is a decrease in the shadow area, which implies that the coverage of the map is improved. Figures 5.2a-d depicts the final results for that particular map with two through five robots. A plot of the shadow areas versus the processing time is shown in figure 5.3. As more robots are added to the problem, the processing time increases and the shadow area decreases. A solution is obtained for five robots in a time of 13 seconds. As the complexity of the map increases the processing time increases slightly, by a matter of seconds. The algorithm converges when it cycles through all of the 100 generations.

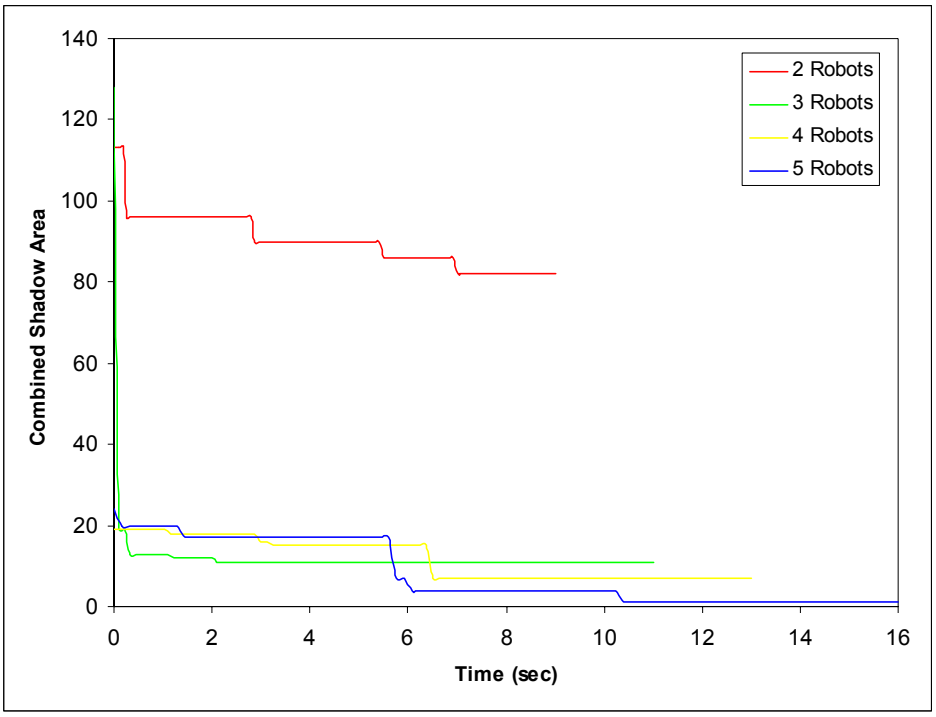

Figure 5.3: Map 1 convergence rates for 2-5 robots after 100 generations

Map Two

This map is slightly more complicated than map one, in that it has more obstacles. This map was generated using a simple program that reads in a file containing

the number of obstacles, the coordinates for the center points and the corresponding radius of each obstacle. This method limited the shape of the obstacles to being square yet overlapping several obstacles could create other shapes. The results for map two are shown in figure 5.4.

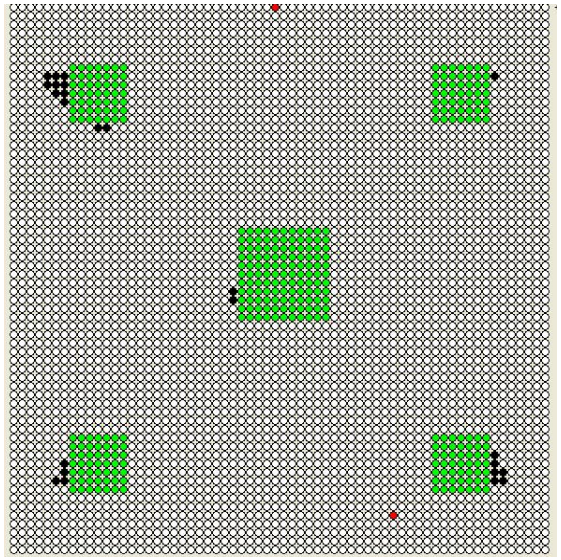

Figure 5.4a: Map 2, 2 Robots,
Shadow Area = 24

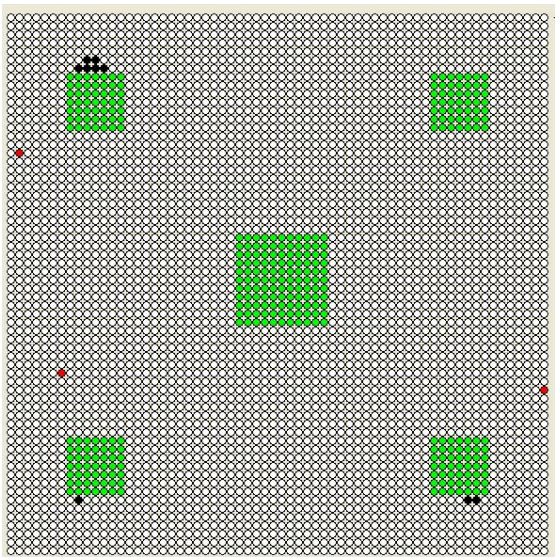

Figure 5.4b: Map 2, 3 Robots,
Shadow Area = 9

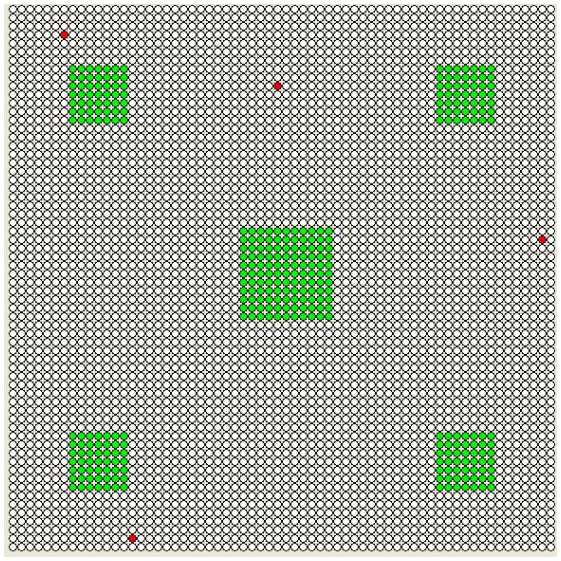

Figure 5.4c: Map 2, 4 Robots,
Shadow Area = 0

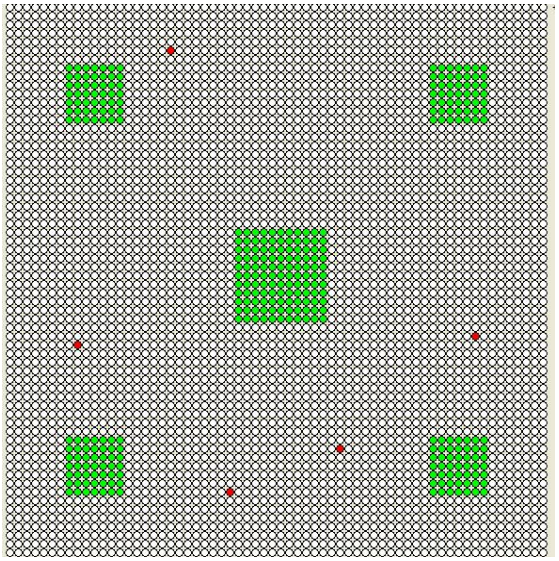

Figure 5.4d: Map 2, 5 Robots,
Shadow Area =0

Figure 5.4: Results for Map 2

The four-robot configuration seems to be the optimal situation with a shadow area of zero. The five-robot configuration results in a shadow area of zero also, which is an intuitive result. As more robots are added to the solution, the better the coverage of the search space. The algorithm converged after cycling through 100 generations of the genetic algorithm. Plots of the shadow areas versus time are shown in figure 5.5.

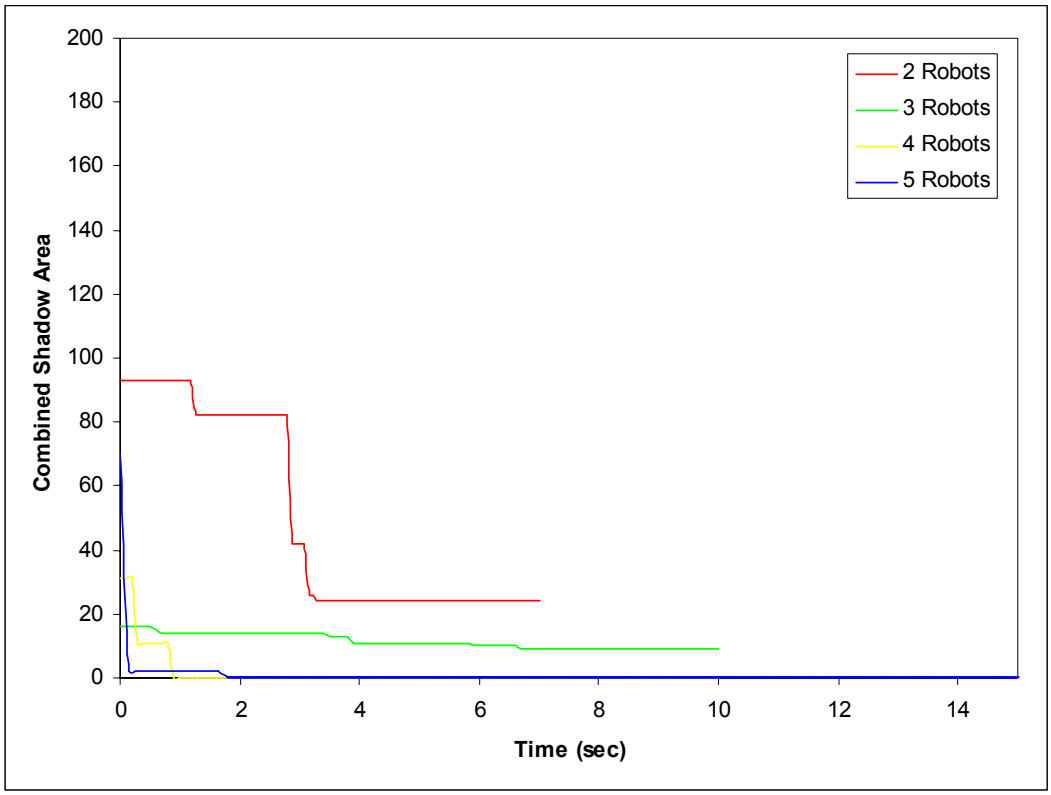

Figure 5.5: Map 2 convergence rates for 2-5 robots after 100 generations

The five-robot configuration converged in 15 seconds and the solution of four robots converged in 10 seconds. A recurring theme in this work is that the genetic algorithm will provide 'near-perfect' results. That is, the optimal solution will not always be obtained but something close to optimal will be obtained. The final solution will be dependent on how long the algorithm runs for and ultimately the number of generations that are iterated through to get a final solution.

Map Three

Map three is a simple map with square obstacles. Again, this map and the rest of the maps referenced were generated using the map-making program. The results are shown in figure 5.6.

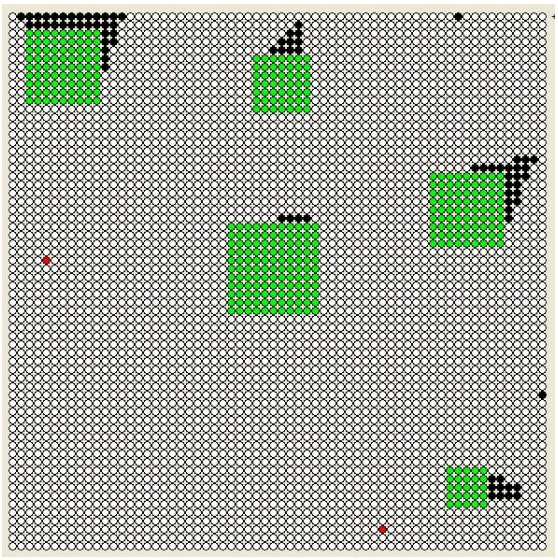

Figure 5.6a: Map 3, 2 Robots,
Shadow Area = 78

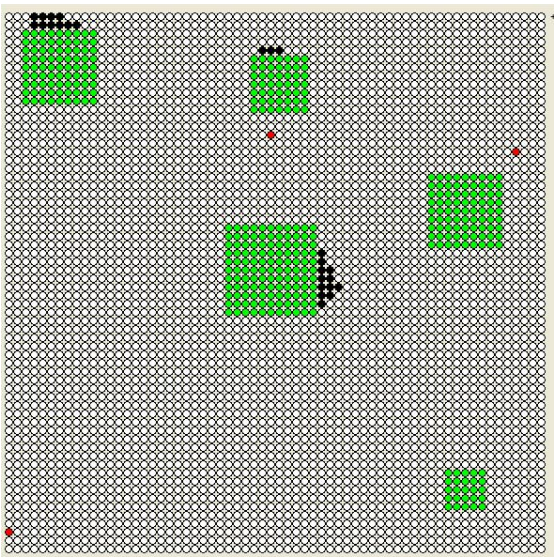

Figure 5.6b: Map 3, 3 Robots,
Shadow Area = 25

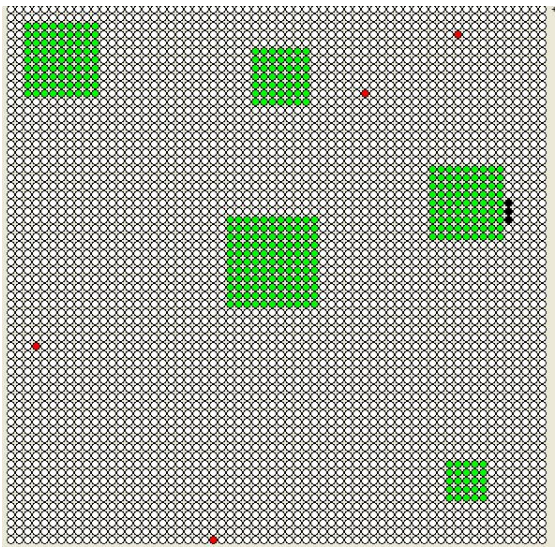

Figure 5.6c: Map 3, 4 Robots,
Shadow Area = 3

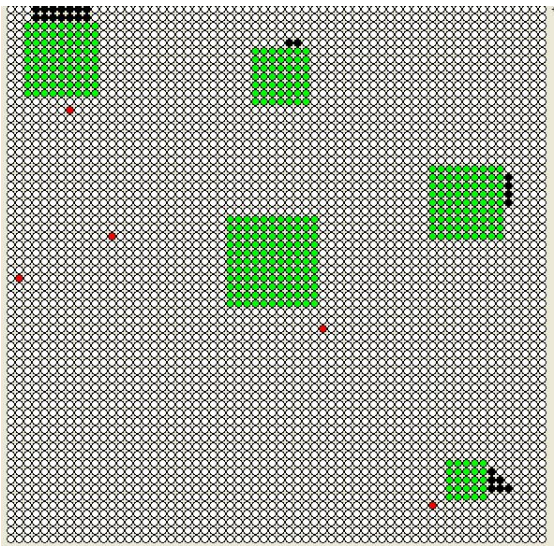

Figure 5.6d: Map 3, 5 Robots,
Shadow Area = 26

Figure 5.6: Results for Map 3

The results obtained from this map are interesting because the best solution is found to be the four-robot configuration. With a total shadow area of three grid points this solution outperforms the five-robot configuration. By visual inspection, the four-robot configuration appears to cover more of the map area, which is confirmed by the algorithm.

Plots of the shadow areas versus time are shown in figure 5.7. The four-robot configuration took a total of 11 second to converge and the five-robot configuration took a total of 15 second to converge. Once again, the algorithm terminates after it has cycled through 100 generations in the genetic algorithm.

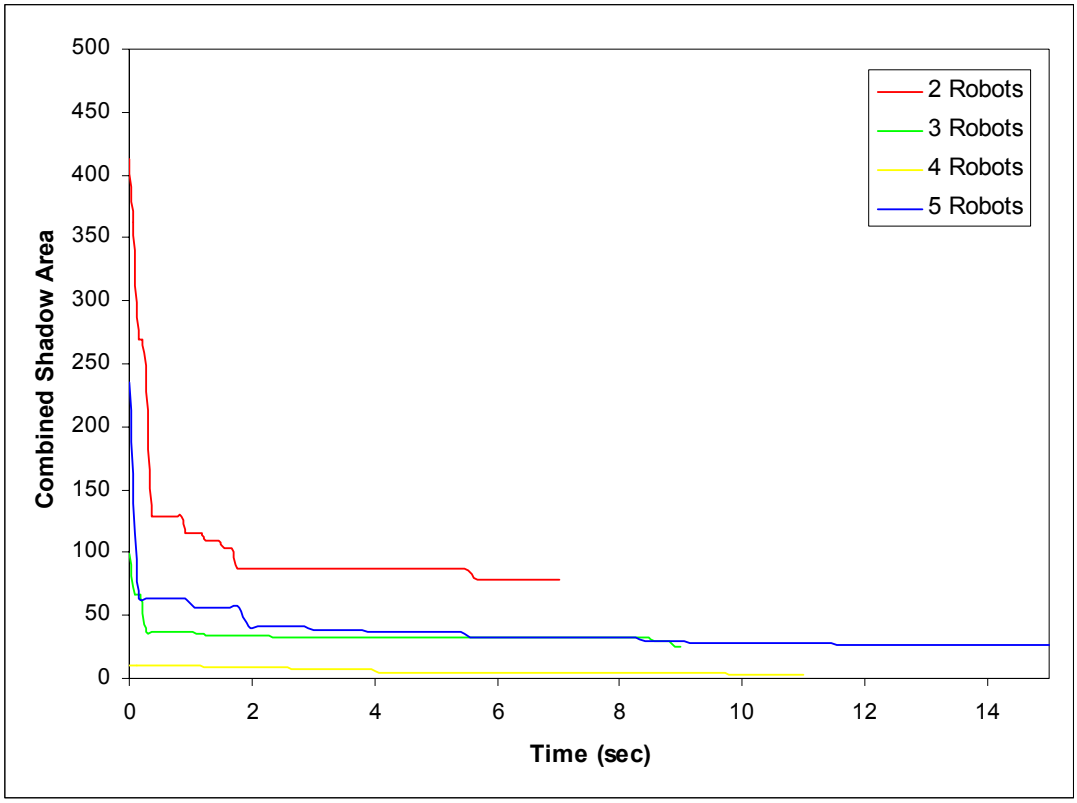

Figure 5.7: Map 3 convergence rates for 2-5 robots after 100 generations

Map Four

This map also contains square obstacles; the main difference being that they are all the same size and the arrangement is not randomized as in the previous example. This search space is slightly more complex due to the number of obstacles present and the circular arrangement throughout the space. The results are shown in figure 5.8.

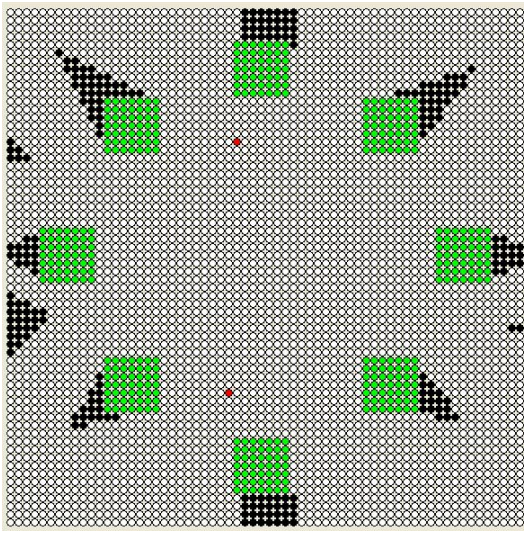

Figure 5.8a: Map 4, 2 Robots,
Shadow Area = 220

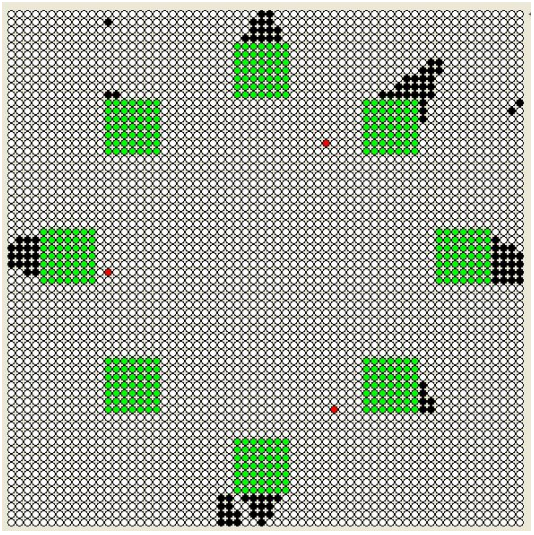

Figure 5.8b: Map 4, 3 Robots,
Shadow Area = 108

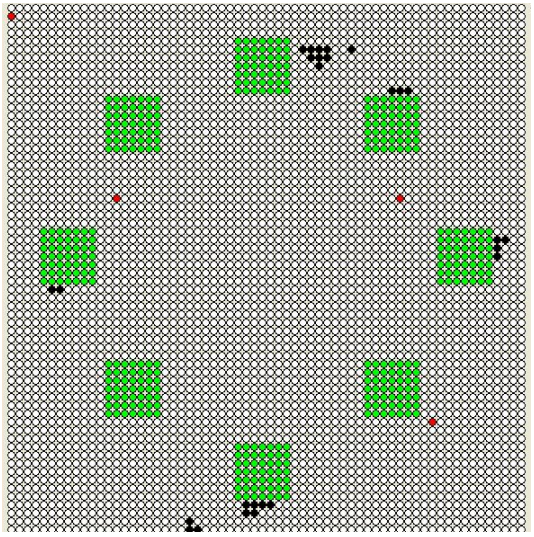

Figure 5.8c: Map 4, 4 Robots,
Shadow Area = 27

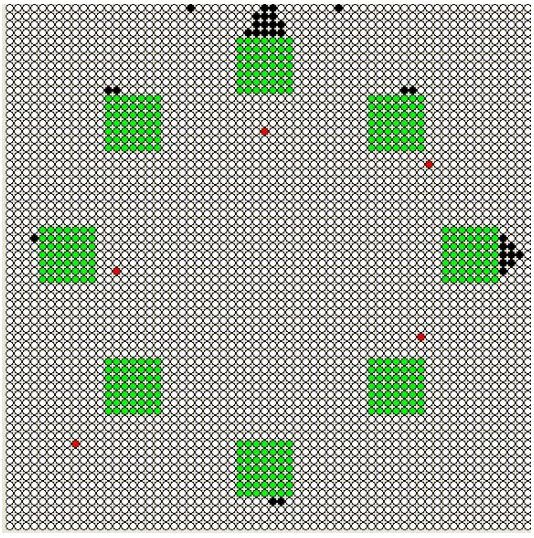

Figure 5.8d: Map 4, 5 Robots,
Shadow Area = 32

Figure 5.8: Results for Map 4

By visual inspection of the two-robot case, this arrangement of obstacles within the search space limits the robot positions such that they still fulfill the line-of-sight criteria. The algorithm successfully positions the robots for all four scenarios. It is interesting to note that the four-robot configuration has a resulting shadow area of 27 and the addition of another robot does not improve this result. Therefore, if the objective were to minimize the shadow area and minimize the number of robots used, then the solution to this problem would be the four-robot configuration. The convergence rates for each scenario are shown in figure 5.9.

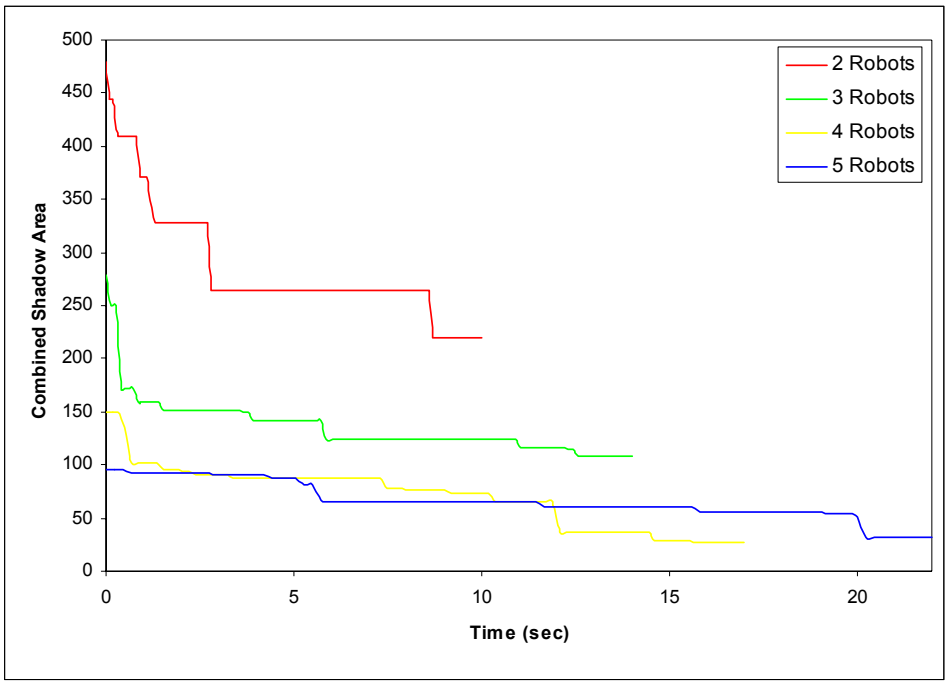

Figure 5.9: Map 4 convergence rates for 2-5 robots after 100 generations

Map Five

This map contains different sized square obstacles that are randomly placed throughout the search space. For this particular space the five-robot configuration has the lowest shadow area. This is an intuitive result and is primarily due to the placement of a

robot in the right-hand corner of the map. Doing so eliminates any unseen area along the right-hand side of the map (see figure 5.10).

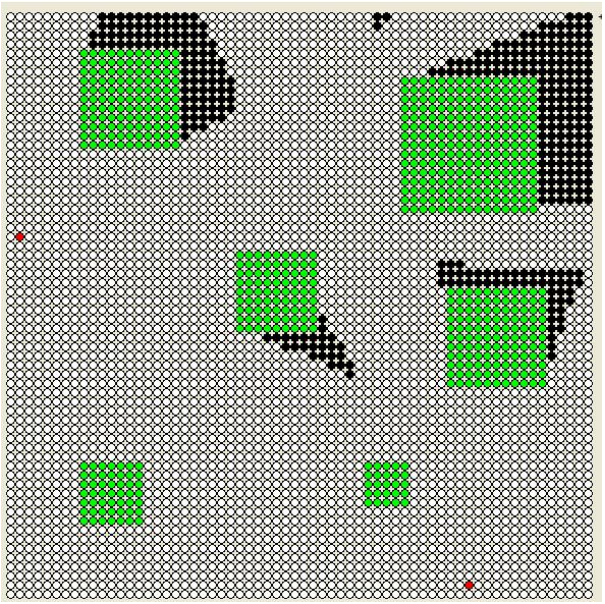

Figure 5.10a: Map 5, 2 Robots, Shadow Area = 334

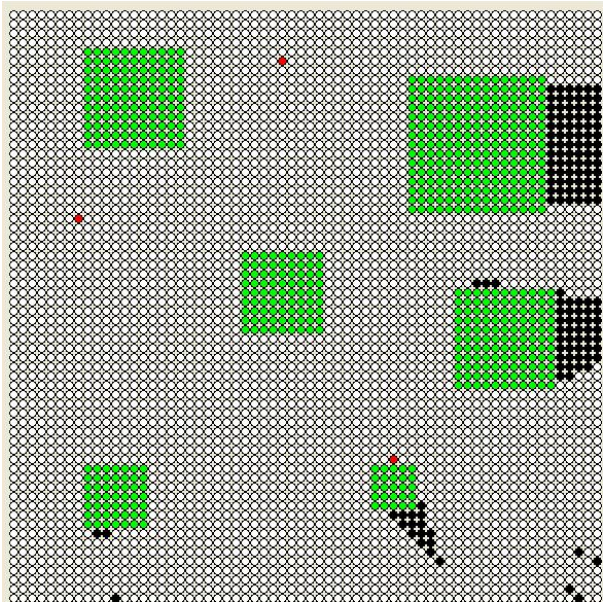

Figure 5.10b: Map 5, 3 Robots, Shadow Area = 145

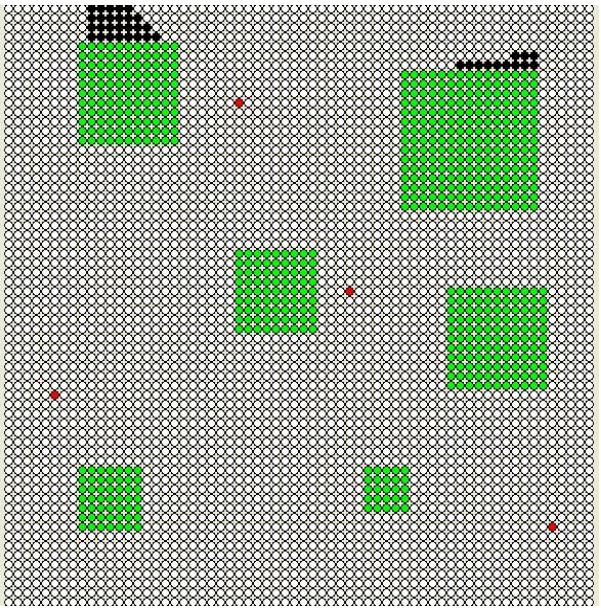

Figure 5.10c: Map 5, 4 Robots, Shadow Area = 38

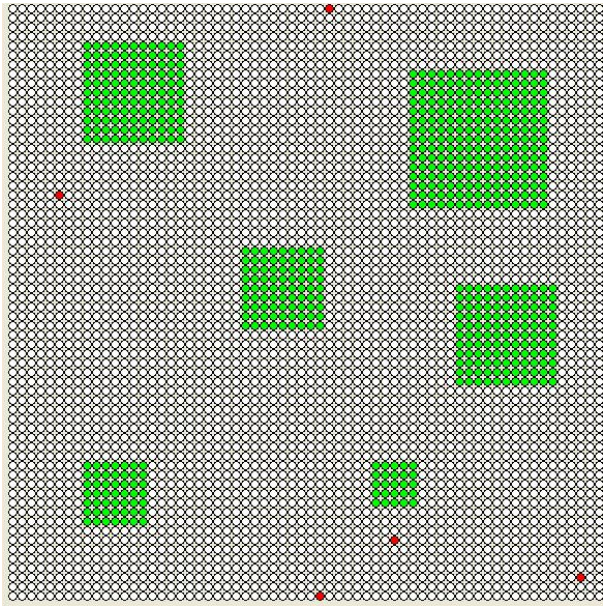

Figure 5.10d: Map 5, 5 Robots, Shadow Area = 0

Figure 5.10: Results for Map 5

The five-robot configuration provides the optimal solution for this search space. It is intuitive to think that addition of more robots to the solution should maximize the coverage area and for this situation this does occur. The addition of more robots results in a gradual decrease in the shadow areas and eventually a shadow area of zero is achieved. The convergence rates are shown in figure 5.11.

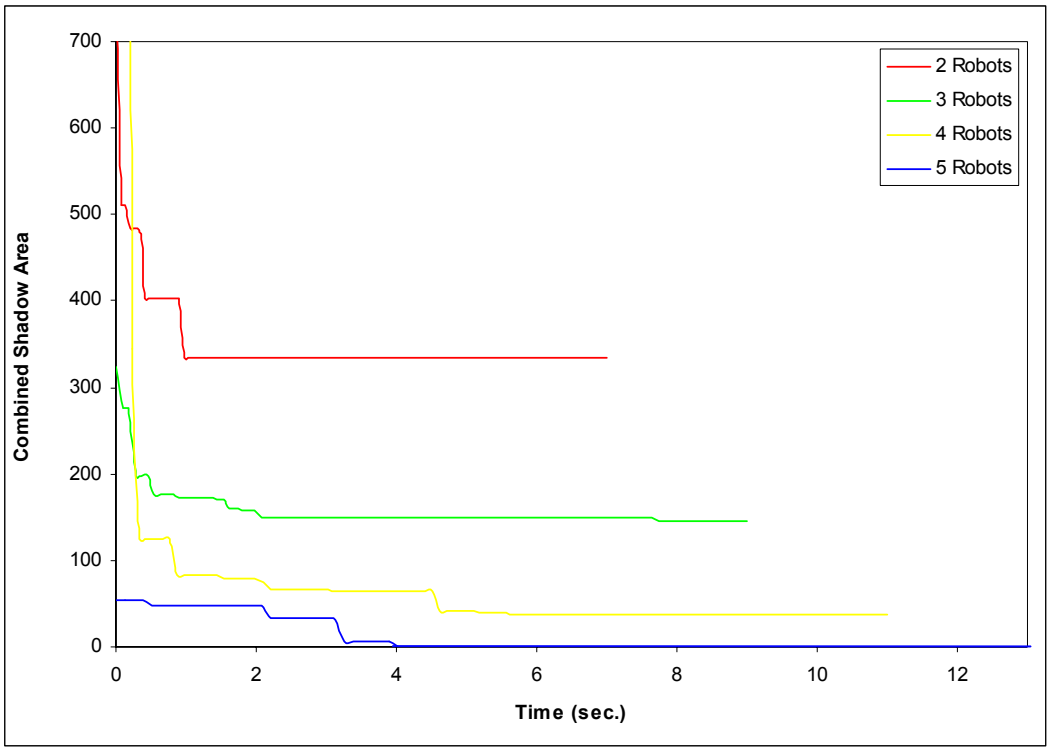

Figure 5.11: Map 5 convergence rates for 2-5 robots after 100 generations

Map Six

This is one of the more complex maps that were processed. There are quite a few obstacles of varying sizes and there is an obstacle created from two square obstacles. The algorithm does not appear to have any problems with this shape. This map was designed to push the limits of the algorithm, specifically the line-of-sight criteria. With this many

obstacles present the algorithm still positions the robots such that the line-of-sight criteria is fulfilled. The results are presented in figure 5.12.

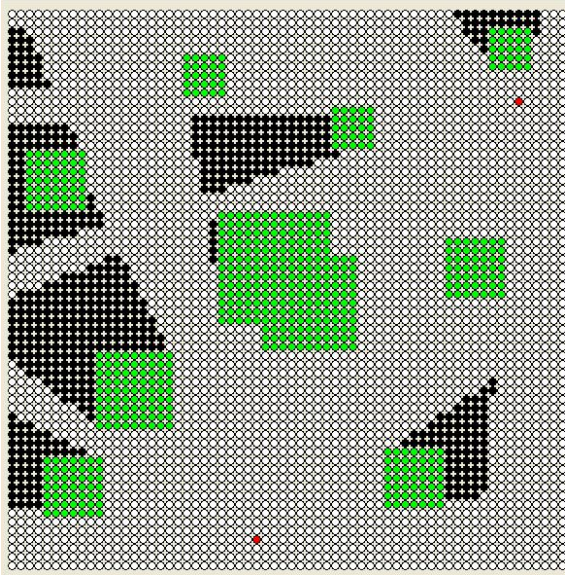

Figure 5.12a: Map 6, 2 Robots,
Shadow Area = 549

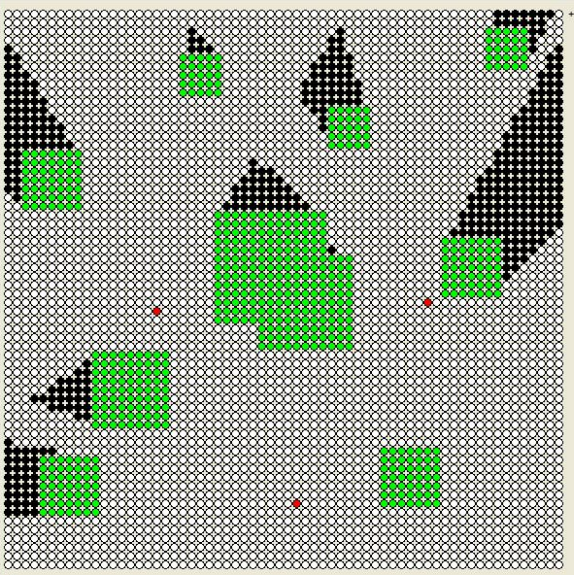

Figure 5.12b: Map 6, 3 Robots,
Shadow Area = 396

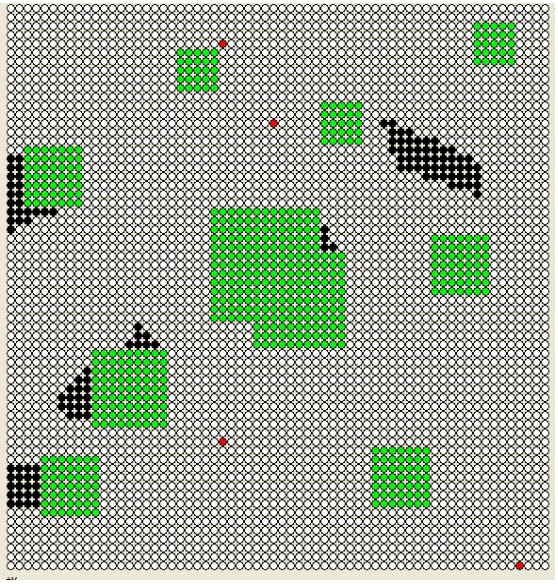

Figure 5.12c: Map 6, 4 Robots,
Shadow Area = 120

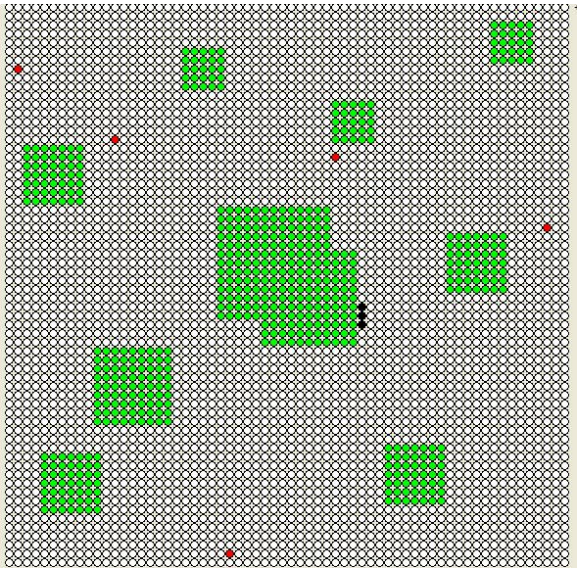

Figure 5.12d: Map 6, 5 Robots,
Shadow Area = 3

Figure 5.12: Results for Map 6

The rates of convergence are shown in figure 5.13. Although this is a somewhat complicated map the time in which a solution is calculated does not suffer. For the five-robot case a solution is found in 18 seconds.

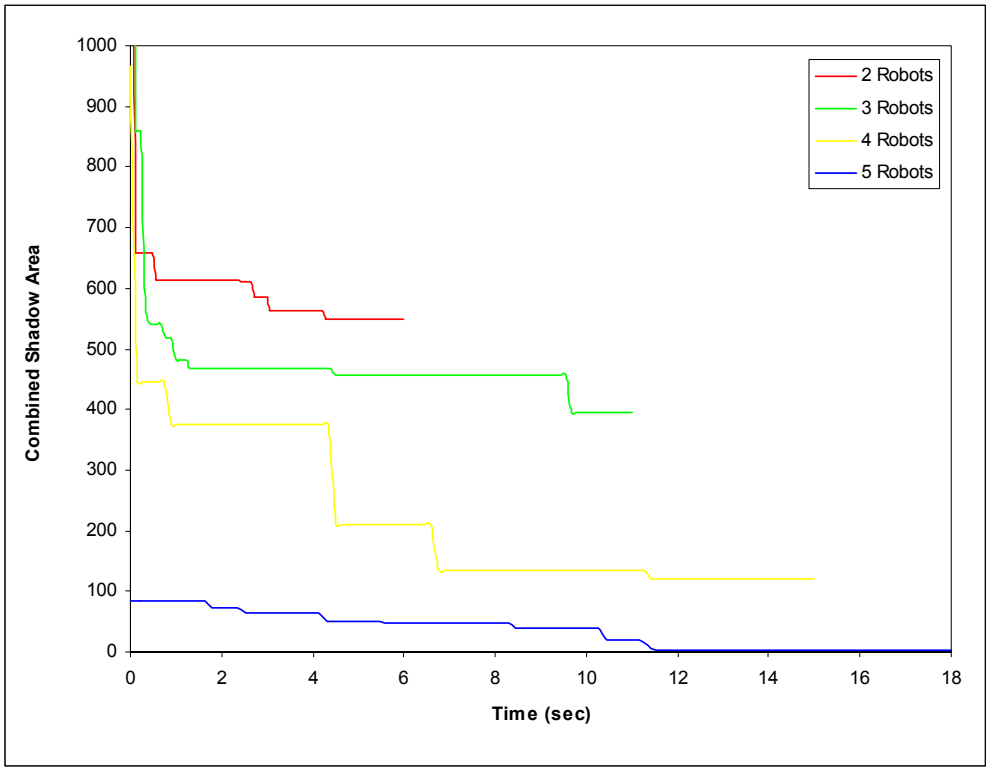

Figure 5.13: Map 6 convergence rates for 2-5 robots after 100 generations

256×256 Resolution Maps

The higher resolution maps were created and tested in order to demonstrate that the algorithm is applicable to larger search spaces. This demonstrates that this algorithm could be used on an actual aerial image of an area where corresponding robotic vehicles would be positioned.

For this particular application the algorithms were run on a Linux platform. This was done in the interest of time. Due to the increased search space and the number of maps that are analyzed it was easier and faster to create a script file that would cycle through

each map and save the corresponding results to a file. A GUI was not created, however the corresponding results and data were saved in a ppm (portable pixmap) file format that could be opened and viewed at a later time. This file format is relatively easy to encode, as it is the most simplistic format for color images (see figure 5.14).

```
P6
256 256
255
0 255 63 9 85 2A……
```

Figure 5.14: Sample ppm file and header information (first 3 lines)

The first line indicates the ppm type, which can either be a binary or ASCII format. The second line contains the height and width of the image. For these images the height and width are the same. The third line contains the maximum RGB values contained within the image. The rest of the file contains all of the RGB values for each pixel in the image. The results for the higher resolution maps will be presented in the next six sections

Map One

The first map or search space that was tested consisted of a single obstacle placed in the center of the space. This is an extremely simplistic search space and yields interesting results as more robots are added to the solution. It would be difficult, if not impossible, to eliminate the shadow area for the two-robot configuration while simultaneously fulfilling the line-of-sight criteria.

The addition of a third robot provides an optimal solution for this map. Addition of a fourth and fifth robot also result in optimal solutions, each having shadow areas of zero. The graphical results are provided in figure 5.15.

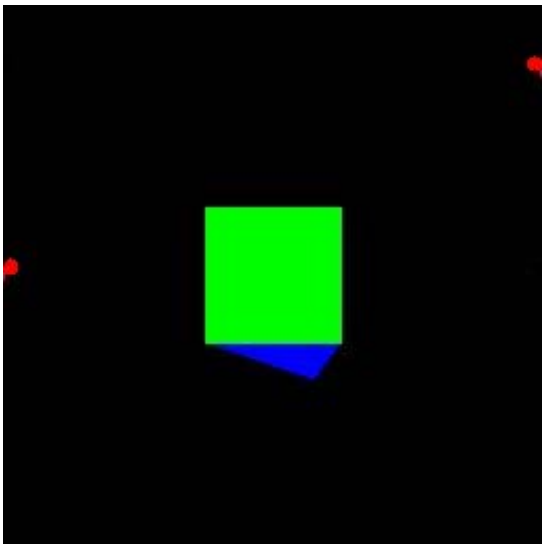

Figure 5.15a: Map 1, 2 Robots,
Shadow Area = 533

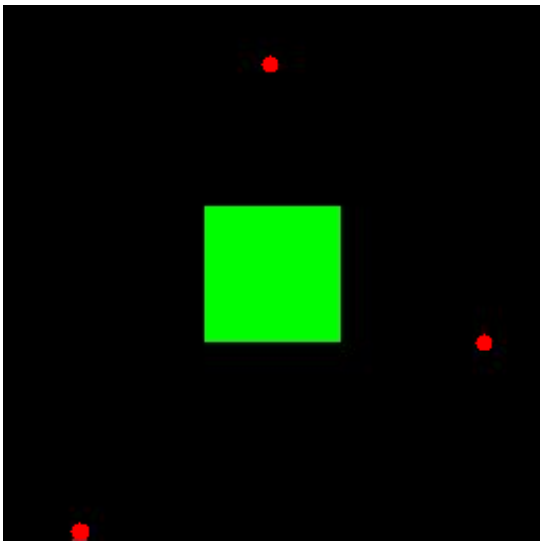

Figure 5.15b: Map 1, 3 Robots,
Shadow Area = 0

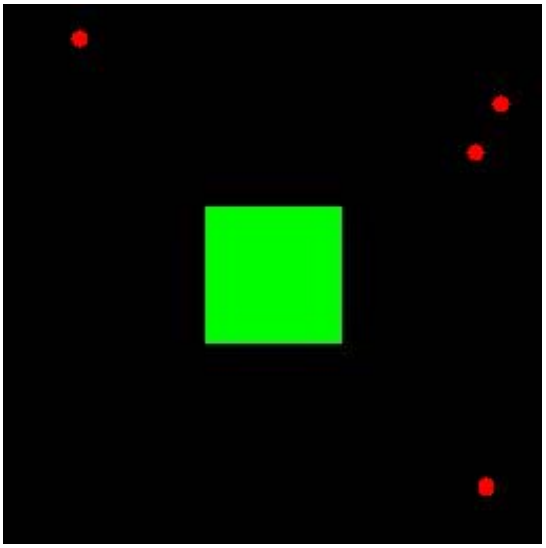

Figure 5.15c: Map 1, 4 Robots,
Shadow Area = 0

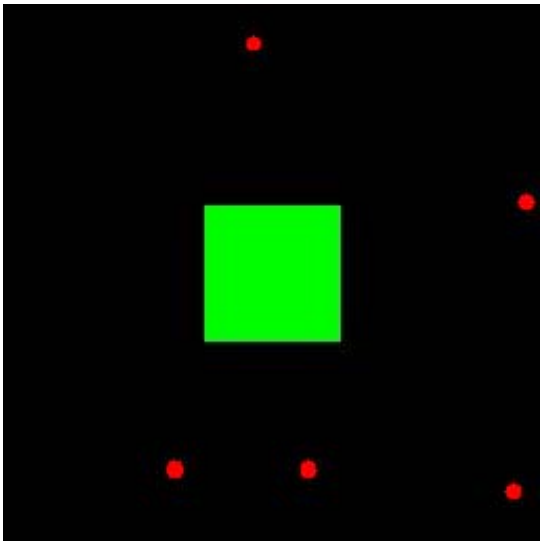

Figure 5.15d: Map 1, 5 Robots,
Shadow Area = 0

Figure 5.15: Results for Map 1

These results are intuitive and provide insight into the genetic algorithm technique that is being used to search through the space. Plots of the rates of convergence for each configuration are shown in figure 5.16. The only data displayed on the plot is that for the two-robot configuration. The three-, four-, and five-robot configurations converge to the

optimal solution (shadow area = 0) in less than one second; therefore the corresponding data is zero. The total runtime for the two-robot case has increased to approximately two minutes and this is due to the increase in the search space.

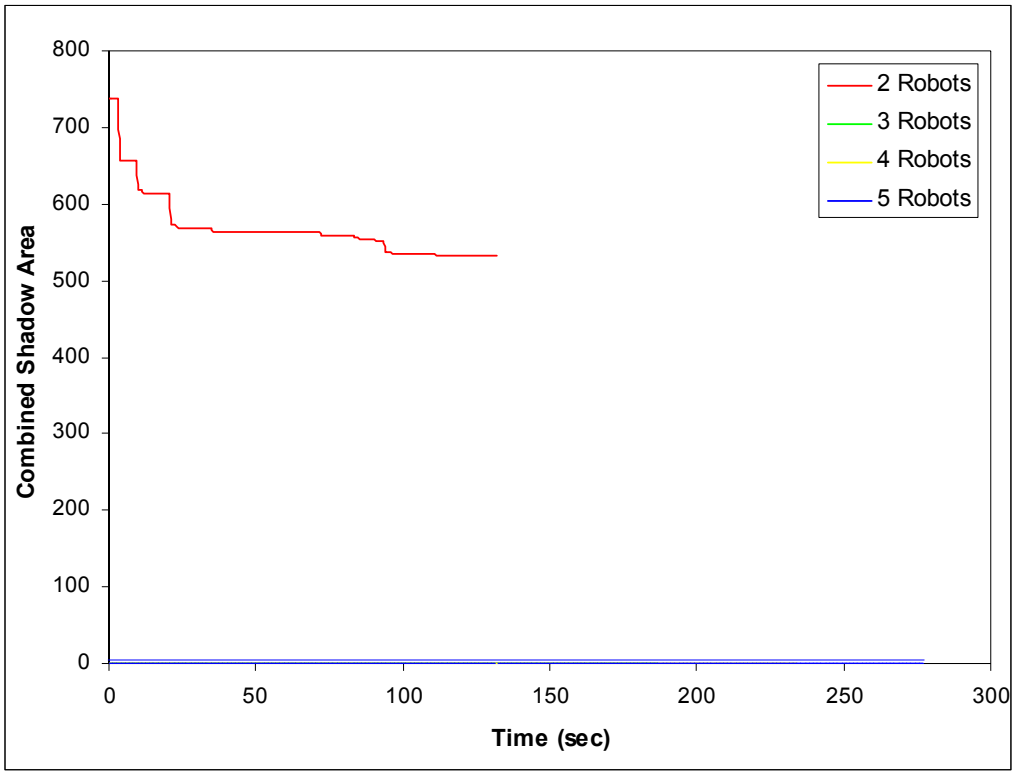

Figure 5.16: Map 1 convergence rates for 2-5 robots after 200 generations

Map Two

This map contains three obstacles that are located in the top half of the search space. Previous results have indicated that the number of obstacles and the obstacle locations affect how well the algorithm performs especially in the two- and three-robot cases. In these cases less area can be covered resulting in greater shadow areas and fewer options regarding final robot positions.

The results presented in figure 5.17 show that the addition of a third robot is detrimental to the resulting shadow area. Addition of a fourth and fifth robot drastically improves the final solution and an optimal solution is obtained for the five-robot case.

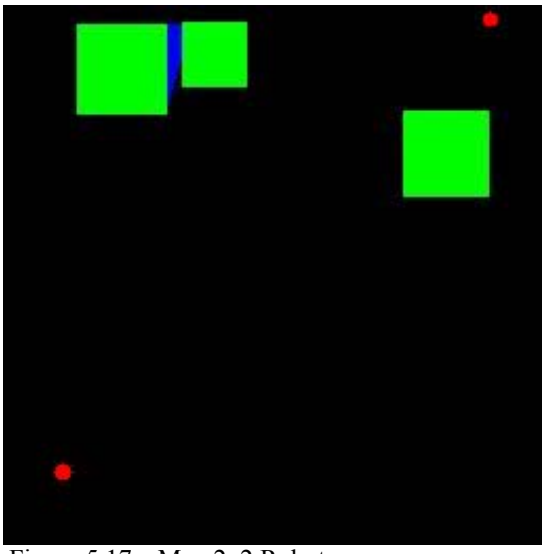

Figure 5.17a: Map 2, 2 Robots, Shadow Area = 198

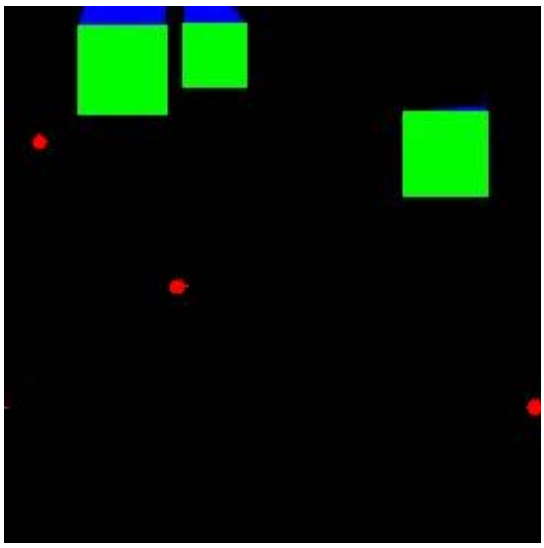

Figure 5.17b: Map 2, 3 Robots, Shadow Area = 611

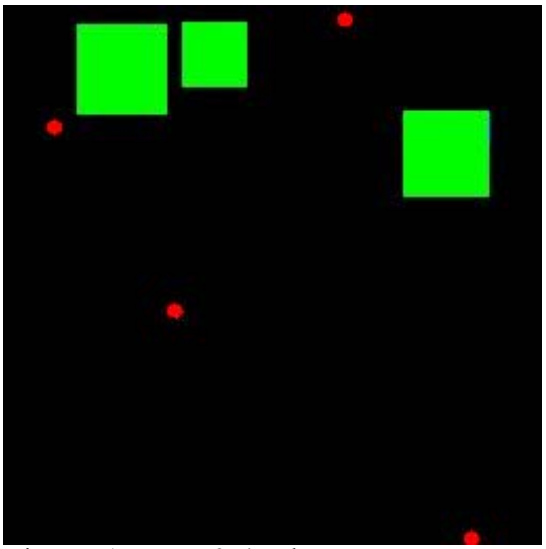

Figure 5.17c: Map 2, 4 Robots, Shadow Area = 16

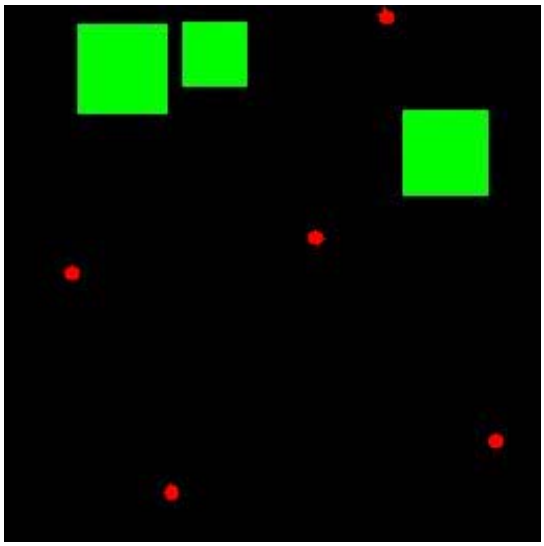

Figure 5.17d: Map 2, 5 Robots, Shadow Area = 0

Figure 5.17: Results for Map 2

The shadow area for the four-robot configuration is 16 pixels, but this is barely visible on an image of this scale. As stated earlier, the increase in the size of the search space

affects the time in which a solution is obtained. Figure 5.18 shows the corresponding rates of convergence for the four situations. The three-robot takes approximately six minutes to arrive at a solution and while the five-robot configuration takes approximately four minutes to converge to the optimal solution of zero shadow area.

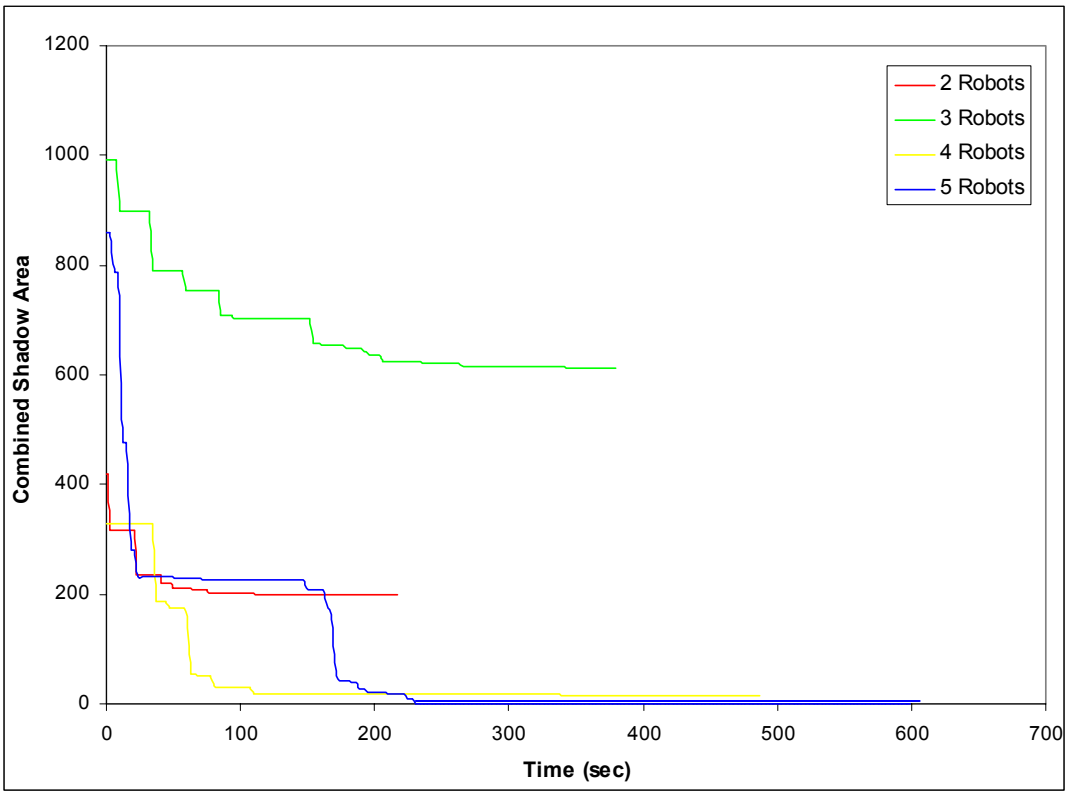

Figure 5.18: Map 2 rates of convergence for 2-5 robots after 200 generations

Map Three

This map incorporates four obstacles into the search space. The results shown in figure 5.19 indicate that the addition of a fourth robot increases the shadow area considerably. This may be attributed to the random nature of the genetic algorithm. Addition of a fifth robot to the solution further improves the final results. Only a minimal amount of area is hidden from the five robots' views. This algorithm is

interesting because it tries to maximize the area covered and maintain line-of-sight among the robots.

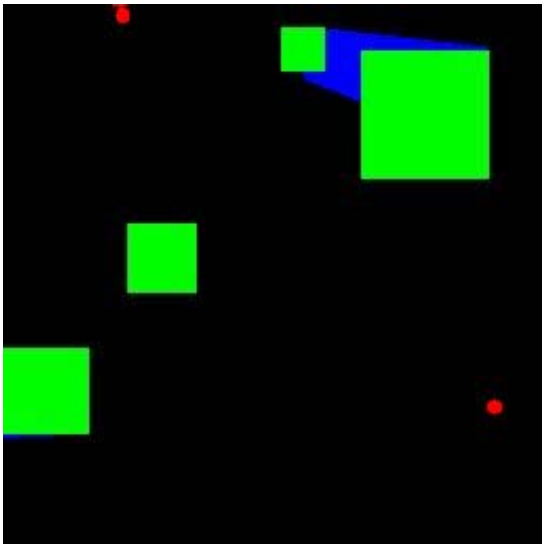

Figure 5.19a: Map 3, 2 Robots, Shadow Area = 944

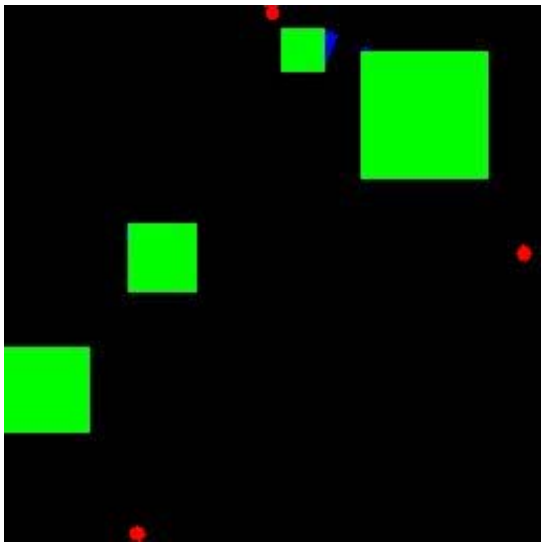

Figure 5.19b: Map 3, 3 Robots, Shadow Area = 70

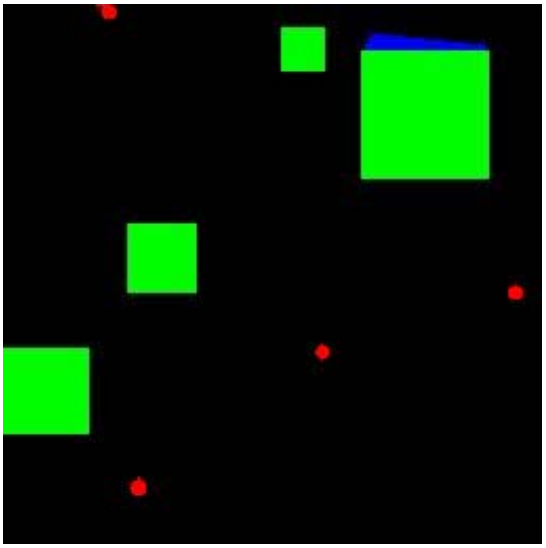

Figure 5.19c: Map 3, 4 Robots, Shadow Area = 306

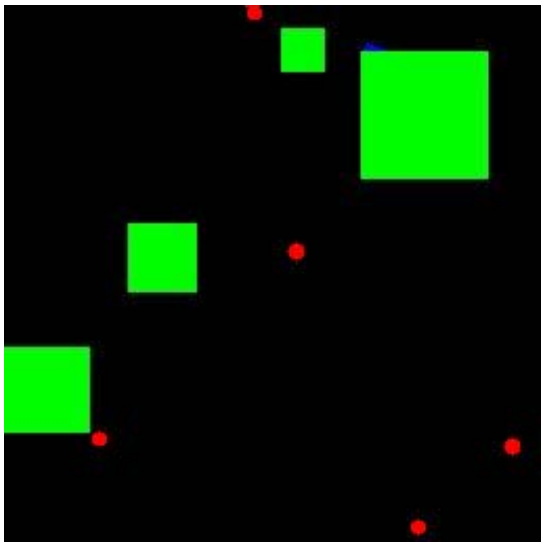

Figure 5.19d: Map 3, 5 Robots, Shadow Area = 25

Figure 5.19: Results for Map 3

The processing time for the two-robot configuration is approximately four and a half minutes and steadily increases as more robots are added to the solution. The processing

time for the five-robot configuration is approximately 11 minutes. A plot of the rates of convergence for this map is shown in figure 5.20.

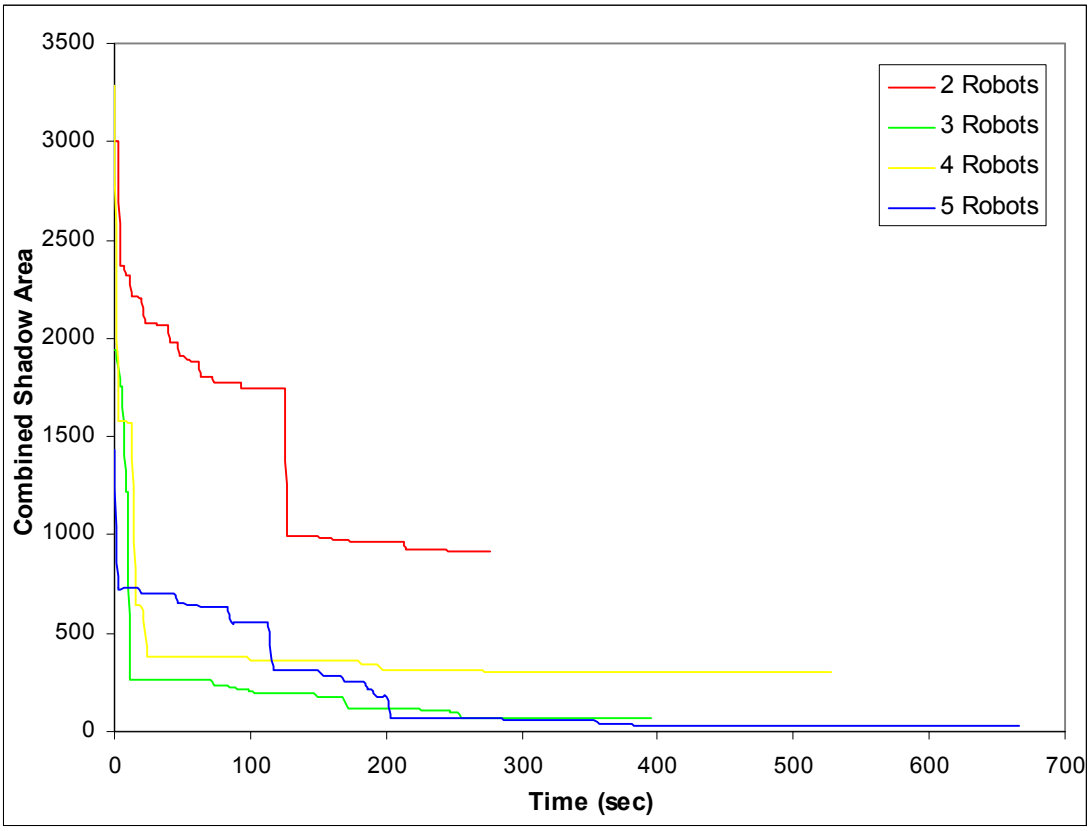

Figure 5.20: Map 3 rates of convergence for 2-5 robots after 200 generations

Map Four

Four larger obstacles are incorporated into this search space. Although this map is somewhat cluttered and a higher-resolution image the algorithm performs as expected. Once again, the obstacles have been randomly placed throughout the space using a custom program written in the C programming language.

The two-robot case resulted in the greatest shadow area followed sequentially by the other three cases. Both the four- and five-robot configurations result in the optimal solution of a shadow area equal to zero. For the four- and five-robot cases the final robot

positions are optimal for maximizing map coverage while maintaining line-of-sight communications. The results are shown in figure 5.21.

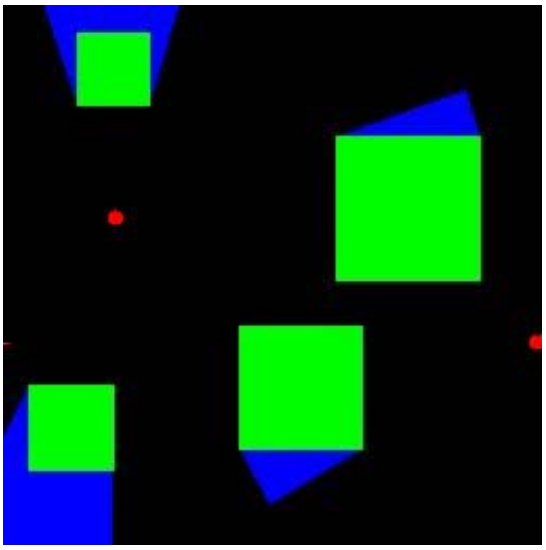

Figure 5.21a: Map 4, 2 Robots,
Shadow Area = 4749

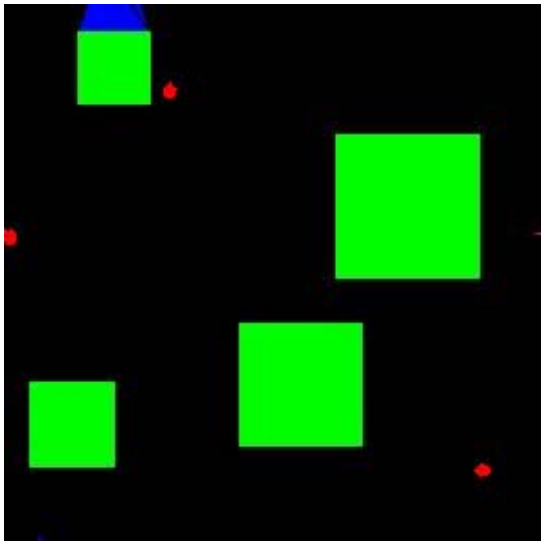

Figure 5.21a: Map 4, 3 Robots,
Shadow Area = 364

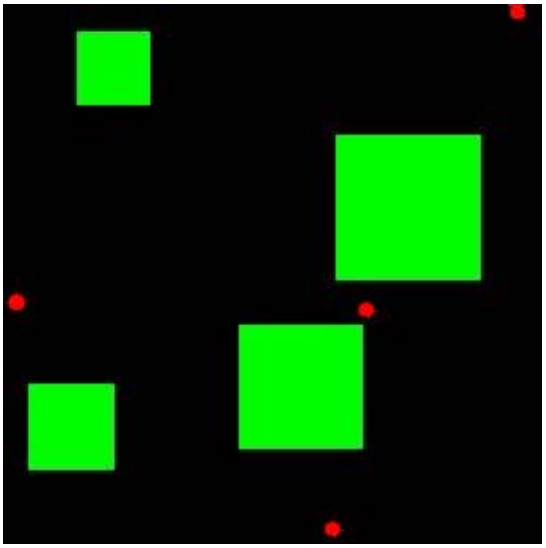

Figure 5.21c: Map 4, 4 Robots,
Shadow Area = 0

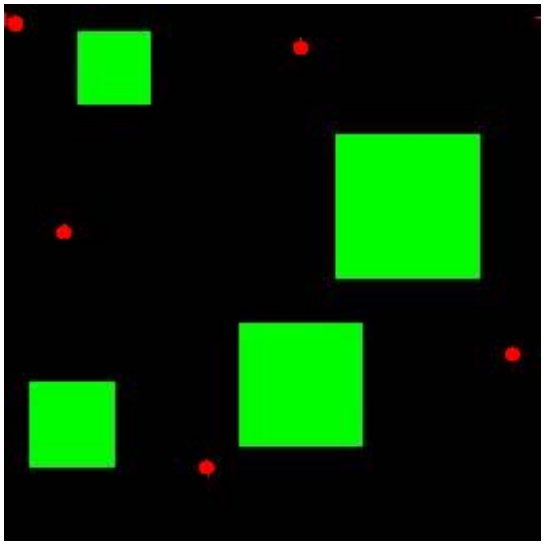

Figure 5.21d: Map 4, 5 Robots,
Shadow Area = 0

Figure 5.21: Results for Map 4

From the plots shown in figure 5.22 it is interesting to note that the four-robot case converged to a zero solution in approximately four minutes, while the five-robot case converged to a zero solution in approximately two minutes. Addition of a fifth robot decreased the time in which the algorithm converged to zero. This is an intuitive result because addition of another robot serves to maximize the coverage area therefore expediting the search for the optimal solution.

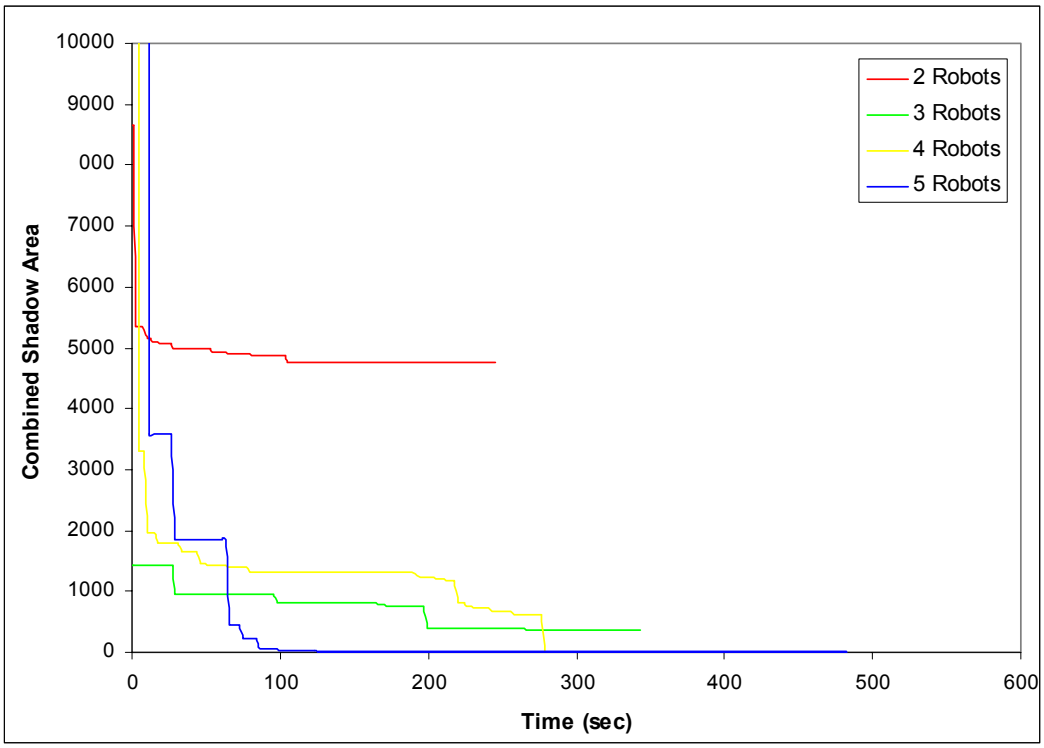

Figure 5.22: Map 4 rates of convergence for 2-5 robots after 200 generations

Map Five

This map is composed of six obstacles; four of which are clustered together and the remaining two are spread farther apart. The results gradually improve as another robot is added to the solution. Although the five-robot case does not obtain a shadow area of zero, a drastic improvement is made when compared with the two-robot case.

Again the two-robot case resulted in the greatest shadow area followed sequentially by the three other cases. Increasing the number of generations and the size of the population would greatly affect the final results for all four cases. In addition this would increase the overall processing time. The results are shown in figure 5.23.

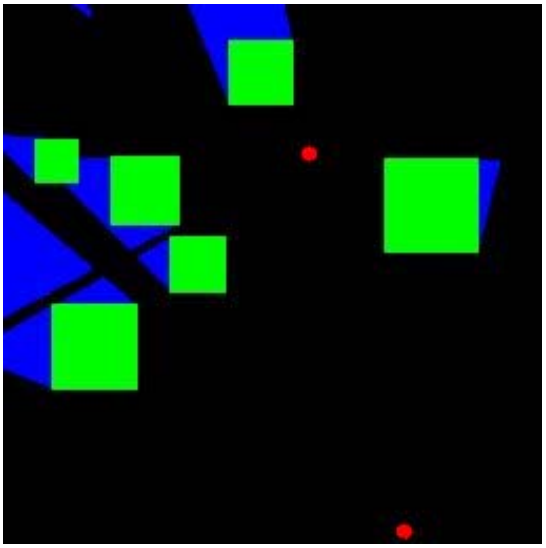

Figure 5.23a: Map 5, 2 Robots,
Shadow Area = 6576

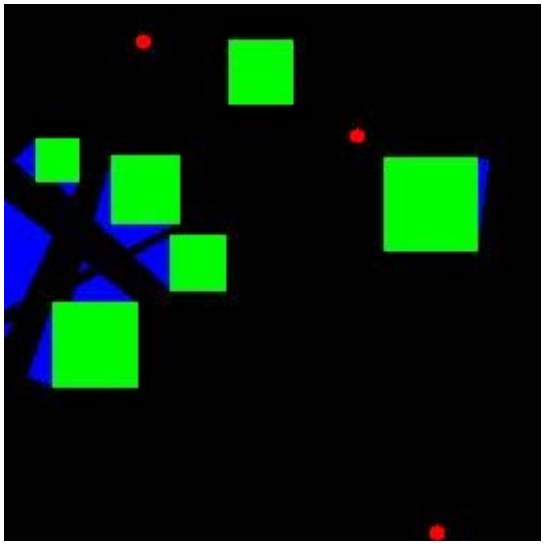

Figure 5.23b: Map 5, 3 Robots,
Shadow Area = 2022

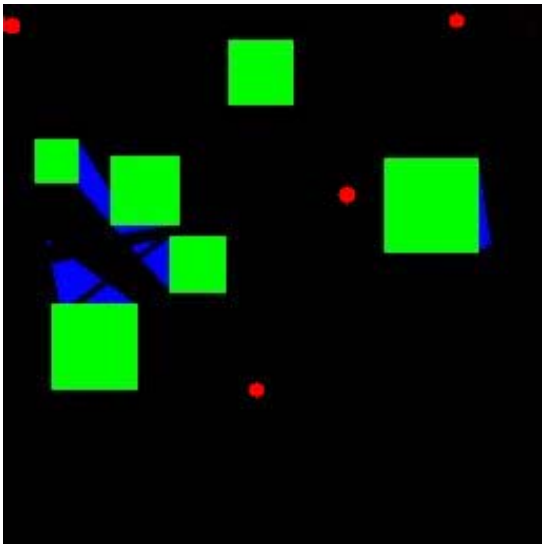

Figure 5.23c: Map 5, 4 Robots,
Shadow Area = 1072

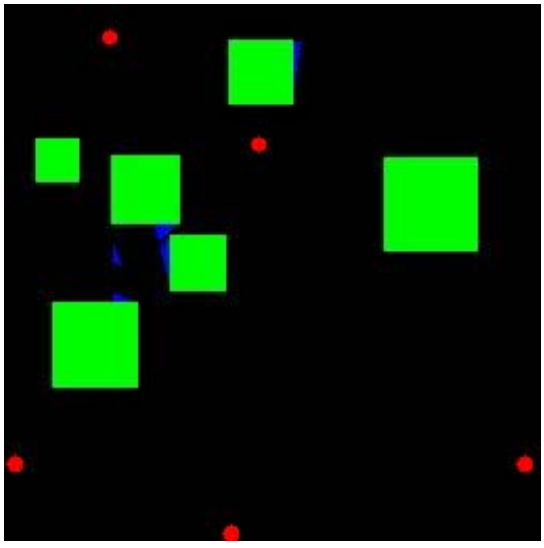

Figure 5.23d: Map 5, 5 Robots,
Shadow Area = 184

Figure 5.23: Results for Map 5

This is one of the more complicated search spaces therefore there is an increase in the total processing time. The two-robot case takes approximately six minutes to obtain a solution and this time gradually increases as more robots are considered in the solution. Plots of the convergence rates are shown in figure 5.24.

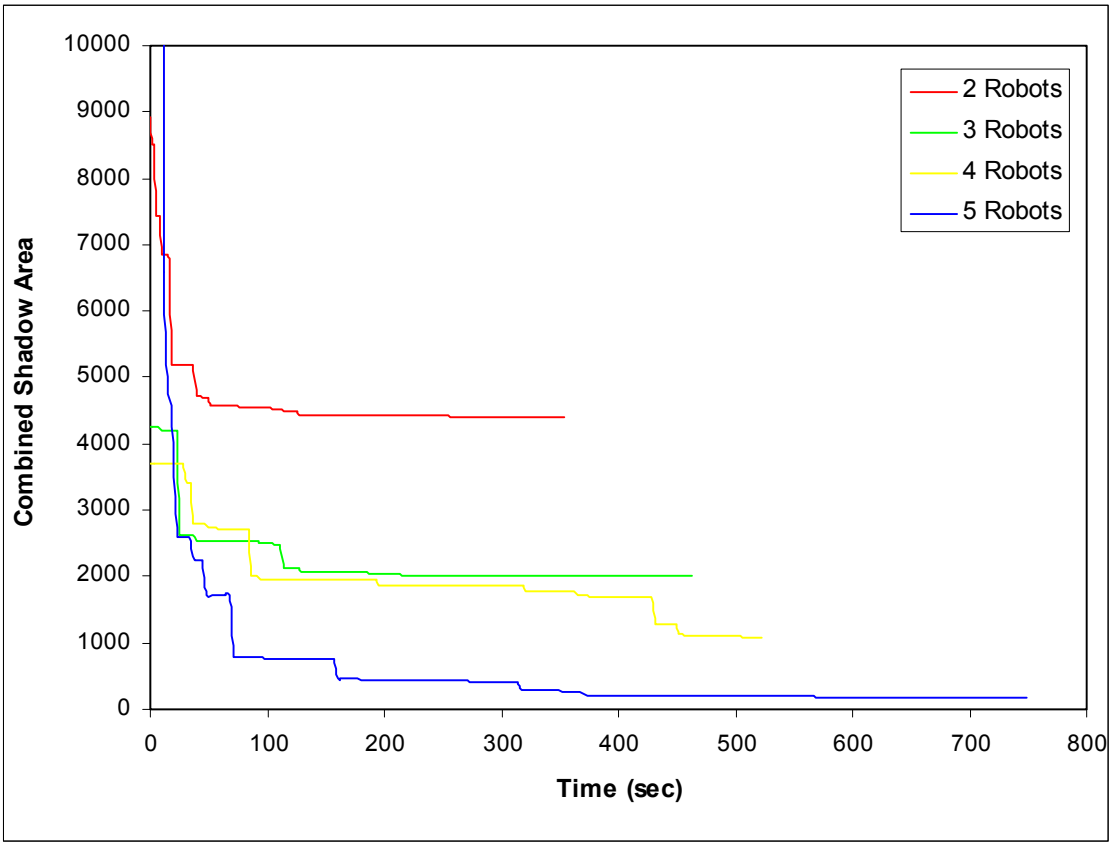

Figure 5.24: Map 5 rates of convergence for 2-5 robots after 200 generations

Map Six

This final map consisted of six obstacles randomly placed throughout the search space. The obstacles in this map are spread throughout the space and again, two to five robots will be positioned throughout the map. Two robots result in a considerable amount of hidden area. Addition of a third robot provides desirable results as the hidden regions decrease dramatically. The shadow area continues to decrease with the addition

of a fourth robot, however, addition of the fifth robot results in an increase in the shadow area. Although this is a complex map, the algorithm succeeds at finding reasonable solutions for each situation. The results are displayed in figure 5.25.

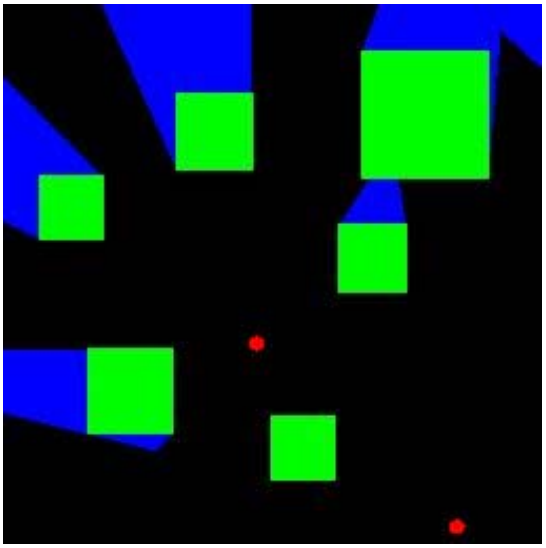

Figure 5.25a: Map 6, 2 Robots, Shadow Area = 8348

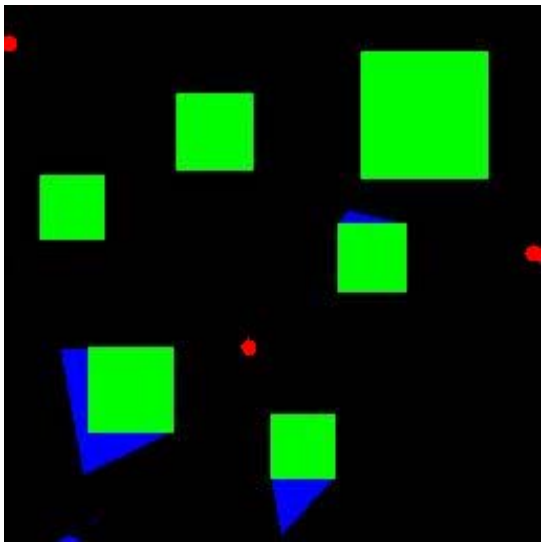

Figure 5.25b: Map 6, 3 Robots, Shadow Area = 1320

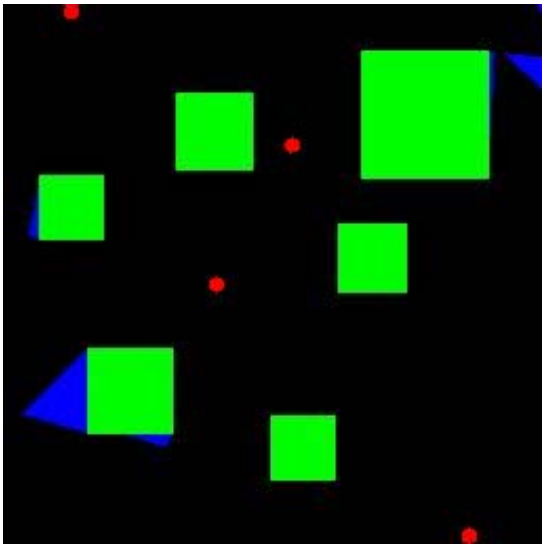

Figure 5.25c: Map 6, 4 Robots, Shadow Area = 962

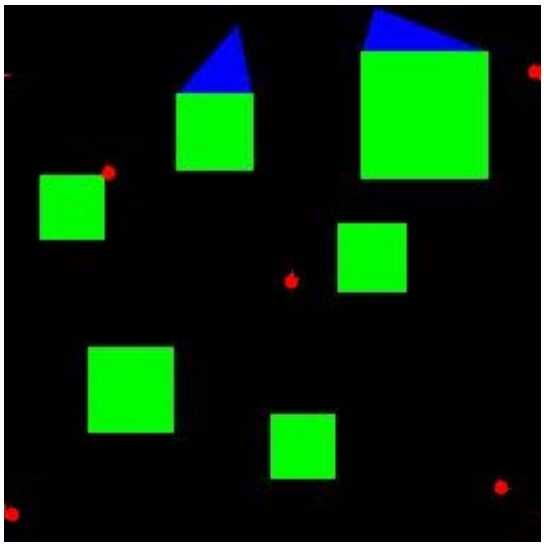

Figure 5.25d: Map 6, 5 Robots, Shadow Area = 1197

Figure 5.25: Results for Map 6

This algorithm not only serves to determine the optimal robot placement but also inadvertently provides some insight as to the number of robots necessary to cover the search space most efficiently. The plots of the rates of convergence are shown in figure 5.26 and note that the four-robot case provides smallest shadow area in approximately 580 seconds or 9.6 minutes.

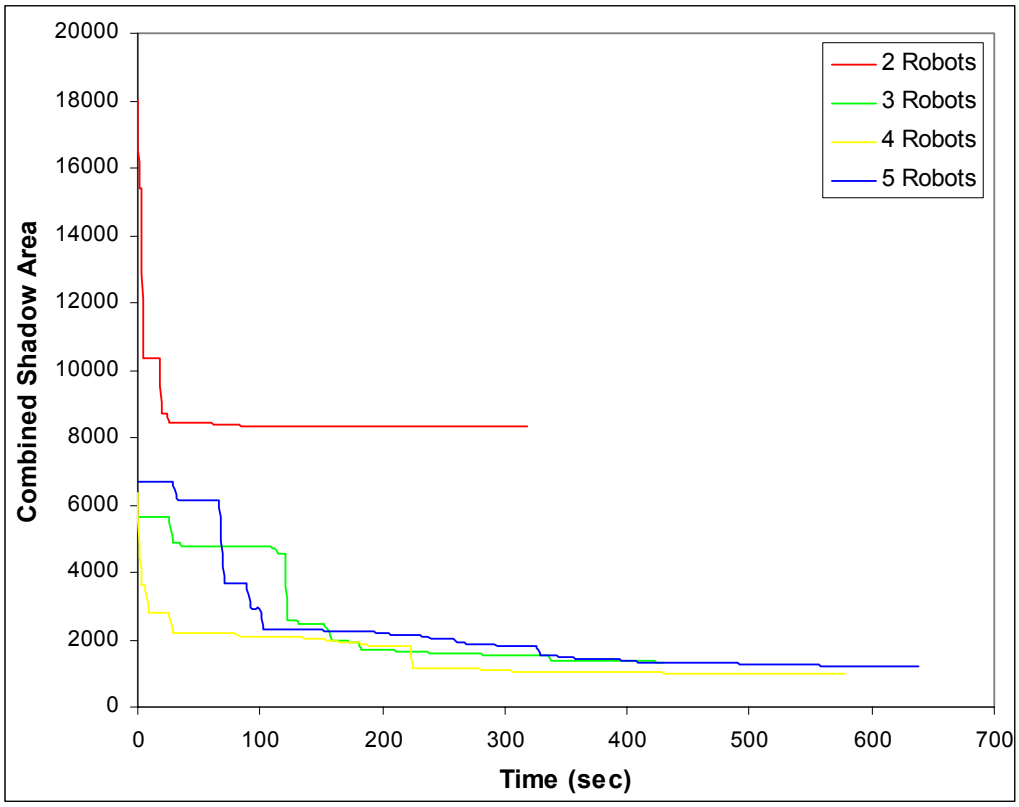

Figure 5.26: Map 6 rates of convergence for 2-5 robots after 200 generations

All of the examples shown in the previous sections provide a reasonable estimate of the visible as well as hidden area from the robots' perspectives. These results are affected by the number of individuals used in the population and the number of generations incorporated into the genetic algorithm. Better results may be obtained by allowing the algorithm to run for a longer period of time. The final results proved to accomplish the overall objective of this work. The next section will compare the results

from an exhaustive search technique to the genetic algorithm technique described throughout this chapter.

Exhaustive Search Technique versus Genetic Algorithm Technique

The focus of this section is to present the results from two different techniques and observe the differences in the final solutions and the overall processing times. The exhaustive search technique calculates the shadow area for every possible two-robot configuration and keeps track of the current "best" solution. For the two-robot case, this method is computationally intensive and takes hours to arrive at a final solution. Therefore, the results compared will only be for the two-robot case.

Three 64×64 resolution maps will be compared and the results will be displayed graphically as in previous sections in addition to similar plots of the convergence rates for each technique. All of the maps have been presented in previous sections. The data in the convergence plots contain the current best solution for both techniques.

Comparison #1

The first map is the most simplistic of the three maps that will be compared. Both methods arrive at a similar solution for the shadow area with slightly different robot positions. The main difference that should be noted is the overall processing time required to arrive at the solution.

For this particular example, the exhaustive search technique takes approximately four hours to complete while the genetic algorithm technique finds a solution in seven seconds. The exhaustive search would have found multiple solutions that have a shadow area of 14, but the first one found was reported as the best solution. The results from the two methods are shown in figure 5.27. Data plotted are the best of each run and are not the entire data set.

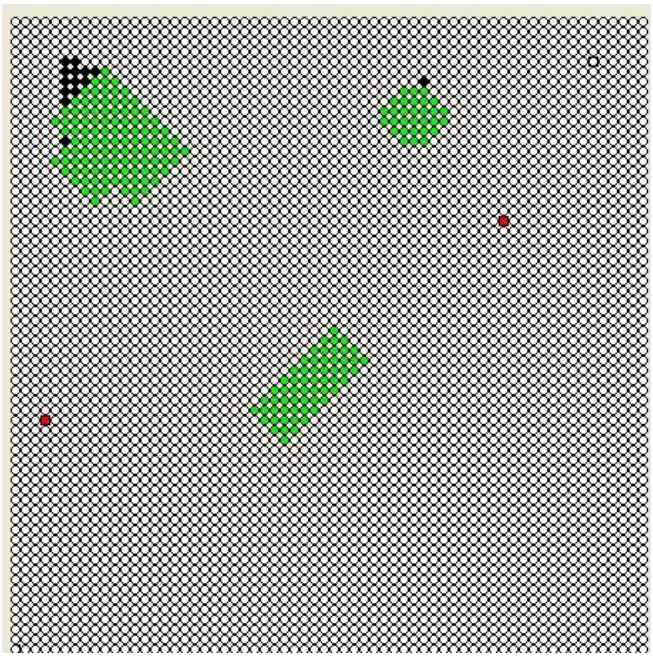

Figure 5.27a: Exhaustive Search Results
Shadow Area: 14
Robot 1 Position: (49, 20)
Robot 2 Position: (3, 40)
Total Time: 4.27 hours

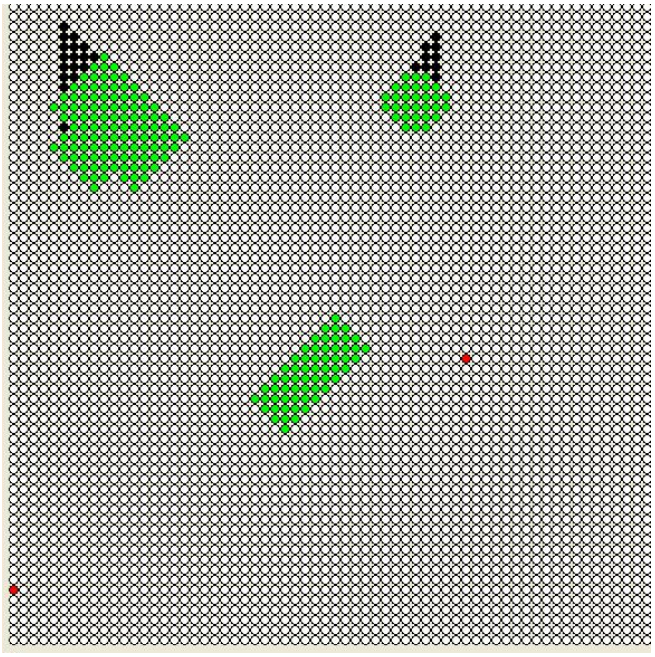

Figure 5.27b: Genetic Algorithm Results
Shadow Area: 26
Robot 1 Position: (0, 58)
Robot 2 Position: (45, 35)
Total Time: 7 seconds

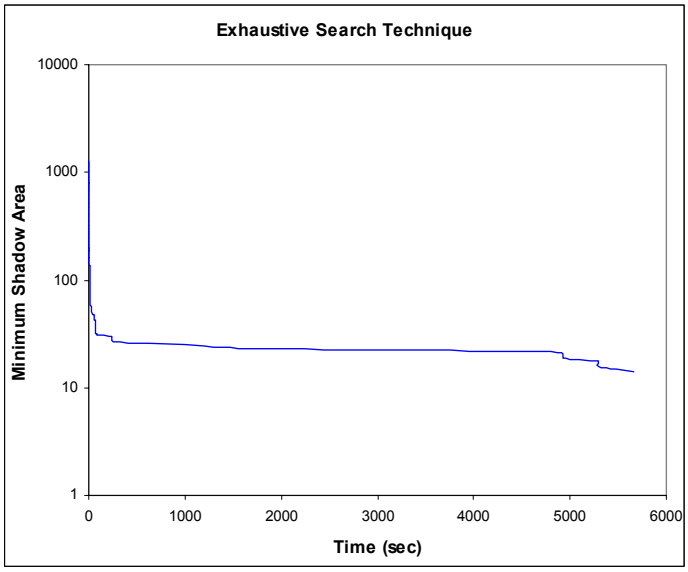

Figure 5.27c: Exhaustive search convergence rate

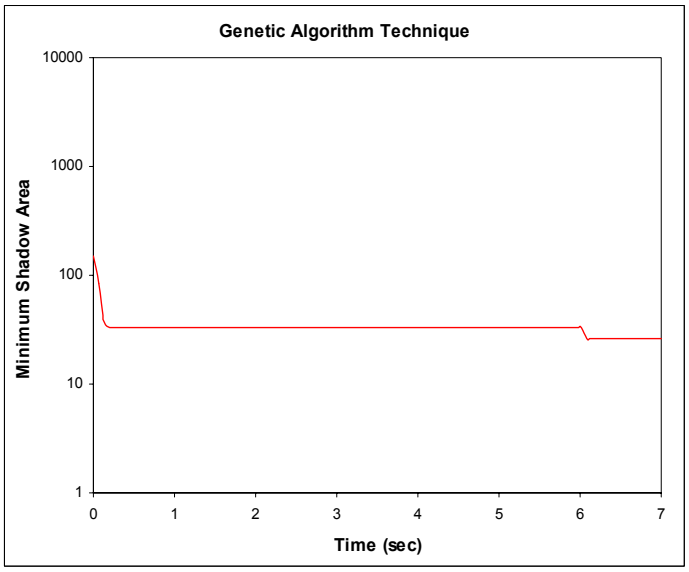

Figure 5.27d: Genetic algorithm convergence rate

Figure 5.27: Results for Comparison #1

Comparison #2

This map contains more obstacles than the previous example. The increase in the number of obstacles increased the processing time for the exhaustive search technique. The increase in the number of obstacles increases the number of calculations that must be performed throughout the search space. The results are shown in figure 5.28. The genetic algorithm ran for 100 generations and then selects the best solution from each generation. After the algorithm has cycled through all 100 generations the final solution is reported to be the best of the final generation.

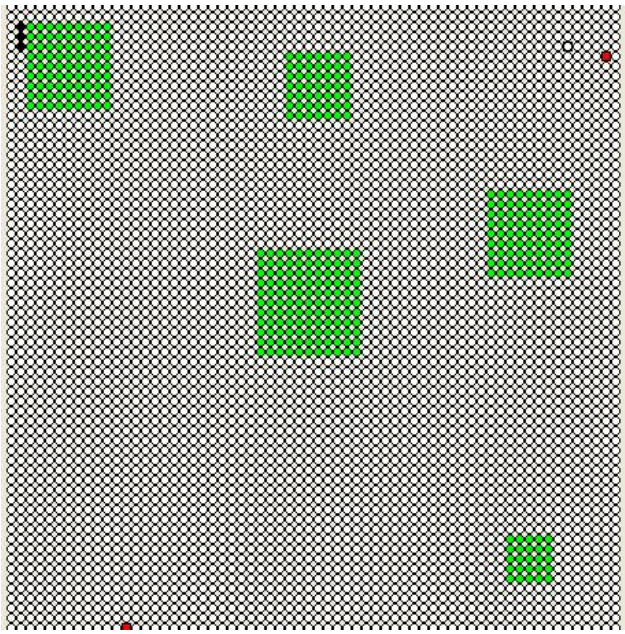

Figure 5.28a: Exhaustive Search Results
Shadow Area: 3
Robot 1 Position: (62, 5)
Robot 2 Position: (12, 63)
Total Time: 17.4 hours

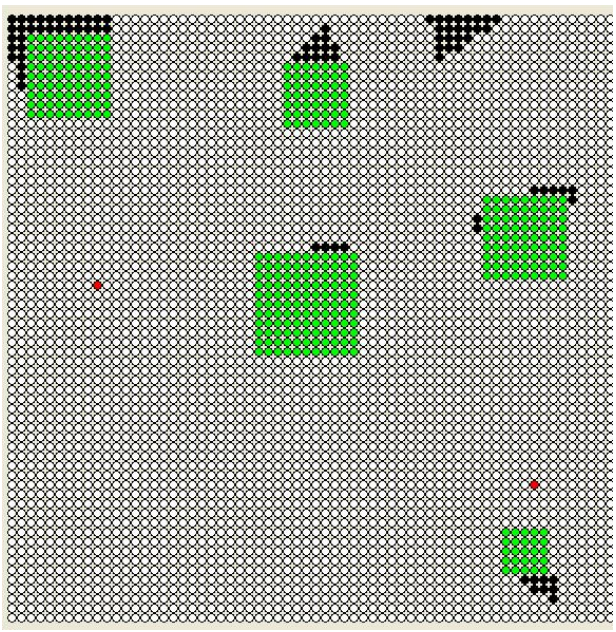

Figure 5.28b: Genetic Algorithm Results
Shadow Area: 85
Robot 1 Position: (55, 49)
Robot 2 Position: (9, 28)
Total Time: 7 seconds

Figure 5.28: Results for Comparison #2

There is a definite difference in the solution for the shadow areas. The genetic algorithm does not obtain the optimal solution, however it obtains a solution in far less time. It should also be noted that the size of the map for the exhaustive search technique

was increased from 50×50 to 64×64; this results in 1,596 extra data points that must be checked which accounts for the drastic increase in the total processing time. Initial tests were performed with a 50×50 resolution search space and the space was later increased to a resolution of 64×64 to accommodate the genetic algorithm library specifications. Plots of the convergence rates are shown in figure 5.28c-d.

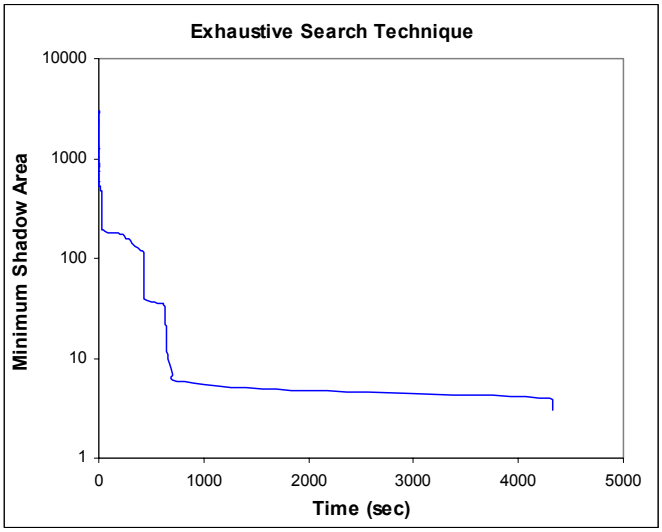

Figure 5.29a: Exhaustive search convergence rate

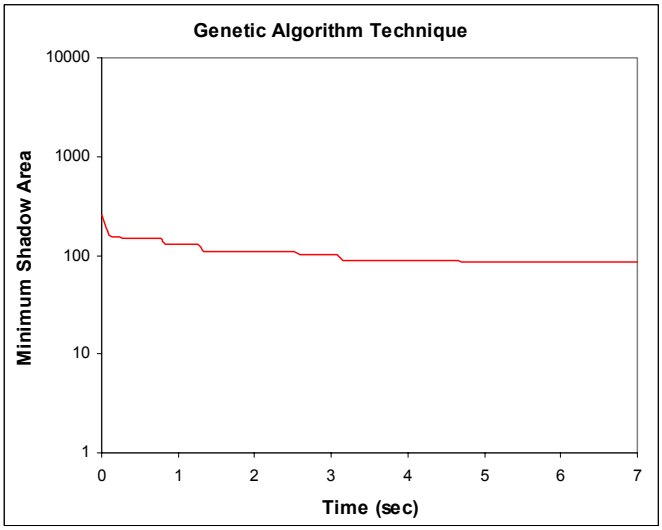

Figure 5.29b: Genetic algorithm convergence rate

Figure 5.29: Convergence Rates for Comparison #2

Comparison #3

The final map is shown in figure 5.30. There is an increase in the processing time for the exhaustive search technique due to the increase in the number of obstacles as well as the size of the search space. It should be noted that the robot positions obtained from both techniques maintain line-of-sight very narrowly. This problem can be mitigated if the boundaries of the obstacles were expanded.

Very similar results are obtained for each of the techniques. The main difference being the time in which a solution was calculated.

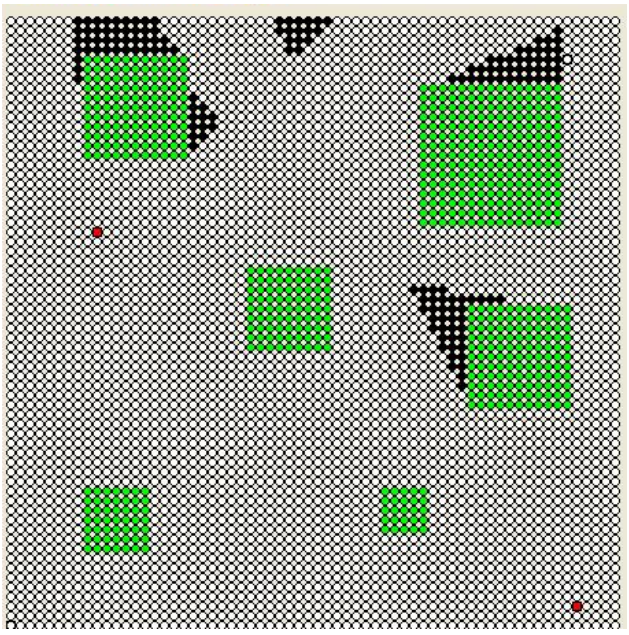

Figure 5.30a: Exhaustive Search Results
Shadow Area: 147
Robot 1 Position: (9, 22)
Robot 2 Position: (59, 61)
Total Time: 13.7 hours

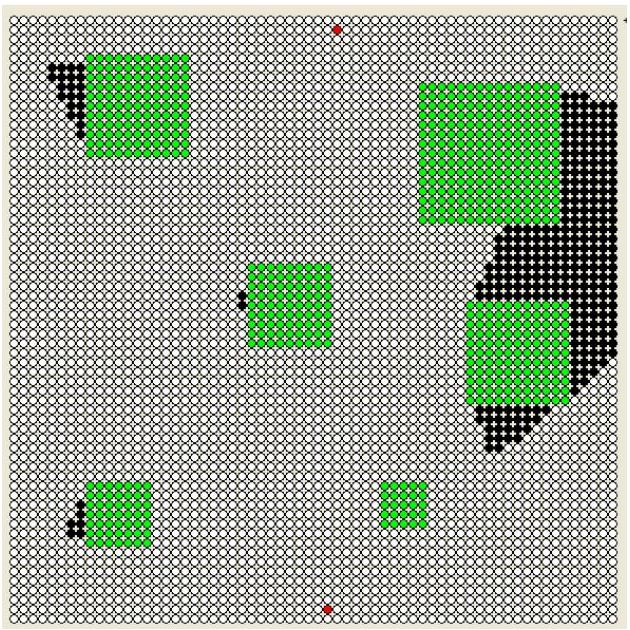

Figure 5.30b: Genetic Algorithm Results
Shadow Area: 284
Robot 1 Position: (33, 62)
Robot 2 Position: (34, 1)
Total Time: 6 seconds

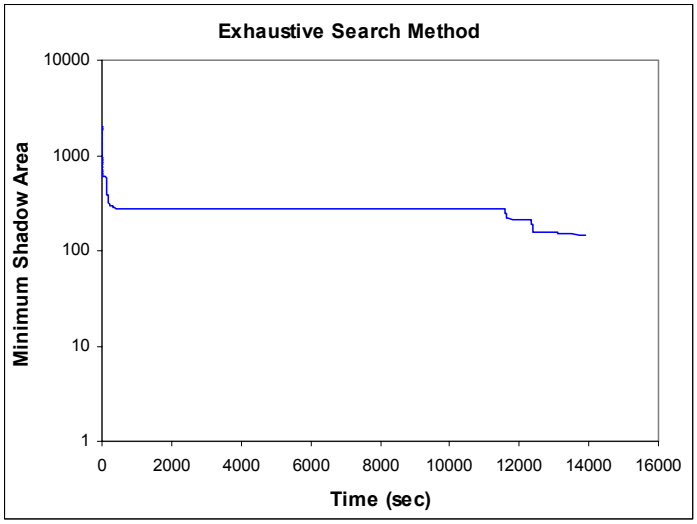

Figure 5.30c: Exhaustive search convergence rate

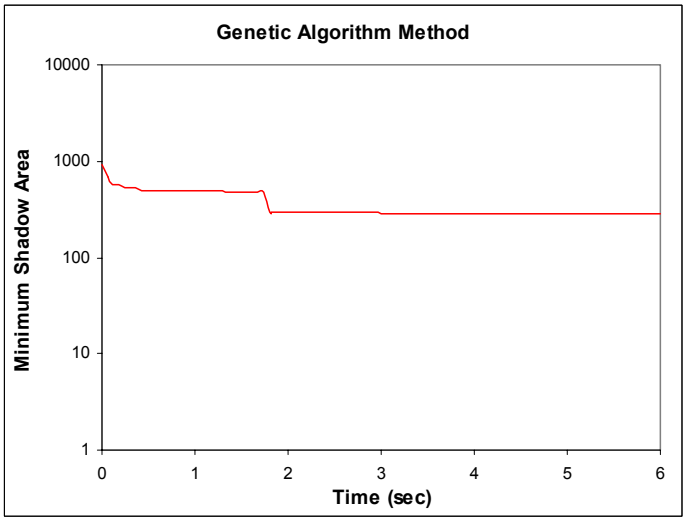

Figure 5.30d: Genetic algorithm convergence rate

Figure 5.30: Results for Comparison #3

This section presented the results obtained for two vastly different techniques. The exhaustive search technique was a starting point for this research. Initial experiments were conducted with various maps using the exhaustive search technique for the two-robot case. Collecting data using this technique was extremely laborious and time consuming. Addition of even a third robot to the problem would increase the processing time exponentially. Other techniques such as the golden section method, steepest descent technique, and Monte Carlo programming, were investigated and the genetic algorithm optimization technique was finally chosen due to the ease of implementation with the current algorithm.

The following table summarizes the specifications of the computers that were used and the specifications that the genetic algorithm required.

	64×64 Maps	**256×256 Maps**
Computer Specifications:	Pentium 3, 1GHz Processor 256 MB RAM	Pentium 4, 2GHz Processor 512 MB RAM
Genetic Algorithm:		
• Population	10	20
• Generations	100	200
• Mutation Rate	2%	2%

Table 5.1: Summary of computer specifications and genetic algorithm specifications for both types of maps

CHAPTER 6
SUMMARY AND CONCLUSION

This research demonstrates that simulation robot positioning is essential when planning for multiple robotic entities. This simulation provides insight into where the communications repeater robots should be positioned throughout the search space so that communication is maintained and coverage is maximized. This program is a very useful tool when coordinating the actions of a multi-agent system. The genetic algorithm technique provided a method for obtaining a solution for several types of maps and multiple robots in a timely fashion.

The genetic algorithm technique was compared with the exhaustive search technique. The results indicated that the exhaustive search technique provided the optimal solution for a two-robot configuration however, the time required to obtain a solution was on the order of several hours. In contrast, the genetic algorithm technique provided a sub-optimal solution for a two-robot configuration and the time required to obtain a solution was on the order of seconds. Although the genetic algorithm provides a solution in a timely manner, the global optimality of the solution is sacrificed.

The research presented is strictly a two-dimensional evaluation but can be extended to a three-dimensional problem. The three-dimensional situation will account for elevation of the terrain in addition to obstacle locations. The current research could be extended to account for the x, y, and z components of the map and thus create a three-dimensional grid. The line-of-sight algorithm would have to be revised so that it

accounts for how to determine if a path is obstructed between two grid points when dealing with three-dimensional contours. The method currently used may not be sufficient for the three-dimensional problem.

In addition, this research could be extended to incorporate ground and aerial vehicles. The system would function as follows:

1. An aerial vehicle would navigate to a desired location or waypoint.
2. Once at the waypoint, a series of photographs would be taken with an onboard camera.
3. These photographs would be discretized, similar to the way the maps were as presented in this research. The landmarks, obstacles, and boundaries would be extracted from the images to create a map.
4. This map would be used and the algorithm would determine the positions for the ground robots and the corresponding combined shadow area.

This is an example of a hybrid collaborative control system. Two specific types of robots are being controlled to achieve an overall task of maximizing the coverage of a search space while maintaining communications. This is a very simplified approach to this problem and there is a great deal of work involved before actual implementation of this type of system. Some problems are: automation of a small aerial vehicle that is able to navigate to waypoints, image processing that will extract obstacles from the photographs taken by the aerial vehicle, and having enough ground vehicles to accomplish this task.

This research has provided some initial results for robot coordination and collaborative control. The results reinforce the validity of the method as well as the technique used for optimization of the solution. The speed at which a solution is obtained is also an attractive feature of this algorithm. The genetic algorithm technique for optimization

provided a solution for this type of global optimization problem. The final algorithm was easily applicable to larger search spaces consisting of different maps and a variable number of robots. This simulation is a starting point for future research and implementation on an actual system.

This research has provided some initial results for robot coordination and collaborative control. The results reinforce the validity of the method as well as the technique used for optimization of the solution. The speed at which a solution is obtained is also an attractive feature of this algorithm. The genetic algorithm technique for optimization provided a solution for this type of global optimization problem. The final algorithm was easily applicable to larger search spaces consisting of different maps and a variable number of robots. This simulation is a starting point for future research and implementation of an actual system.

APPENDIX A
SOURCE CODE FOR FINAL ALGORITHM

This appendix contains the code written to implement a graphical user interface as well as all functions necessary to carry out all calculations. The implementation of the genetic algorithm library (GAlib) is also included. The appendix begins with the header files and then the function prototypes used throughout the body of the code.

```
////////////////////////////////////////////////////////////////////////////
//      This program was adapted from Matthew Wall's example 1 using his

//      genetic algorithm library, GAlib.

//

//      DESCRIPTION: The following program has modified ex1.c to extend to my
//                              particular application.  I am using the Simple Genetic
//                              Algorithm and a 2D binary string genome.
//                              In particular, I have modified the Objective function
//                              so that it performs shadow area calculations for a given
//                              multi-robot configuration and a specified number of
robots.
//                              The program outputs the final results to the terminal and
//                              also creates a data file that logs GA data periodically.
//                                      (GAdata.txt)

//                              A dialog window has been implemented which displays
the
//                              data to the user.

//

//      DATE:           Oct. 21, 2002
//      PROGRAMMER:     Erica Zawodny
////////////////////////////////////////////////////////////////////////////

/* robotDialog.cpp : Defines the entry point for the application.*/

#include "stdafx.h"
```

```
#include <windows.h>
#include "resource.h"

#include <math.h>
#include <time.h>
#include <string.h>
#include <ga/GASimpleGA.h>          // we're going to use the simple GA
#include <ga/GA2DBinStrGenome.h> // and the 2D binary string genome

//Erica's Stuff
#define ROW 64
#define COLUMN 64
#define DONT_INTERSECT 0
#define DO_INTERSECT 1
#define COLLINEAR 2
#define RANGE 50
#define MAX_RAND 32767.0          //maximum random number

////////////////////////////////////////////////
//          This macro is used to determine if 2 numbers //
//          have the same sign.                                                                    //
////////////////////////////////////////////////
#define SAME_SIGNS(a, b) ((((long) (a) * (long) (b)) >= 0) ? 1 : 0)

//////////////////////////
//          Function Prototypes    //
//////////////////////////
void
drawObstacles(void) ;

int
shadows(int num, int robot[][2]) ;

int
LOS(int mat1[][COLUMN],int xs, int ys, int xe, int ye) ;

int
lines_intersect(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2,
unsigned int x3, unsigned int y3, unsigned int x4, unsigned int y4) ;

void
randomRobot(int robots, int robot_positions[][2]) ;

int
multiRobotLOS(int number,int robot_loc[][2],unsigned int line_segments,unsigned int
*x3,unsigned int *y3,unsigned int *x4,unsigned int *y4) ;
```

```
void
Statistics(GAGenome& g) ;

float
Objective(GAGenome &) ;    // This is the declaration of our obj function.
                           // The definition comes later in the file.

int
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow) ;

BOOL CALLBACK DlgProc(HWND hwnd, UINT Message, WPARAM wParam,
LPARAM lParam) ;

void
DrawGrid(HDC hdc, HWND hwnd) ;

void
DrawMap(HDC hdc, HWND hwnd) ;

void
DrawRobots(int best_robots[][2], HDC hdc, HWND hwnd) ;

void
DrawShadows(HDC hdc, HWND hwnd) ;


//////////////////////////
//        Global Variables
//////////////////////////
        int obstacles_matrix[ROW][COLUMN] = {0};    //A matrix containing the
obstacle locations
        int shadow_matrix[ROW][COLUMN] = {0};
        int num_robots ;        //User-specified number of robots
        int best_robots[20][2] ;        //a global containing the best robots the GA found
        int best_shadow = 0 ;           //a global var. containing the final combined
shadow area
        int i,m,n ;
        int Solution_Found = 0 ;        //A flag that is set when the GA has found a
solution
//      FILE *fp1, *fp2 ;       //File pointers for accessing map file and segment file
        char map_file[80] = "map1.txt";                 //Default map
        char segment_file[80] = "segments1.txt" ; //Default segment file
```

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT Message, WPARAM wParam,
LPARAM lParam)
{
	//Variables for drawing in the dialog
	PAINTSTRUCT ps ;
	HDC hdc ;
	static int iSelection = IDM_MAP1 ;
	HMENU hMenu ;

	switch(Message)
	{
		hMenu = GetMenu (hwnd) ;

		case WM_INITDIALOG:
			// This is where we set up the dialog box, and initialise any default
values
			SetDlgItemInt(hwnd, IDC_NUMBER, 2, FALSE);

		break;

		case WM_PAINT:

			hdc = BeginPaint(hwnd, &ps) ;

			DrawGrid(hdc, hwnd) ;

			DrawMap(hdc, hwnd) ;

			if(Solution_Found)
			{
				DrawRobots(best_robots, hdc, hwnd) ;
				DrawShadows(hdc, hwnd) ;
			}

			EndPaint(hwnd, &ps) ;

		break ;


		case WM_COMMAND:
			switch(LOWORD(wParam))
			{
				case IDC_SOLVE:
				{
					// When somebody clicks the Solve button, first we
get the number
```

```
// they entered

//Local Variables for timer
time_t start, end ;
double diff ;
char temp[80] ;

//I will place all code to run the GA here
// Declare variables for the GA parameters and set
them to some default values.
int width    ;                          //the number
of bits
int height   ;          //the number of individuals
generated
int popsize  ;
int ngen     ;
float pmut   ;
float pcross ;

num_robots = GetDlgItemInt(hwnd,
IDC_NUMBER, NULL, FALSE) ;
width    = 12 ;                         //the number
of bits
height   = num_robots ;                 //the number
of individuals generated
popsize  = 20;
ngen     = 200;
pmut   = 0.002;
pcross = 0.9;

//->Here I will enter my GA Stuff!
//ERICA'S GA STUFF STARTS HERE

drawObstacles() ;

//Start the timer
time (&start) ;


// Now create the GA and run it.  First we create a
genome of the type that
// we want to use in the GA.  The ga doesn't operate
on this genome in the
// optimization - it just uses it to clone a population
of genomes.
```

```
                                        GA2DBinaryStringGenome genome(width, height,
Objective);

                                        // Now that we have the genome, we create the
genetic algorithm and set
                                        // its parameters - number of generations, mutation
probability, and crossover
                                        // probability.  And finally we tell it to evolve itself.

                                        GASimpleGA ga(genome);
                                        ga.populationSize(popsize);
                                        ga.nGenerations(ngen);
                                        ga.pMutation(pmut);
                                        ga.pCrossover(pcross);
                                        ga.scoreFilename("GAdata.txt");       // name of file
for scores
                                        ga.scoreFrequency(1);                 // keep
the scores of every 1 generation
                                        ga.flushFrequency(50);                //
specify how often to write the score to disk
                                        ga.selectScores(GAStatistics::AllScores);
                                        ga.evolve();

                                        time (&end) ;                         //Stop
timer
                                        diff = difftime(end, start) ;     //Times functions

                                        //sprintf(temp,"Total Processing Time: %f ",diff) ;
                                        //MessageBox(hwnd, temp, "Processing Time",
MB_OK) ;

                                        //Display the Processing Time to the dialog window
                                        SetDlgItemInt(hwnd, IDC_TIME, diff, FALSE) ;

                                        //Call statistics function

        Statistics((GAGenome&)ga.statistics().bestIndividual()) ;

                                        //Set the Solution_Found flag
                                        Solution_Found = 1 ;

                                        //Display the combined shadow area to the dialog
window
                                        SetDlgItemInt(hwnd, IDC_SHADOW,
best_shadow, FALSE) ;
```

```
                                        //Clear the list box
                                        SendDlgItemMessage(hwnd, IDC_LIST,
LB_RESETCONTENT, 0, 0);

                                        //Display the best robots to the dialog window (list
box)
                                        for(m=0; m<num_robots; m++)
                                        {
                                                sprintf(temp, "%d     %d",
best_robots[m][0], best_robots[m][1]) ;
                                                int index = SendDlgItemMessage(hwnd,
IDC_LIST, LB_ADDSTRING, 0, (LPARAM)temp);
                                        }

                                        InvalidateRect(hwnd, NULL, TRUE) ;
                                        UpdateWindow(hwnd) ;

                                        //Create a message box that lets user know when
calculations are complete
                                        MessageBox(hwnd, "The GA successfully found a
solution.", "Solution Found", MB_OK) ;

                                        //ERICA'S GA STUFF ENDS HERE

                                }
                                break;

                                //Menu stuff STARTS here
                                case IDM_ABOUT:
                                        MessageBox(hwnd, "Written by: Erica Zawodny",
"About RobotDialog", MB_ICONINFORMATION | MB_OK) ;

                                break ;

                                case IDM_MAP1:
                                        sprintf(map_file, "map1.txt") ;
                                        sprintf(segment_file, "segments1.txt") ;
                                        Solution_Found = 0 ;
                                        InvalidateRect(hwnd, NULL, TRUE) ;
                                        UpdateWindow(hwnd) ;
                                        return 0;
                                case IDM_MAP2:
                                        sprintf(map_file, "map2.txt") ;
                                        sprintf(segment_file, "segments2.txt") ;
                                        Solution_Found = 0 ;
                                        InvalidateRect(hwnd, NULL, TRUE) ;
```

```
                                UpdateWindow(hwnd) ;
                                return 0 ;
                        case IDM_MAP3:
                                sprintf(map_file, "map3.txt") ;
                                sprintf(segment_file, "segments3.txt") ;
                                Solution_Found = 0 ;
                                InvalidateRect(hwnd, NULL, TRUE) ;
                                UpdateWindow(hwnd) ;
                                return 0 ;
                        case IDM_MAP4:
                                sprintf(map_file, "map4.txt") ;
                                sprintf(segment_file, "segments4.txt") ;
                                Solution_Found = 0 ;
                                InvalidateRect(hwnd, NULL, TRUE) ;
                                UpdateWindow(hwnd) ;
                                return 0 ;
                        case IDM_MAP5:
                                sprintf(map_file, "map5.txt") ;
                                sprintf(segment_file, "segments5.txt") ;
                                Solution_Found = 0 ;
                                InvalidateRect(hwnd, NULL, TRUE) ;
                                UpdateWindow(hwnd) ;
                                return 0 ;
                        case IDM_MAP6:
                                sprintf(map_file, "map6.txt") ;
                                sprintf(segment_file, "segments6.txt") ;
                                Solution_Found = 0 ;
                                InvalidateRect(hwnd, NULL, TRUE) ;
                                UpdateWindow(hwnd) ;
                                return 0 ;

                        case IDM_EXIT:
                                EndDialog(hwnd, IDM_EXIT);      //Exits the
program when Exit option is chosen
                        break ;
                        //Menu stuff ENDS here

                        case IDC_EXIT:
                                EndDialog(hwnd, IDC_EXIT);      //Exits the
program when Exit button is pressed
                        break;
                }
        break;

        case WM_CLOSE:
                EndDialog(hwnd, 0);
```

```
                break;
                default:
                        return FALSE;
        }

        return TRUE ;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{

        return DialogBox(hInstance, MAKEINTRESOURCE(IDD_MAIN), NULL,
DlgProc) ;

}


//////////////////////////////////////////////////////////////////////////
//////////////////////////LIST OF FUNCTIONS/////////////
//////////////////////////////////////////////////////////////////////////

// This is the objective function.  All it does is check for alternating 0s and
// 1s.  If the gene is odd and contains a 1, the fitness is incremented by 1.
// If the gene is even and contains a 0, the fitness is incremented by 1.  No
// penalties are assigned.
//   We have to do the cast because a plain, generic GAGenome doesn't have
// the members that a GA2DBinaryStringGenome has.  And it's ok to cast it
// because we know that we will only get GA2DBinaryStringGenomes and
// nothing else.

float
Objective(GAGenome& g)
{
        GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
        float score=0.0;
        int count=0;
        int m, n ;
        int robot[20][2] ;


        //Zero the robot 2d array
        for(m=0; m<20 ; m++)
        {
                for(n=0; n<2; n++)
                {
```

```
                        robot[m][n] = 0 ;
                }
        }

/*
        for(int i=0; i<genome.width(); i++){
                for(int j=0; j<genome.height(); j++){
                        if(genome.gene(i,j) == 0 && count%2 == 0)
                                score += 1.0;
                        if(genome.gene(i,j) == 1 && count%2 != 0)
                                score += 1.0;
                                count++;
                }
        }
        return score;
*/

        //Convert the binary string to integers
        for(m=0 ; m<genome.height() ; m++)
        {
                for(n=(genome.width()/2)-1 ; n>=0 ; n--)
                {
                        robot[m][0] += genome.gene((genome.width()/2)-1 - n, m) *
pow(2,n) ;
                        robot[m][1] += genome.gene((genome.width()-1) - n, m) *
pow(2,n) ;
                }
        }

        //Call the shadow area function
        score = ROW*COLUMN - shadows(num_robots, robot) ;

        return score ;
}


void
Statistics(GAGenome& g)
{
        GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
        float score=0.0;
        int count=0;
        int m, n ;

        FILE *fp ;
```

```
	fp = fopen("debug.out","w") ;

	//Zero the robot 2D array
	for(m=0; m<20 ; m++)
	{
		for(n=0; n<2; n++)
		{
			best_robots[m][n] = 0 ;
		}
	}

	fprintf(fp, "%d \n", genome.height()) ;
	//Convert the binary string to integers
	for(m=0 ; m<genome.height() ; m++)
	{
		for(n=(genome.width()/2)-1 ; n>=0 ; n--)
		{
			best_robots[m][0] += genome.gene((genome.width()/2)-1 - n, m) *
pow(2,n) ;
			best_robots[m][1] += genome.gene((genome.width()-1) - n, m) *
pow(2,n) ;

		}
		fprintf(fp, "%d %d\n", best_robots[m][0], best_robots[m][1]) ;
	}

	//Call the shadow area function
	best_shadow = shadows(num_robots, best_robots) ;
	fprintf(fp, "%d \n", best_shadow) ;

	fclose(fp) ;


}

////////////////////////////////////////////////////////////////////////////
//FUNCTION NAME:	drawObstacles
//DESCRIPTION:		This function opens a file that contains obstacle location
//			information.  This info. is placed into a matrix
//		called 'obstacles_matrix'.
////////////////////////////////////////////////////////////////////////////
void drawObstacles(void)
{

	int i = ROW;
```

```
	int j = COLUMN;
	FILE *fp;
	int m, n;

	//Open the file to be read

	fp=fopen(map_file, "r");	/*******WARNING: map_file is a global
var.**********/


	//Scans the file and colors in the points until it reaches the end of the file
	while (fscanf(fp, "%i%i", &m, &n) > 0)
	{
		if(m<ROW && n<COLUMN)
		{

			obstacles_matrix[m][n] = 1;   //NOTE: Check on this (LOS acts
weird; Inverted Map??)
		}
	}

	//Close the file
	fclose(fp);

}

////////////////////////////////////////////////////////////////////////////
//FUNCTION NAME:		shadows
                                                        //
//DESCRIPTION:		This function will calculate the shadow areas for one robot
	//
//			testing every point in the matrix for line-of-sight.
	//
//			It will cycle through every possible robot position and store
	//
//		the shadow area values to shadow_matrix.  This info. will
	//
//			later be saved to a data file
                                        //
////////////////////////////////////////////////////////////////////////////
int
shadows(int num, int robot[][2])
{
	//Local Variables
//	unsigned int robotX[100];				//the current robot position in
the x
```

```
//      unsigned int robotY[100];                       //the current robot position in
the y
//      int local_shadows[ROW][COLUMN] = {0};//A matrix contaning shadow info.
locally
//      int dummy[ROW][COLUMN] = {0};                   //Passed to the LOS
function
        unsigned int i,j,k,n;                           //counters
        unsigned int x3[100], y3[100], x4[100], y4[100] ;       //line segments that
make up the obstacles
        unsigned int segments = 0 ;
//      int total_shadow[ROW][COLUMN] = {0} ;
        int current_shadow[ROW][COLUMN] = {0} ;
        int previous_shadow[ROW][COLUMN] = {0} ;
        int shadow = 0 ;

        FILE *fp1 ;

        /*
        for(i=0; i<num; i++)
        {
                printf("Robot Positions: %d %d \n", robot[i][0], robot[i][1]) ;
        }
        */

        //Open the file containing the line segments to be
        //checked for segment intersection
        fp1=fopen(segment_file, "r") ;          /*******WARNING: segment_file
is a global var.**********/

        //Scans until it reaches the end of file
        while (fscanf(fp1, "%i%i%i%i", &x3[segments], &y3[segments], &x4[segments],
&y4[segments]) > 0)
        {
                segments+=1 ;
        }
        segments -= 1 ;

/*      //Open the file containing the # of robots and robot positions
        fp3 = fopen("robotPositions.txt", "r") ;
        fscanf(fp3, "%i", &num) ;               //This is the number of robots
        for(n=0; n<num; n++) fscanf(fp3, "%i%i", &robot[n][0], &robot[n][1]) ;
*/

        //Check to see if robot positions lie within obstacle
        for(i=0; i<num; i++)
        {
```

```
                if(obstacles_matrix[robot[i][0]][robot[i][1]]==1)
                {
                        shadow = ROW*COLUMN ;
                        //Print out the final, combined shadow area
                        //printf("Combined Shadow Area: %i\n", shadow) ;
                        return shadow ;
                }
        }

        if(multiRobotLOS(num,robot,segments,x3,y3,x4,y4))
        {
                for(n=0; n<num; n++)
                {
                        for(i=0; i<ROW; i++)
                        {
                                for(j=0; j<COLUMN; j++)
                                {
                                        current_shadow[i][j] = 0 ;              //Clear out the
current shadow matrix
                                        shadow_matrix[i][j] = 0 ;               //clear out
global combined shadow matrix
                                }
                        }

                        //Begin calculations
                        for(i=0; i<ROW; i++)
                        {
                                for(j=0; j<COLUMN; j++)
                                {
                                        if(!obstacles_matrix[i][j]==1)
                                        {
                                                int done = 0 ;
                                                k = 0 ;
                                                while(!done)
                                                {
                                                        //Call lines_intersect function
(instead of LOS function)
                                                        if(lines_intersect(robot[n][0],
robot[n][1], i, j, x3[k], y3[k], x4[k], y4[k]) == 1)
                                                        {

                                                                current_shadow[i][j] = 1;
                                                                done = 1 ;
                                                        }
                                                        else if (!(k < segments)) done = 1 ;
                                                        k++ ;
```

```
                                        }
                                }
                        }
                }
                for(i=0; i<ROW; i++)
                {
                        for(j=0; j<COLUMN; j++)
                        {
                                if(n==0) previous_shadow[i][j] =
current_shadow[i][j] ;
                                else previous_shadow[i][j] = previous_shadow[i][j]
&& current_shadow[i][j] ;
                        }
                }
        }
        //Count up the shadow points in previous_shadow matrix
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COLUMN; j++)
                {
                        if(previous_shadow[i][j]==1)
                        {
                                shadow++ ;
                                shadow_matrix[i][j] = 1 ; //a global matrix
containing combined shadows
                        }
                }

        }

}

else shadow = ROW*COLUMN ;

//Print out the final, combined shadow area
//printf("Combined Shadow Area: %i\n", shadow) ;

fclose(fp1) ;
//      fclose(fp2) ;

return shadow ;

}

//////////////////////////////////////////////////////////////////
//DATE:                 Aug. 20, 2002
```

```
//NOTE:               I have modified this program such that it returns a '0'
//                    if there is no intersection and a '1' if the segments do
//                    intersect.  This function will open a file containing the
//                    line segments that make up the obstacles in a map and check
//                    for intersection.
//////////////////////////////////////////////////////////////////////
/* lines_intersect:  AUTHOR: Mukesh Prasad
 *
 *   This function computes whether two line segments,
 *   respectively joining the input points (x1,y1) -- (x2,y2)
 *   and the input points (x3,y3) -- (x4,y4) intersect.
 *   If the lines intersect, the output variables x, y are
 *   set to coordinates of the point of intersection.
 *
 *   All values are in integers.  The returned value is rounded
 *   to the nearest integer point.
 *
 *   If non-integral grid points are relevant, the function
 *   can easily be transformed by substituting floating point
 *   calculations instead of integer calculations.
 *
 *   Entry
 *        x1, y1,  x2, y2   Coordinates of endpoints of one segment.
 *        x3, y3,  x4, y4   Coordinates of endpoints of other segment.
 *
 *   Exit
 *        x, y              Coordinates of intersection point.
 *
 *   The value returned by the function is one of:
 *
 *        DONT_INTERSECT    0
 *        DO_INTERSECT      1
 *        COLLINEAR         2
 *
 * Error conditions:
 *
 *     Depending upon the possible ranges, and particularly on 16-bit
 *     computers, care should be taken to protect from overflow.
 *
 *     In the following code, 'long' values have been used for this
 *     purpose, instead of 'int'.
 *
 *////////////////////////////////////////////////////////////////////////

int
```

```
lines_intersect(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2,
unsigned int x3, unsigned int y3, unsigned int x4, unsigned int y4)

{
   long a1, a2, b1, b2, c1, c2; /* Coefficients of line eqns. */
   long r1, r2, r3, r4;         /* 'Sign' values */
   long denom, offset, num;     /* Intermediate values */

         //Check to see if the two points overlap
         if(x1==x2 && y1==y2) return DONT_INTERSECT ;

   /* Compute a1, b1, c1, where line joining points 1 and 2
    * is "a1 x  +  b1 y  +  c1  =  0".
    */

   a1 = y2 - y1;
   b1 = x1 - x2;
   c1 = x2 * y1 - x1 * y2;

   /* Compute r3 and r4.
    */


   r3 = a1 * x3 + b1 * y3 + c1;
   r4 = a1 * x4 + b1 * y4 + c1;

   /* Check signs of r3 and r4.  If both point 3 and point 4 lie on
    * same side of line 1, the line segments do not intersect.
    */

   if ( r3 != 0 &&
        r4 != 0 &&
        SAME_SIGNS( r3, r4 ))
      return ( DONT_INTERSECT );

   /* Compute a2, b2, c2 */

   a2 = y4 - y3;
   b2 = x3 - x4;
   c2 = x4 * y3 - x3 * y4;

   /* Compute r1 and r2 */

   r1 = a2 * x1 + b2 * y1 + c2;
   r2 = a2 * x2 + b2 * y2 + c2;
```

```
   /* Check signs of r1 and r2.  If both point 1 and point 2 lie
    * on same side of second line segment, the line segments do
    * not intersect.
    */

   if ( r1 != 0 &&
        r2 != 0 &&
        SAME_SIGNS( r1, r2 ))
       return ( DONT_INTERSECT );

   /* Line segments intersect: compute intersection point.


   denom = a1 * b2 - a2 * b1;
   if ( denom == 0 )
      return ( COLLINEAR );
   offset = denom < 0 ? - denom / 2 : denom / 2;

   /* The denom/2 is to get rounding instead of truncating.  It
    * is added or subtracted to the numerator, depending upon the
    * sign of the numerator.


   num = b1 * c2 - b2 * c1;
   *x = ( num < 0 ? num - offset : num + offset ) / denom;

   num = a2 * c1 - a1 * c2;
   *y = ( num < 0 ? num - offset : num + offset ) / denom;
         */

   else return ( DO_INTERSECT );
} /* lines_intersect */


////////////////////////////////////////////////////////////////
//FUNCTION NAME:randomRobot
                  //
//DATE:                         Sept. 20, 2002
                  //
//DESCRIPTION:      This function will take the value entered by  //
//                              the user and generate that many random robot          //
//                              positions.  The positions will be stored to an //
//                              array called robot_positions.
          //
////////////////////////////////////////////////////////////////
void
```

```
randomRobot(int robots, int robot_positions[][2])
{

        int x, y ;                  //Robot's x and y position
        int i ;

        for(i=0; i<robots; i++)
        {

                x = (rand()/MAX_RAND) * RANGE ;         //generates a random # from
0 to 49
                robot_positions[i][0] = x ;

                y = (rand()/MAX_RAND) * RANGE ;
                robot_positions[i][1] = y ;

        }

}

////////////////////////////////////////////////////////////////
//FUNCTION NAME:multiRobotLOS
//DATE:                         Sept. 25, 2002

//DESCRIPTION:      This function will determine if there is an
//                              unobstructed path between two robots.  If the
//                              path is blocked the function will return a '0',
//                              (FALSE) otherwise it will return a '1', (TRUE).
////////////////////////////////////////////////////////////////
int
multiRobotLOS(int number,int robot_loc[][2],unsigned int line_segments,unsigned int
*x3,unsigned int *y3,unsigned int *x4,unsigned int *y4)
{
        int robot1, robot2 ;
        int line_of_sight ;             //A flag that's set to '1' when the path between 2
robots isn't blocked
        unsigned int k ;

        for(robot1=0; robot1<number; robot1++)
        {
                line_of_sight = 1 ;     //set the line-of-sight flag

                for(robot2=0; robot2<number ; robot2++)
                {
                        if(!robot1==robot2)
                        {
```

```
                                for(k=0; k<line_segments ; k++)
                                {

        if(lines_intersect(robot_loc[robot1][0],robot_loc[robot1][1], robot_loc[robot2][0],
robot_loc[robot2][1],x3[k],y3[k],x4[k],y4[k]))
                                        {
                                                line_of_sight = 0 ;
                                        }
                                }
                        }

                }
                if (!line_of_sight) return 0 ;
        }

        return 1 ;


}



////////////////////////////////////////////////////////////////////////
//FUNCTION NAME:        DrawGrid
//DESCRIPTION:          This function draws a blank grid to the window.  The
//                              size of the grid can be modified via the define
statements
//                              at the top of the program (ROW, COLUMN).
////////////////////////////////////////////////////////////////////////
void DrawGrid(HDC hdc, HWND hwnd)
{
        int i = ROW;
        int j = COLUMN;
        int w, h;               //Window width and height counters
        int t;                  //Based on the shape of the window, t is used to ensure the
ellipses are acutally circles.
        HPEN hpen, hpenOld ;
        HBRUSH hbrush, hbrushOld ;
        RECT dimensions ;  //Variable to a structure containing the window dimensions

        //Get the Current Window Size
        GetClientRect(hwnd, &dimensions) ;

        //Create pen
        hpen = CreatePen(PS_SOLID, 1, RGB(0,0,0));
```

```
	//Select the pen
	SelectObject(hdc, hpen) ;

	//Draw an Array of Circles in window
	w = (dimensions.right-100) / COLUMN;
	h = dimensions.bottom / ROW;

	if (w > h) t=h;

	else t=w;

	for (i=0; i < ROW; i++)
	{
		for (j=0; j < COLUMN; j++)
		{
			Ellipse(hdc,t*i+10,t*j+10,t*(i+1)+10,t*(j+1)+10);

		}
	}

}


////////////////////////////////////////////////////////////////////////////
//FUNCTION NAME:	DrawMap
//DESCRIPTION:		This function opens a file that contains obstacle location
//				information.  It then re-draws the obstacle and
colors the
//				circle green and displays it to the window.
////////////////////////////////////////////////////////////////////////////
void DrawMap(HDC hdc, HWND hwnd)
{

	int i = ROW;
	int j = COLUMN;
	FILE *fp;
	int m, n;
	int w, h;		//Window width and height counters
	int t;			//Based on the shape of the window, t is used to ensure the
ellipses are acutally circles.
	HPEN hpen, hpenOld ;
	HBRUSH hbrush, hbrushOld ;
	RECT dimensions ;  //Variable to a structure containing the window dimensions

	//Get the Current Window Size
	GetClientRect(hwnd, &dimensions) ;
```

```
	//Create a Brush
	hbrush = CreateSolidBrush(RGB(0,255,0));		//A Green Brush
	//Select the brush
	SelectObject(hdc, hbrush) ;

	//Re-Draw the Obstacle Points in window
	w = (dimensions.right-100) / COLUMN;
	h = dimensions.bottom / ROW;

	if (w > h) t=h;

	else t=w;


	//Open the file to be read

		fp=fopen(map_file, "r");	/*******WARNING: map_file is a global
var.**********/

	//Scans the file and colors in the points until it reaches the end of the file
	while (fscanf(fp, "%i%i", &m, &n) > 0)
	{
		if(m<ROW && n<COLUMN)
		{
			Ellipse(hdc,t*m+10, t*n+10, t*(m+1)+10, t*(n+1)+10);
			obstacles_matrix[m][n] = 1;
		}
	}

	//Close the file
	fclose(fp);

}

////////////////////////////////////////////////////////////////////////////
//FUNCTION NAME:	DrawRobots
//DESCRIPTION:		This function retrieves the best individuals from the GA.
//				The corresponding robot positions on the grid will
be
//				designated by a red grid point.
////////////////////////////////////////////////////////////////////////////
void DrawRobots(int best_robots[][2], HDC hdc, HWND hwnd)
{
	int w, h;		//Window width and height counters
```

```
	int t;			//Based on the shape of the window, t is used to ensure the
ellipses are acutally circles.

	//GA2DBinaryStringGenome & genome = (GA2DBinaryStringGenome &)g;
	//float score=0.0;
	//int count=0;
	int m, n ;

	HPEN hpen, hpenOld ;
	HBRUSH hbrush, hbrushOld ;
	RECT dimensions ;  //Variable to a structure containing the window dimensions

	//Get the Current Window Size
	GetClientRect(hwnd, &dimensions) ;

	//Create a Brush
	hbrush = CreateSolidBrush(RGB(255,0,0));		//A Green Brush
	//Select the brush
	SelectObject(hdc, hbrush) ;

	//Draw the robot positions in window
	w = (dimensions.right-100) / COLUMN;
	h = dimensions.bottom / ROW;

	if (w > h) t=h;

	else t=w;

	for(m=0; m<num_robots; m++)
	{
		Ellipse(hdc, t*best_robots[m][0]+10, t*best_robots[m][1]+10,
t*(best_robots[m][0]+1)+10, t*(best_robots[m][1]+1)+10);
	}
}

//////////////////////////////////////////////////////////////////////
//Function Name:	DrawShadows

//Description:  This function will draw the combined shadow area of
//			robot 1 and robot 2 to the window.  It will fill in
//			any points turned ON (1) in shadow_matrix.
//////////////////////////////////////////////////////////////////////
void DrawShadows(HDC hdc, HWND hwnd)
{
	int m, n;
	int w, h;		//Window width and height counters
```

```
        int t;                          //Based on the shape of the window, t is used to ensure the
ellipses are acutally circles.

        HPEN hpen, hpenOld ;
        HBRUSH hbrush, hbrushOld ;
        RECT dimensions ;  //Variable to a structure containing the window dimensions

        //Get the Current Window Size
        GetClientRect(hwnd, &dimensions) ;

        //Create a Brush
        hbrush = CreateSolidBrush(RGB(0,0,0));                  //A Black Brush
        //Select the brush
        SelectObject(hdc, hbrush) ;

        //Re-Draw the Shadow Points in window
        w = (dimensions.right-100) / COLUMN;
        h = dimensions.bottom / ROW;

        if (w > h) t=h;

        else t=w;

        for(m=0; m<ROW; m++)
        {
                for(n=0; n<COLUMN; n++)
                {

                        if(shadow_matrix[m][n]==1)
                        {
                                Ellipse(hdc, t*m+10, t*n+10, t*(m+1)+10, t*(n+1)+10);
                        }
                }
        }

}
```

APPENDIX B
MAP MAKER SOURCE CODE

This appendix contains the code that was used to generate most of the maps used in this research.

```
/*
        Written by:   Erica Zawodny
        Date:         Oct. 14, 2002
        Title:        maps
        Description:  This program will produce a map using the file
          containing the map data. (mapdata.txt)

        Files Read:       mapdata.txt

              N                 //# of obstacles
                   radius1 xcenter1 ycenter1 //for obstacle 1
                   radius2 xcenter2 ycenter2 //for obstacle 2
                                                          ....
                                                          ....

                 radiusN xcenterN ycenterN
    Files Written:  map.txt

    Contains the (x,y) points that make up each obstacle
                   segments.txt
                   x1 y1 x2 y2       - segment 1
                   x2 y2 x3 y3       - segment 2
                   ...
                   ...
                   xN yN x1 y1      - Final segment
                   NOTE:The final segment closes the polygon ;
                   The method used here is specific to square-shaped obstacles.

    */

    #include "stdafx.h"
    #include <stdio.h>
    #include <math.h>
    #include <time.h>
    #include <stdlib.h>
```

```
#include <iostream.h>

#define RANGE 50
#define MAX_RAND 32767.0     //maximum random number
#define SIZE 5               //maximum size (radius) of an obstacle

/*Function Protoypes*/
void
randomMap() ;

/*Global Variables*/

int main(int argc, char* argv[])
{
     //cout << "Please enter the desired number of obstacles: " ;
     //cin >> num_obstacles ;

     randomMap() ;

     printf("The obstacle information was written to:     map.txt\n") ;
     printf("The line segment information was written to: segments.txt\n") ;

     return 0 ;
}

/*
FUNCTION NAME:       randomMap

DATE:                Oct. 14, 2002

DESCRIPTION:    This function will take the value entered by
                     the user and generate that many random
                     obstacles.  The positions will be stored to a
                     file called 'map.txt'
                     This function will also save the line segments
                     that make up the boundary of each obstacle to
                     a file named 'segments.txt'.
*/
void
randomMap()
{

     int xcenter, ycenter ;          //Obstacle's center point
     int i,j,k ;
     int radius ;                         //Radius of the obstacle
     int num_obstacles ;                  //User specified number of obstacles
```

```
	FILE *fp1 ;
	FILE *fp2 ;
	FILE *fp3 ;

	/*Open the files to be written to or read*/
	fp1 = fopen("map.txt", "w") ;
	fp2 = fopen("mapdata.txt", "r") ;
	fp3 = fopen("segments.txt", "w") ;

	fscanf(fp2, "%d", &num_obstacles) ;


	for(i=0; i<num_obstacles; i++)
	{
		fscanf(fp2, "%d %d %d", &radius, &xcenter, &ycenter) ;
	/*	//Generate the center points for each obstacle
		xcenter = (rand()/MAX_RAND) * RANGE ;		//generates a random
# from 0 to 49
		ycenter = (rand()/MAX_RAND) * RANGE ;

		//Generate the radius of each obstacle
		radius = (rand()/MAX_RAND) * SIZE ;		//generates a random
# from 0 to 5
		radius += 2 ;						//the radius
will be at least to points
	*/
		for(j=abs(xcenter-radius); j<=xcenter+radius; j++)
		{
			for(k=abs(ycenter-radius); k<=ycenter+radius; k++)
			{
				fprintf(fp1, "%d %d\n", j, k) ;
			}
		}

		/*Write the line segments that make up each obstacle to file:
segments.txt*/
		fprintf(fp3, "%d %d %d %d\n", xcenter-radius,ycenter-
radius,xcenter+radius,ycenter-radius) ;
		fprintf(fp3, "%d %d %d %d\n", xcenter+radius,ycenter-
radius,xcenter+radius,ycenter+radius) ;
		fprintf(fp3, "%d %d %d %d\n", xcenter+radius,ycenter+radius,xcenter-
radius,ycenter+radius) ;
		fprintf(fp3, "%d %d %d %d\n", xcenter-radius,ycenter+radius,xcenter-
radius,ycenter-radius) ;
```

```
        }
        fclose(fp1) ;
        fclose(fp2) ;
        fclose(fp3) ;

}
```

LIST OF REFERENCES

1. Prasad, M., Arvo, J., *Graphics Gems II*, Academic Press, Boston, 1991.

2. Chinneck, J. W., "Practical Optimization: A Gentle Introduction," http://www.sce.carleton.ca/faculty/chinneck/po.html, 10/21/2002.

3. Conley, W., *Computer Optimization Techniques*, Petrocelli Books, Inc., New York, 1980.

4. Daniels R. W., *An Introduction to Numerical Methods and Optimization Techniques*, North-Holland, Inc., New York 1978.

5. Dixon K., Dolan J., Huang W., Khosla P., Paredis C., "RAVE: A Real and Virtual Environment for Multiple Robot Systems," *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, vol. 3, pp. 1360-1367, 1999.

6. Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

7. Holland, J. H., "Genetic Algorithms," http://www.arch.columbia.edu/DDL/cad/A4513/S2001/r7/, 1/16/2003.

8. Meiszer K.A., "Using a Genetic Algorithm to Determine Optimal Beacon Placement for a Beacon Navigation System," Master's Thesis, University of Florida, 2001.

9. Nilsson, N. J., *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

10. O'Rourke, J., *Computational Geometry in C*, 2nd Edition, Cambridge University Press, New York, New York, 1998.

11. Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design with Applications*, McGraw Hill, New York, 1984.

12. Wall, M., "GAlib: A C++ Library of Genetic Algorithm Components," http://lancet.mit.edu/ga, 11/14/2002.

BIOGRAPHICAL SKETCH

Erica Frances Zawodny was born on July 6, 1978, of parents Joseph F. and Yvonne D. Zawodny in Miami, FL. She graduated from MAST Academy High School in June 1996 and shortly thereafter began attending college at the University of Florida. She received a Bachelor of Science in Mechanical Engineering with honors in August 2000. She began studying for a Master of Science degree in the fall of 2000 and will be receiving that degree in May 2003. After graduation she intends to begin PhD work at the University of Florida in the Department of Mechanical and Aerospace Engineering in the field of robotics with an emphasis on collaborative robotic control.