

# Genetic Algorithm for Pathfinding in Robots

Ryan McGreevy

April 29, 2009

## **Abstract**

In our project, a genetic algorithm was created to perform pathfinding for a robot in order for it to find its way through a set of obstacles to a specified goal. Traditional pathfinding algorithms such as A\* can be computationally intensive but generate optimal paths, a feature not needed for this project. A set of robots demonstrating predator-prey dynamics was the original goal of the project, so less than optimal and more natural appearing paths were desired over perfect and rigid ones. The algorithm developed produces correct and generally near optimal paths, though future work will include efforts to make optimal outcomes more certain especially when dealing with complex obstacles.

## **1 Introduction**

Genetic algorithms are heuristic search algorithms in the guided random search category created by John Holland in 1975[2]. Genetic algorithms are

based off of natural selection dynamics of living organisms and the genetic principles that guide the process. In the natural selection of a species, environmental stress is applied to the organisms such that the fittest individuals survive to reproduce while the weakest die[2]. The offspring contain a mix of genetic material from both parents such that hopefully the children of fit parents will themselves be fit, though this is not always the case[2].

These basic biological principles can be applied to the development of algorithms in an attempt to guide the search to finding a more fit, i.e. better, solution to the given problem[2]. Data representing parts of a possible solution can be thought of as a gene, where an entire solution is made up of multiple genes constituting a chromosome[1]. These chromosomes are then generally used to represent an individual in the population, though more complex algorithms can be created in which individuals are made up of multiple chromosomes[1]. The simplest way of representing a gene is John Holland's original approach of using bit strings[1]. These strings of data can then be manipulated using a few methods borrowed from the biological process of natural selection to hopefully generate a good solution to whatever problem is being examined.

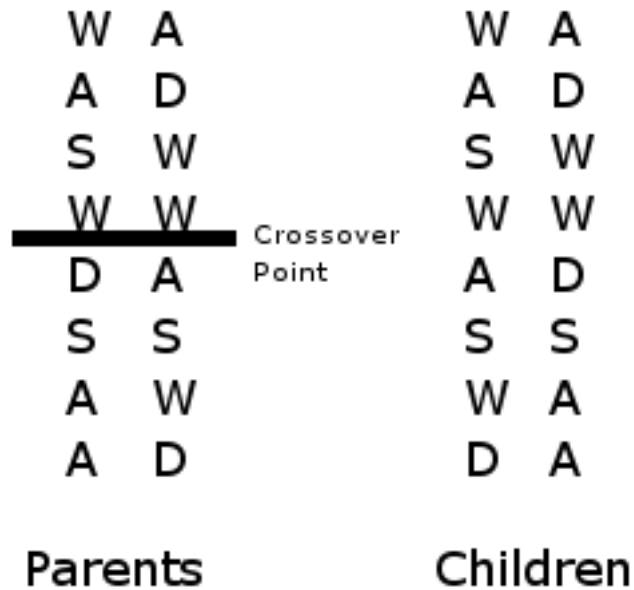
One of the most important aspects of a genetic algorithm is the fitness function, the driving force behind the guided selection of genes. Generally a fitness function's purpose is to evaluate how "good" a particular individual is as defined by the environmental stresses placed on the population[1]. What the definitions of good and bad fitness are depend on the implementation

of the algorithm and the problem to which it is being applied. The fitness function is used to guide the development of the population by determining which individuals should mate and which should die.

Mating of fit individuals in the population is done through a crossover function, another method derived from genetic principles[1]. The crossover function (demonstrated in Figure 1 on page 4) takes the genes from two parents and combines their information to produce offspring genes (commonly two)[2, 1]. A random crossover point is selected, then the offspring are created by combining the information before the point with the information after the point on the other gene. This fairly simple process follows the biological model by which DNA recombines in gametes[1]. More complex forms of this function can be used, such as selecting multiple crossover points or using uniform crossover where a probability is used for each bit to determine from which parent the information comes[1]. Selection of candidate parents is handled differently across implementations, but generally only the fittest members of the population are given the chance to mate, hopefully allowing for the propagation of fit genetic data[1].

One problem in many forms of artificial intelligence techniques and generalized searching algorithms is the convergence to a local maximum instead of a global maximum[1, 2]. To avoid this problem in genetic algorithms, usually seen as a stagnant population, another function borrowed from nature, called mutation (demonstrated in Figure 2 on page 5), is used. The mutation function randomly selects individuals from the population and changes

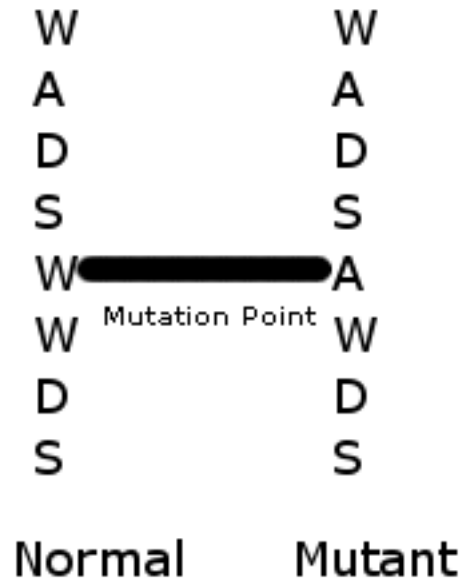
Figure 1: Simple crossover function between two parent genes.



a random bit of information in one of its genes[1, 2]. This mutation may or may not lead to an individual which is more fit than it was previously, but it ultimately helps to ensure that the population dynamically changes with new information in the form of genes[1, 2]. It is hard to predetermine the probability and type of mutation that will be most beneficial to the algorithm, so these variables are generally tested with various values until an experimental best is found[1]. Following the biological model, mutations do not occur often as they most likely will result in a less fit gene, so the probability of the function being carried out for a given gene is kept fairly low[1].

These three functions (fitness, crossover and mutation) can be used in

Figure 2: Example mutation applied to a single gene.



conjunction to form the basis of any genetic algorithm implementation of the following general form, with more specific details of implementation in Figure 3 on page 7.[1]:

1. Generation of starting population
2. Determination of fitness of each individual within population
3. Mate individuals via crossover
4. Mutate individuals within the population
5. Repeat from step 2 until termination criteria are met

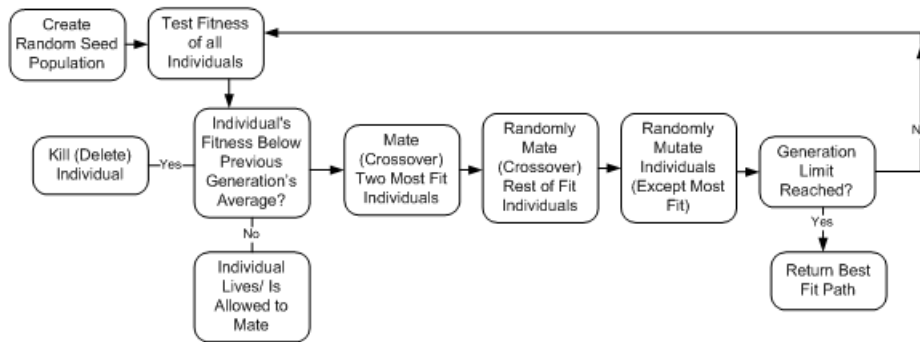
Termination of this cycle is determined on an implementation basis, but often

the finding of an individual that is “fit enough” or the iteration through a fixed number of generations is used[1]. Another issue which needs addressing as the algorithm is implemented is how to control the size of the population. Some genetic algorithms choose to replace the entire population with the offspring, while others allow individuals from the parent generation to remain[1]. One problem with total replacement is the likelihood of losing very fit genes found in one or more individuals from the parent population which may not be represented in the offspring[3].

Regardless of the implementation details, the overall goal of the genetic algorithm is to produce generally more fit populations with each generation to eventually create a most fit individual representing a solution to the problem. Genetic algorithms can work on problems where the search space is incredibly large or where the exact form of the solution is unknown[1]. The algorithm detailed in this paper exploits the ability of genetic algorithms to quickly find candidate solutions in the large search space created by all possible paths through a series of obstacles. This works especially well because the solution paths generated need *not* be optimal; a “good enough” path is all that is desired. The reasoning behind this criteria is largely due to how actual organisms perform path planning which rarely produces optimal paths. Previous work has been done which demonstrated the use of genetic algorithms on robotics platforms and software based artificial intelligence agents which showed using these imperfect paths leads to a more natural look of the movements conducted.[3, 2]. This previous research also demonstrates that

there is a significant decrease in runtime associated with utilizing genetic algorithms for path planning, so if optimal paths are not necessary, genetic algorithms are certainly a viable option for use on robotics platforms[2, 3].

Figure 3: Outline of genetic algorithm developed for this project.



## 2 Implementation

### 2.1 Genes, Populations and World Representation

The first decision made when implementing this algorithm was how the genes would be represented. A similar technique to Holland's original bit string representation is used, though instead of a string of bits, the gene is a string of absolute directions. The characters "w", "a", "s", and "d" are used to represent forward, backward, left and right respectively (as seen in Figure 1 on page 4 and Figure 2 on page 5). Instead of using multiple genes for a chromosome, only a single gene represents an individual in the population. The initial population is created by randomly assembling 100 strings of these four values.

Before the robot can carry out the movements in the solution path, the gene must be translated from absolute movements to relative ones, such that the appropriate  $90^{\circ}$  turns are made before moving forward.

The paths were chosen to be represented this way because of how the world the robot sees around it is represented in the program. The 2D world the robot sees is turned into a binary 2D array of fixed size cells where the only information for a block of real-world space is whether it is occupied or not. The genes therefore represent absolute directional movement through this 2D map. This 2D representation works well for irregular shaped objects, but its accuracy relies on its resolution and requires a relatively large amount of memory[2]. Several other common methods of modeling the environment exist, each having their own improvements and drawbacks. Representing objects as polygon approximations may have reduced accuracy but drastically decreases the amount of memory required as only the coordinates of the vertices are stored[2]. Another common technique, CSG, represents objects as unions and intersections of sets of primitive shapes, allowing for the accurate representation of curves[2]. The 2D array was chosen for this project because it is relatively easy to implement and accurate representation of the environment was experimentally achieved with a small enough resolution such that memory usage was not a problem.

## 2.2 Fitness Function

The fitness function used in this path planning algorithm relies heavily on the Manhattan distance heuristic. The Manhattan distance is the sum of the absolute values of the horizontal and vertical distances between the robot and the goal[1]. Because the original idea for the project was to model predator-prey dynamics, the fitness function is also set up to find the Manhattan distance between the robot and the predator though this functionality is currently unused.

To keep the robot from moving back and forth, especially when faced with a large obstacle, it is necessary to penalize the path for repeating states. To do this, a unique hash id is created for each state by saving the row and column of the 2D map which the robot currently occupies. For each repeated state found while the fitness is being evaluated, the path is penalized a set amount.

The genetic algorithm tries to *minimize* (higher numbers are worse) the fitness value associated with each path which is determined using the following equation (1):

$$\sum_{i=0}^{N-1} 1.5 \times f(x_i) + 2 \times \left( \frac{1}{g(x_i)} \right) + p \quad (1)$$

where  $x$  is the path being evaluated,  $N$  is the length of the path,  $f(x)$  is the Manhattan distance to the goal and  $g(x)$  is the Manhattan distance to the predator. Both of these functions are weighted by experimentally

chosen constants used to balance the priorities of the fitness function. The variable  $p$  represents any special penalizations placed against the path that each have their own set values. Penalizations occur when the path takes the robot through an obstacle or predator, repeats a state (as mentioned above), moves backwards, or when the path places the robot directly around more obstacles.

### **2.3 Crossover Function and Population Control**

After each individual in the population has been evaluated for fitness, the best solutions at this point are selected for mating via crossover. In the first randomly selected generation, only the most and the second most fit individuals are mated. After that, a threshold is set which is calculated by getting the average fitness of the previous generation. Any individual with a fitness value equal to or less than that threshold is allowed to mate with another randomly selected individual with a fitness value below the same threshold. Whether or not these individuals actually successfully mate is again chosen randomly. The most and second most fit individuals of each generation *always* produce offspring. Individuals with a fitness value above the threshold die and are taken out of the population without creating offspring. This model allows for a dynamically sized population with parent generations continuing onto the next.

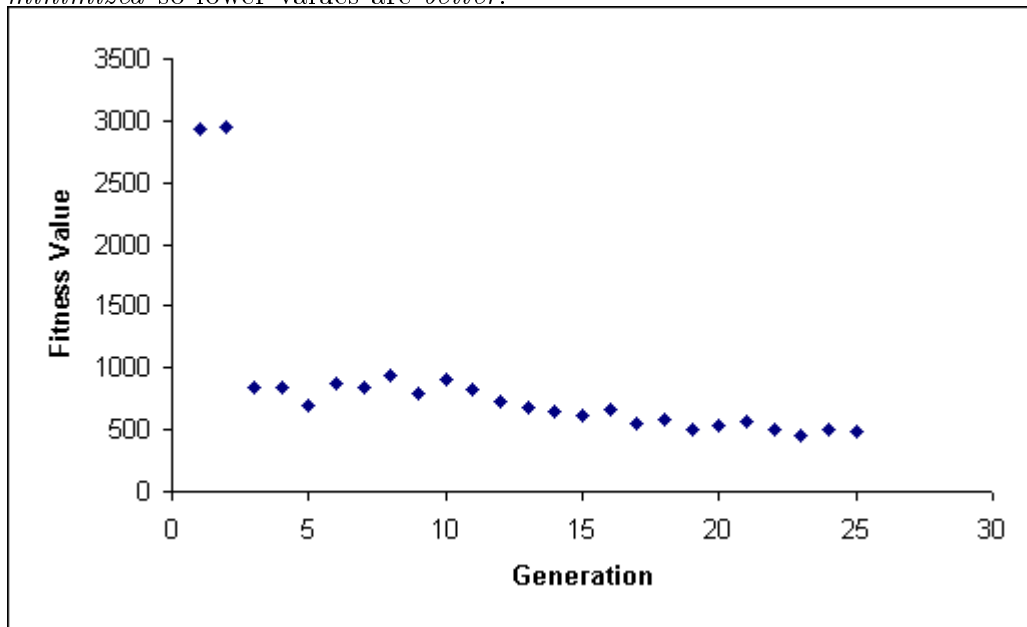
The mating of two individuals is done through a uniform crossover function. Each piece of information in the genes of the two offspring (the absolute

direction) is given a probability of whether it comes from parent 1 or 2. The genes of the offspring are then constructed and placed into the population along with their parents. Even though this method attempts to increase the average fitness between generations, the creation of offspring through this crossover function does not necessarily create genes that are even anywhere near as fit as the parent genes. Coupled with mutation of the population, the average fitness can actually be reduced between generations (see Figure 4 on page 12). The hope is that even though unfit genes may enter the population, at least one will be created that is more fit than the previous best. Larger population sizes and increased chances of successful mating increase the likelihood of a better fit offspring being created, though these factors increase the overall runtime of the algorithm. A starting population of 100 individuals with a 1 in 15 chance of successfully mating between any two individuals above the threshold over 25 generations has been experimentally shown to produce nearly optimal paths which are good enough for navigation (see Figure 4 on page 12).

## **2.4 Mutation Function**

The mutation function used in this genetic algorithm works as outlined in Figure 2 on page 5. An unbounded random amount of genes are randomly selected from the population and mutated at a random spot within the gene. The mutation may or may not create a gene which is more fit than it was previously, but the mutation helps the population to remain dynamic and

Figure 4: Average fitness values of 25 generations from six iterations of the genetic algorithm on the same starting state. Note: fitness values are being *minimized* so lower values are *better*.



not stagnate around a local maximum. The most and second most fit path in the current generation are always kept from mutation to keep that gene from being lost. Genes currently have a 1 in 6 chance of being selected for mutation. The probability of mutation is reasonably high for this algorithm because the crossover function does not often result in a child which is more fit than the current best fit individual, so the population would tend to favor a local maximum. Having a relatively high chance of mutation keeps the population dynamic and helps to prevent it from stagnating. Population and generation size are fairly small for this algorithm to reduce the runtime required to generate a path to make the decision process as quick and natural as

possible. These factors make it necessary to ensure rapid, positive changes in the population represented by a limited amount of data, so constant injection of new paths through mutation is required.

## 3 Discussion

### 3.1 Integration With Robot

The robot used in this project is an OPEN-ROBOT, a robotics kit from Abe Howell's Robotics([http://www.abotics.com/open\\_robot.htm](http://www.abotics.com/open_robot.htm)). The robot utilizes two forward looking infrared sensors to gather information about the world around it which it then sends wirelessly using an XBee wireless module to a host computer. When the map of the world needs to be updated, the program running on the computer issues a command to the robot, causing it to rotate 360<sup>o</sup> and collect data from the infrared sensors. The computer uses the data received from the robot to generate a 2D array of binary values representing the space around the robot. Since the array contains only binary values, a relatively crude 2D map of open and closed space is generated. The algorithm only considers the area around the robot which it must immediately path find through, so the map is then reduced in resolution and the distance it represents. This 2D array is then used by the genetic algorithm to construct a path four absolute moves in length, representing the genes in the population. These absolute moves are turned into relative moves, as mentioned previously, and then issued as commands to the robot. Once these

moves are complete, the robot performs another 360<sup>o</sup> turn, collecting infrared data and generating a new map which is used by the genetic algorithm to generate the next path. Currently the robot does not actually find a real world goal, instead a goal is placed into the 2D map by the program. This goal is placed in front of the robot when it first starts pathfinding, being updated each time the robot generates a new map. Using this method, the robot's ability to pathfind can be tested by placing the robot in front of a maze of obstacles and allowing it to find its way through.

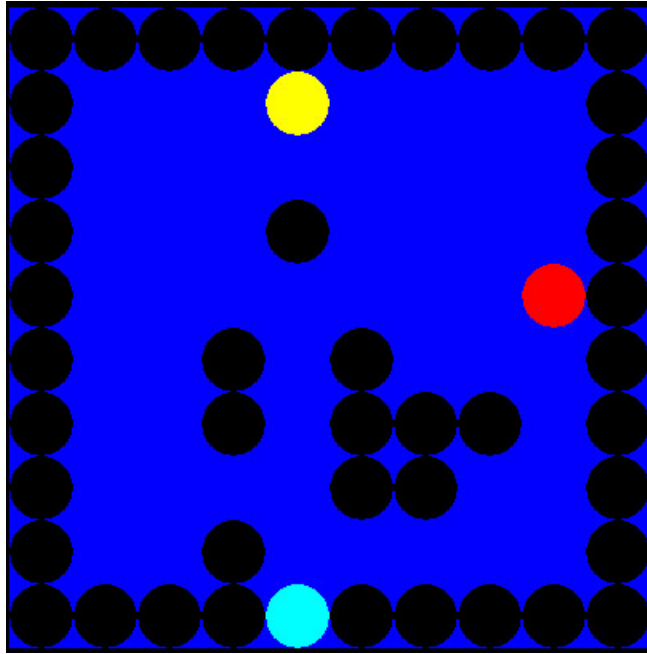
The robot can accurately map large objects with flat surfaces, allowing the path planning algorithm to avoid the obstacles. Increasing the amount of objects, decreasing their size and utilizing rounded or oddly shaped objects all contribute to a decline in the paths generated and how well the robot avoids the obstacles. Currently only one robot exists, so experimentation with the originally planned predator-prey dynamics has not been done but is a major part of future work on this project. At this time the robot mostly models the prey bot with its goal finding priorities, while predator avoidance is already set up allowing for rapid testing once one is developed.

## **3.2 Development**

When this project was started, the genetic algorithm for pathfinding was intended to be run on the robot itself. The first generation of the algorithm was written in C to maximize runtime and memory efficiency. Before integration with the robot, a program was written using OpenGL to visualize

the movements generated by the pathfinding algorithm for testing purposes (Figure 5 on page 15).

Figure 5: OpenGL visualization of original algorithm written in C. Dark blue areas are open, black dots are obstacles, red is the predator, yellow is the goal and light blue is the prey.



The first robot used in this project was built from scratch with relatively cheap parts to keep overall costs down. At this point the envisioned outcome of the project was still to have multiple robots operating, demonstrating predator-prey dynamics. Eventually it was realized that the parts being used in the robot would not be able to efficiently run the pathfinding algorithm, so the decision was made to run the program remotely and use XBee wireless modules for communication.

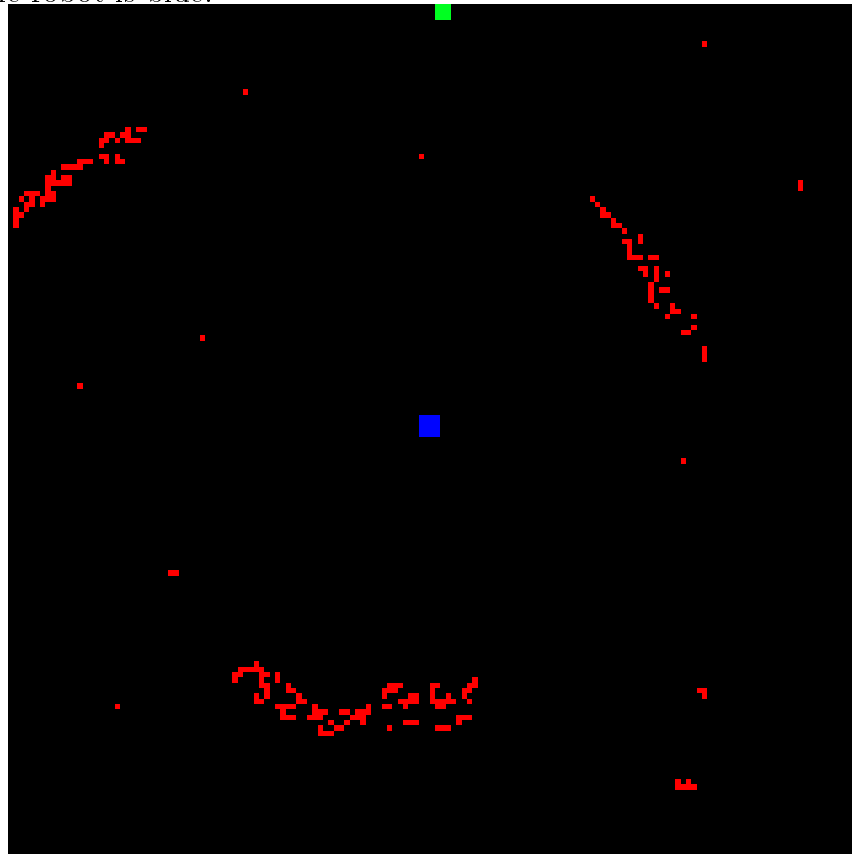
Moving the program to a new, more powerful platform of a desktop or lap-

top computer allowed for greater flexibility when implementing the pathfinding algorithm. Creating the graphical front-end and wireless communication code was done in Python since fast, memory efficient programming was no longer absolutely necessary. To avoid having to port the pathfinding code and in order to keep it as efficient as possible, SWIG was used to generate wrapper code in C, allowing the original algorithm to be compiled as a dynamic-link library which could be called from the Python program.

It started to become apparent that building a robot from the ground up was not feasible with the time remaining in the project. The OPEN-ROBOT kit provided for a relatively cheap platform which could perform all of the hardware operations required. The robot came with a library of functions written in C# for performing the various operations it was capable of. The Python front-end code was easily ported to IronPython (a Python distribution integrating the .NET framework) allowing the robot function library written in C# to be compiled and imported as a dynamic-linked library similar to how the pathfinding algorithm was used. Though this was supposed to avoid the time and problems associated with porting code, using three different languages for one program along with several compatibility issues resulted in an unnecessary burden. The original pathfinding algorithm and front-end Python code was eventually ported to and later completely rewritten in C#, the final form of the code for the algorithm presented in this paper. The code that runs on the remote computer has a graphical user interface allowing the robot operator to perform simple tasks such as

starting and stopping path planning and environment traversal. Functions to help with testing and debugging also exist, such as forcing the robot to perform  $360^\circ$  turns to map its environment. The most important and useful feature of the interface is the canvas displaying a visualization of what the robot has mapped from the environment (see Figure 6 on page 17).

Figure 6: Visualization of the environment as mapped by the robot. Red areas indicate objects while black areas are open space. The goal is green and the robot is blue.



### 3.3 Future Work

The original goal of this project was to develop genetic algorithms used for pathfinding by robots acting as either a predator or prey. The prey would seek out one or more goals while trying to avoid the predators, while the predators would try to track down the prey. All of the movements performed by the robots would be based on the paths planned by the genetic algorithm. Unfortunately time and money constraints kept this goal from being met. Currently, there is only one robot but it successfully visualizes, maps, and finds its way through an environment of obstacles. The project is at a stage where the rest of the features can be implemented without too much difficulty. Previous research on utilizing genetic algorithms to generate movements for robots and software based artificial intelligence agents exist, but there is little in regards to using this method to model predator and prey dynamics[2, 3].

Further work on this project will hopefully bring the generation of better paths and more precise mapping of the physical world to the 2D binary array representation the algorithm uses. Along with these more fundamental improvements, the project will hopefully be extended to accomplish the original goal of modeling predators and prey. Prey bots will have multiple physical goals which they can seek, choosing one based on the actions of one or more predator bots. Efforts may also be made to improve the robot's vision by way of how it actually sees, using methods other than infrared range finding. These new vision techniques will most likely change how the data is repre-

sented on which the genetic algorithm works. Changes in the genotype of the paths and the mechanics of the genetic algorithms may also be pursued in efforts to make the algorithm quicker while producing more complex yet optimal paths. Having the path planning algorithm consider moves *beyond* the immediate set of moves being planned will also help make the overall path through the obstacles more efficient. Currently paths are limited to four absolute movements at a time to reduce the effects of instances of poor accuracy in the 2D environment representation. With the generation of better paths, the algorithm will likely be able to plan much longer paths that remain accurate and optimal as they are traversed.

## 4 Acknowledgments

I would like to thank my fellow project member Steven Turner, my project mentor Dr. Alan Jamieson and Dr. Simon Read for their work and guidance. I would also like to thank Abe Howell for his OPEN-ROBOT robotics platform and his prompt and thorough responses to all of our questions.

## References

- [1] Ben Coppin. *Artificial Intelligence Illuminated*. Jones & Bartlett Publishers, 1 edition, March 2004.

- [2] Mark A. C. Gill and Albert Y. Zomaya. *Obstacle Avoidance in Multi-Robot Systems: Experiments in Parallel Genetic Algorithms*. World Scientific Publishing Company, August 1998.
- [3] R. Leigh, S.J. Louis, and C. Miles. Using a genetic algorithm to explore a\*-like pathfinding algorithms. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 72–79, 2007.