An Area Exploration Strategy Evolved by Genetic Algorithm

by

Yuchao Zhou

(Under the direction of Walter D. Potter)

Abstract

Autonomous robot design is a challenging field in artificial intelligence. It is not easy to design autonomous robots to perform useful tasks in real environments. There are two reasons why designing a control system for an autonomous robot is a difficult task. The first reason is that it is difficult to coordinate different parts of the robot to work together and to operate as the control system requested. The second reason is that autonomous robots interact with an environment, which means they may face unpredictable environment situations so that their control system may be misled. A novel approach using Genetic Algorithm simulation is discussed in this thesis and the results of the simulation are verified physically by real robot implementation.

Index words: Robotics, Genetic Algorithm, Evolvable Hardware, Navigation, Simulation, Rule-based control

An Area Exploration Strategy Evolved by Genetic Algorithm

by

Yuchao Zhou

B.S., Zhongshan University, 1999

M.S., The University of York, 2002

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2005

An Area Exploration Strategy Evolved by Genetic Algorithm

by

Yuchao Zhou

Acknowledgments

Firstly, I would like to thank Dr. Don Potter for giving me an opportunity to work on the robotic project. Since the first day I met him in AI center, he has provided me with everything that I needed to complete my thesis work and my academic career including the support for robot hardware and invaluable academic advice. I also thank Dr. Khaled Rasheed and Dr. Jeffrey W. Smith for being on my thesis committee and for spending their precious time on my thesis work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Autonomous robot design is a challenging field in artificial intelligence. It is not easy to design autonomous robots to perform useful tasks in real environments. There are two reasons why designing a control system for an autonomous robot is a difficult task. The first reason is that it is difficult to coordinate different parts of the robot to work together and to operate as the control system requested. The second reason is that autonomous robots interact with an environment, which means they may face unpredictable environment situations so that their control system may be misled. The control system must send out its commands in real-time, and each command affects not only how well the current goal is completed but also the situation of the robot in the environment. For these reasons, a robot that works well in simulation may not behave well in a real environment.

To solve these problems, we propose the use of evolutionary algorithms in robotics. These algorithms have been used previously to generate rules for rule-based control systems and other related areas where they have shown the ability to produce alternative and better solutions to those created through other means. Evolutionary algorithms are highly adaptable and have been used in research and industry as powerful tools of optimization and invention. Furthermore, a system enhanced by evolutionary algorithms is able to improve itself while it is running.

1.1 Background

Implementing evolutionary algorithms on a robot is not an easy job. The methodology used to evolve a control system for autonomous robots is not well established.[NFMM94] The large population size and the number of generations required in evolutionary algorithms implies that a large number of individuals will be evaluated and long experiment time will be expected to achieve acceptable results. This fact restricts many researchers to try to use simulations before implementation with physical robots. Some researchers have used real robots when they develop autonomous evolutionary robots. Here are some examples:

1. Floreano and Mondada [DM93] developed a neural controller agent which is able to perform a navigation task by an evolutionary approach. The robot used was Khepera. The goal of the project was to develop a robot that could avoid obstacles while it kept moving straight and fast. Good results were generated after 60 hours and 100 generations of evolution.

2. Miglino, Nafasi and Taylor[ONT94] evolved a recurrent neural network to control a Lego mobile robot to explore an open area using a rough simulator or the robot/environment interaction. Their goal was to explore the largest possible area. The experiment runs in simulation first, then the controller is implemented with a real robot. The simulation lasted 600 generations running for three hours on a Sun workstation.

3. Hoffmann [Hof94] evolved a hybrid system which contains neural network and fuzzy logic controllers to control an autonomous rule-based control robot to perform wall following.

The goals of these examples are some basic robotic behaviors such as collision avoidance and wall following. These behaviors are the simplest building blocks of robot behaviors. How to combine these building blocks and implement more complex behaviors becomes an important issue in advance research. These results show that evolutionary algorithms potentially solve complex problems in robotics.

1.2 Introduction to evolutionary algorithms

Biological Evolution is an adaptation system. The environments on earth are constantly changing. Sometimes this change is gentle, while other times the change may be extreme. Entities on earth, such as animals and plants, can adapt to the changing environment by evolution [Dar59]. In this process, some entities become extinct because they could not adapt to the environment. The survivors picked by natural selection have a chance to pass genes to their descendants. These evolutionary cycles have taken place over billions of years. As time passes, this adaption system becomes more delicate and more efficient. Today, the Earth is covered by billions of creatures, from single cell bacteria to human beings, all adapting to the environments in which they live.

Biological Evolution inspired researchers to design adaptive systems via artificial evolution. Computer-simulated evolution is called Evolutionary Computation [Whi93]. The Evolutionary Algorithms performing include Genetic Algorithms (GA) [Hol92], Evolution Strategies (ES)[BS02] and Genetic Programming (GP)[Koz92]. All have a number of similarities but they also have different chromosome design and operators. These related algorithms provide powerful tools to solve the difficult problems in a number of research fields. A principle of employment of evolutionary algorithms is to adapt and optimize the behavior of a system to perform special tasks, such as pattern recognition and analogue design.

1.3 Introduction to rule-based robotic control

Mobile robot control architectures have two main categories: hierarchical and rule-based (behavior-based). A rule is a behavior in the interaction dynamics between an agent and its environment [XB97]. Rule-based robot control uses the blackboard model to organize the relationship between the conditions from the sensors and the actions output to the actuators. The controller receives information from sensors in real-time then updates the blackboard. It evaluates the conditions of the rules with the information stored in the blackboard. If there is a match, the controller can execute the commands of the rules. The advantage of the rule-based architecture is the short response time to environment change. A rule-based controller architecture is used in this project.

1.4 Project goal

The attempt to combine robotics and evolutionary algorithms is still in the early stage of research. Evolutionary algorithms search for solutions within a large search space with high flexibility while the function of the robot hardware available for experiments highly constrains the solutions that can be implemented on it. This fact determines that the project goal should be based on the abilities of the hardware; otherwise it will be an impossible mission. An autonomous robot equipped with the latest technology usually carries sensors with different abilities to receive information. It has arms and wheels to move and interact with the environment. The sensors will provide the robot with complex and dynamic data, and the robot needs to make decisions based on the data and perform a few simple actions. The rule-based control model provides a direct link from complex environment data to a few actions. The functions to translate environment into actions, the rules, are the critical part of the system design. Because of the large amount and complexity of environment data, the searching space of the suitable rule set is a huge.

The goal of this project is to use a genetic algorithm to generate a set of rules to explore unknown space. Robot ER-1 is selected as the robot platform for the experiments because of it has the functionality we need, and it is easy to configure. An IBM T30 laptop is used to connect to the ER-1 framework and acts as the controller of ER-1.

Chapter 2

Evolutionary Algorithms

2.1 Introduction

Biological Evolution has inspired researchers to design adaptive systems via artificial evolution. This computer-based simulation of evolution is called Evolutionary Computation [Whi93] and the algorithms are called Evolutionary Algorithms. Evolutionary algorithms provide powerful tools to solve difficult problems in a number of research fields. They consist of several key components: the encoding of the artificial genotype, the selection, the genetic operators and the fitness function.

2.2 Genotype

In a natural system, living creatures inherit physical characteristics from their ancestors passed on through DNA sequences. The Genotype is the DNA that describes the genetic constitution of an individual organism. This base-4 biological encoded genotype contains the information that can be used to create proteins and enzymes required to construct a creature.

In Evolutionary Algorithms, the genotype is often encoded in a binary string or in a tree-based structure. The binary string genotype is simple in structure, but it may have a complex mapping to the phenotype in an actual implementation. The tree-based genotype is harder to implement, and it is generally suitable for fields such as Genetic Programming [Koz92]. The genotype encoding is a problem dependent task. The solution of the problem is encoded in the genotype, and it is implemented by mapping genotype to phenotype. The simplest mapping is a 1:1 mapping of Genotype $\longrightarrow$ Phenotype [KKI+98], however as systems grow this does not scale well. A good genotype design ought to be able to cover all the possibilities of the resources of the system and represent any minimal changes of the system configuration.

2.3 Criteria for Evolution

After a genotype is mapped to a phenotype, the individual will be evaluated in simulation or on the final implementation medium. The evaluation function is called the fitness function. The fitness function is designed based on the criteria of evolution. It can generate a value called the fitness value or fitness. Different fitness functions are used in different experiments depending on the application, and the fitness function is the critical part of these experiments.

The fitness represents how good an individual is at meeting the requirements of the system, e.g. how good a full adder is if a system is designed to evolve a full adder. Generally, a higher fitness means a better solution has been found. A fitness definition ought to be designed to classify the experiment from the worst to the best results with proper values.

To stop the evolution, there will be a value set by the designer to indicate the goal for the mission of the experiment. If the fitness is greater than this value, the evolution will be stopped and the performance of the best individual will reach the goal. This value is the maximal fitness, in some case, minimal, which the implementation may achieve finally.

2.4 Selection

After all the individuals are evaluated, the algorithm begins selecting individuals for reproduction. Generally high fitness individuals will have a higher chance of being selected, although it is desirable that even low fitness individuals have some chance of being selected in order to maintain genetic diversity. A number of selection algorithms will now be discussed.

2.4.1 Roulette Wheel Selection

The original selection method from Holland [Hol92] is roulette wheel selection. The roulette wheel is divided by sections like a pie. The angle each individual occupies is proportional to its fitness, as shown in Equation 2.1:

$$Angle[i] = 2\pi \frac{fitness[i]}{\sum_{i=0}^{n-1} Fitness[i]} \tag{2.1}$$

The wheel is spun and stops randomly at a position between $0 - 2\pi$. The resulting individual is selected. This selection continues until enough individuals are ready for the new generation. In this scheme individuals with high fitness will have a higher chance of being chosen relative to those with lower fitness.

Roulette wheel selection is a simple selection method. In the early stage of evolution, most individuals have low fitness with only a few having relatively high fitness. These high fitness individuals will occupy a large area on the wheel, which means they are chosen almost every time. This fact makes the algorithm converge to these individuals. However, at this moment the fitness of these individuals is still low compared to the maximum fitness. Because these individuals are over represented, the diversity of the population decreases rapidly, and the average fitness is hard to improve. This problem can be solved if we adjust the relation between selection probability and the individual fitness. For example, the angle that one individual occupies on the wheel should not be a constant, but should be modified after each selection. Once an individual is selected, its probability to be selected again in the next selection can be modified to a lower value so that the probabilities of other individuals to be selected increase.

2.4.2 Tournament Selection

Another method also widely used in evolution algorithms is tournament selection. This selection makes the fitness comparison between a small number of individuals instead of the whole population. The algorithm is as follows:

1. Select randomly two individuals a and b from the population.

2. Generate a random number r between 0 and 1.

3. If $r > k$ (k is the parameter selected by the user), then select the higher fitness individual from a and b, otherwise select the low fitness individual.

This method is still a fitness related selection, however the chance of the low fitness individual being selected is higher than with roulette wheel selection.

2.4.3 Ranking selection

The selection methods mentioned above are directly related to the value of the fitness. Another selection method called ranking (or rank-based) selection usually consists of making a list of all the individuals in the population, ordered by relative fitness, and then using the first 'n' individuals to create the next population. This method has the advantage over roulette wheel selection in which individuals with high relative fitness are not over-represented in the next population, and thus helps to preserve population diversity. Therefore ranking selection is a good choice. [HB91] [Whi89].

2.4.4 Elitism

Elitism is not an independent selection strategy but it works with other selection methods. It clones the best individual in the current generation into the next generation. It prevents the genetic operators (Section 2.5.1) from modifying the sequence in order to keep the best individuals and stabilize the performance of the whole population.

2.5 Reproduction and Recombination

Once the individuals are selected for reproduction, genetic operators such as crossover and mutation are carried out. New children will continue to be generated until their number reaches the population size.

2.5.1 Genetic Operators

Genetic Operators are used to manipulate the chromosome sequence in evolutionary algorithms. They simulate the corresponding mechanisms in biological evolution, such as crossover and mutation.

2.5.2 Cloning

Cloning is an operator that duplicates the old chromosome sequence to a new sequence without any modification. It can be used to perform elitism by duplicating the selected individuals of the current generation into the next generation.

2.5.3 Crossover

Crossover is an operator that constructs a new chromosome. It selects two parts of the chromosomes from two different individuals and combines them into one. This new individual receives the genes from its parents and inherits the benefits from more than one individual of the last generation.

Several kinds of crossovers are used in evolutionary computation. The simplest is one-point crossover. In one-point crossover, the chromosomes of two parental individuals are cut at some randomly chosen common point, and the resulting sub-chromosomes are swapped. Two new chromosomes will be generated: one combines the "head" of parent A and the "tail" of parent B; another one combines the "head" of parent B and the "tail" of parent A, as illustrated in Figure 2.1, two new individuals A' and B' are created.

Two-point Crossover and Multi-point Crossover have two or more crossover points simultaneously in the operation. New individuals are combined with different parts from their parents. The extreme case is called uniform crossover in which the chromosomes from different parents are divided into the smallest building blocks, the basic information unit (gene), then these blocks are mixed to construct new chromosomes. The choice of crossover and its probabilities are dependent on a particular application and representation.

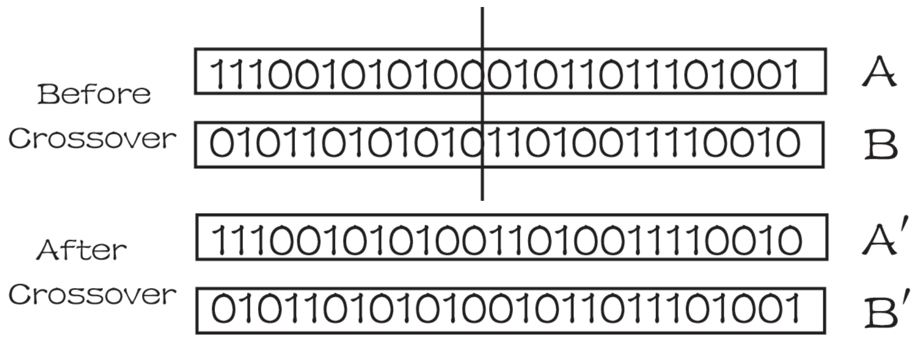

Figure 2.1: One Point Crossover

2.5.4 Mutation

Mutation is an operator that constructs a new chromosome by altering the value of its smallest building blocks. In a bit-independent binary string, mutation inverts one or more of the bits in the sequence. The mutation operator slightly modifies the chromosomes and the new individuals are generated with some information bits altered. In recent years, mutation has been regarded as more important than crossover in evolutionary algorithms [Hol01]. Evolution Strategies(ES) use mutation as the main operator to manipulate chromosomes.

Mutation rate is an important parameter for the mutation operator. A higher mutation rate will make the fitness value change rapidly, and the individuals with lower mutation rate will make steady and slow progress. A special kind of mutation named hypermutation operates as follows: a low mutation rate is applied to high fitness individuals and a high mutation rate is applied to low fitness individuals. This idea is similar to a process carried out by the biological immune system, and it may be used to improve the performance of ES [TKZ04] [dCT02].

2.5.5 Insertion

As the algorithm reproduction cycle repeats, the fitness will tend to be high and the individual chromosomes tend to converge to some similar sequences. The diversity of the genotype will decrease and the improvement of the evolution becomes less significant. A novel and creative individual will rarely appear.

The operation of insertion introduces new randomly generated individuals to the population. Although this operation may not assist in improving the average fitness of the population, it increases the diversity and helps prevent the individuals converging into a small range [TKZ04].

2.6 Genetic Algorithms

The Genetic Algorithm (GA) was first described by John Holland in his book "Adaptation in Natural and Artificial Systems" [Hol92] in 1975.

The algorithm pseudo-code is as follows:

```
1. Initialize()
2. while(criteria =FALSE)
3.     phenotype=mapping(genotype)
4.     fitness=evaluate(phenotype)
5.     while(population size not full)
6.         parent=Select(individual)
7.         new individual=mutation(crossover(parent))
8.     Generation +1
```

2.7 Branches of Evolutionary Algorithms

2.7.1 Evolution Strategy

The Evolution Strategy(ES) [HP87] was developed by students at the Technical University of Berlin in the early 1960s [I71] and [HP75]. The usual goal of an evolution strategy is to optimize a given objective or quality function [BS02].

The simplest form of ES is called two-membered or (1+1)-ES [BS02], which is just one old individual and one new individual in each generation. A more complicated form of ES is (N+1)-ES. Here N is the number of parents and the offspring number is one. Two further versions of multi-membered ES were introduced by Schwefel [BS02]. One of them is $(N+\lambda)$-ES, in which not only one new individual is created in one generation, but $\lambda \geq 1$ offspring are created. The worst λ individuals out of $N+\lambda$ will be discarded. Another is (N,λ)-ES, in which the selection for the next generation will take place among the λ offsprings. Their parents will be discarded no matter how good or bad their fitnesses are.

The $(N+\lambda)$-ES algorithm pseudo-code is as follows:

```
1. Initialize()
2. while(criteria =FALSE)
3.         for new individual 1 to lambda
4.                 parent=Select(individual)
5.                 new individual=mutation(parent)
6.         phenotype=mapping(genotype)
7.         fitness=evaluate(phenotype)
8.         discard lambda individuals
8.         Generation +1
```

2.8 Algorithm used in project

A genetic algorithm is used in this project. It is implemented in a simulation environment to generate rules that will be applied to a robot control example. The implementation is discussed, later in this thesis. The next chapter describes the hardware resources used in this work.

Chapter 3

Robot platform

3.1 ER-1

The robot in this project is model ER-1 from Evolution Robotics. The ER-1 is an easy to use robot platform with many functionalities. It has a reconfigurable chassis which is easy to assemble. It also has the following features that can be utilized for different research purposes [Inc04a]:

- Vision - able to capture, recognize and identify thousands of objects and locations
- Hearing - contains 'listen for' speech recognition; can also respond to sound levels
- Speech - able to talk using a 'phrase to speak' function
- Networking - can send and receive e-mail; able to e-mail commands when used with a wireless card and network
- Remote control - can be tele-operated from an external computer
- Autonomous mobility - able to specify movement parameters such as direction and target characteristics, allowing the robot to move around by itself
- Gripping - optional arm-like gripper grabs and carries objects
- IR sensing - optional IR Sensor Pack provides object presence detection, allowing you to trigger a rule if an object is detected

Figure 3.1 is the front view of the ER-1. It is 24" x 16" x 15" (H x W x D) and 20 lbs without laptop.

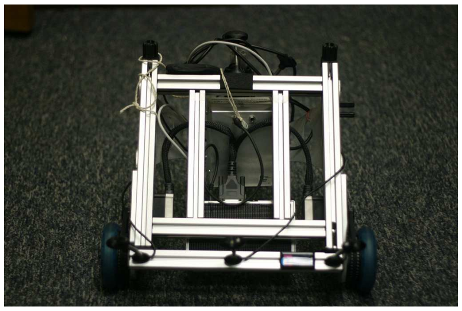

Figure 3.1: ER-1 without computer

3.1.1 Robot Control Module and Power Module

Robot Control Module is an electronics box for interfacing motors and I/O to a laptop. It contains: a USB connection to the laptop, a DC power input port, two ports for the Stepper Motors, a 25 Pin D Sub connector for digital I/O, providing 8 digital outputs and 8 digital inputs, and a 25 Pin D Sub connector provides 15 analog inputs

The Power Module has a 12 V. 5.4 A-hr battery with internal fuse for safety, and there are two 12 V unregulated DC output ports and one DC charger input port on the Power Module.

3.1.2 Sensors

There are several sensors on the ER-1. With infrared sensors, the ER-1 can navigate by detecting and avoiding an object or a person in its way. It can detect objects and people up to 30” away in a 20” wide field of view and the gooseneck tubing allows sensors to be pointed in any direction.

3.1.3 Camera and drive

The ER-1 can connect, at most, to two cameras through USB ports. One camera from Creative with 640*480 resolution provides the visual recognition ability and the other provides the collision avoidance ability. The robot is driven by two pre-assembled drive systems with two stepper motors and 4-inch diameter scooter wheels. The moving speeds are between 5-50 cm/second and the turning speed is 5-90 degrees/second [Inc04a].

3.2 Programming interface

There are three methods to connect and control the ER-1. One is called the Robot Control Center (RCC), a Graphical User Interface (GUI) for controlling and customizing the ER-1. Another is the Socket API/Command Line Interface that lets users directly access the robot's functionality by sending commands through the TCP/IP network. The last method is the ERSP(Evolution Robotics Software Platform) [Inc04b].

3.2.1 Robot Control Center

The Robot Control Center is a program with GUI which allows the ER-1 to recognize objects, colors, sound levels and words, to send and receive email, to act on schedule, to move around autonomously or by remote control, to play sounds and music or to take pictures and video. RCC software can be used to organize these function into rules with an easy-to-use GUI.

3.2.2 Socket API/Command Line Interface

The Socket API/Command Line Interface allows users to create simple programs that send commands directly to the robot. Also the API can be used to gather sensory data from the robot.

The Command Line API begins to listen to a specific TCP port as soon as the RCC starts to run on the laptop attached to the ER-1 robot. Any programming language such as Python, C, C++ and Java which can handle TCP/IP can be used to send commands and receive feedback from the sensors. The Command Line API also supports access to the 15 analog IO ports and 16 digital IO ports of the ER-1 robot control module. This functionality provides flexibility for users to add custom hardware to ER-1. [Inc04a]

3.2.3 ERSP

ERSP (Evolution Robotics Software Platform) [Inc04b] is a software development kit for programming robots. The ERSP can be programmed with C++ and Python.

At a lowest level, ERSP provides a number of core libraries that let users easily access the robot hardware. At an intermediate level, ERSP defines a set of behavior libraries which define some simple reactions of the robot. At a top level, ERSP provides a number of tasks through which the users can command the ER-1 to complete its mission easily.

The ERSP libraries consist of a large set of functions that are useful for a wide variety of robotic applications. The infrastructure can be partitioned into four major components:

- Software Control Architecture
- Computer Vision
- Robot Navigation
- Human-Robot Interaction (HRI)

With the assistance of these libraries, users can maximally access the ER-1 hardware and utilize the functions embedded in ERSP. There are some limitations about ERSP: first, the laptop with ERSP is carried by ER-1 while running, this fact makes the debugging of the program very difficult because it is hard to see the screen while the robot is running. Second, it is not as user friendly as many GUI programs.

3.3 Rule-based control

Mobile robot control architectures have two main categories, hierarchical and rule-based (behavior-based). A rule is the interaction relation between an agent and its environment. [XB97]. Here is a rule example: if the front sensor detects an obstacle, make a 90 degree left turn. Complex rules can often be decomposed into hierarchies of simple rules. In order to organize the relationship between the information received from sensors and the actions being sent to actuators, the blackboard model is introduced into rule-based robots. The blackboard receives updates from the sensors in realtime. All sensor data are listed on the blackboard, when new data comes in, the old data will be erased and replaced by the new data, which is similar to what we do on normal blackboards. The data provided by the blackboard will be evaluated with the conditions in the rule set. The rule is an “if-then” statement. The first part of the rule indicates the conditions associated with the data on blackbroad, and the second part of the statement is the action commands. If the data match the rule conditions, the controller will send out commands to the actuators to perform the designed task.

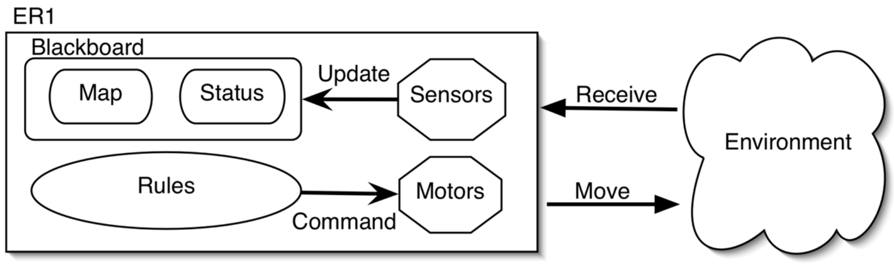

Figure 3.2: Rule-based control model

The advantage of the rule-based architecture is the short response time to environmental changes. All rules are designed before actual running, therefore generating the robot’s actions is straightforward. When the blackboard receives information from sensors, it compares them with the stored information and makes decisions without much calculations. The blackboard enables the communication of different knowledge sources asynchronously and provides a link between rules and sensors/actuators. For example, the blackboard receives data from source A at time t_1, and receives data from source B at time t_2. When the controller checks the rules at time t_3, the data received asynchronously from A and B match the rules and the controller fires the action. Figure 3.2 shows the Rule-based control model used in this thesis.

Chapter 4

Software design

4.1 Introduction

The software approach for the project contains two major parts. The first part is to evolve control rules. The second part is to implement these rules on the ER-1. In the first part, the rules are evolved by genetic algorithm and evaluated in the simulation environment. They are the phenotype of the genomes in the genetic algorithm. In the second part, the rules from the first part are used to control the real ER-1 robot to test their functions. The rules connect the first part and the second part of the approach.

4.2 Simulation

4.2.1 Genotype encoding

The genotype of the genetic algorithm is an important part of the design. A binary number with a fixed length is used to represent the rules of the rule-based controller.

The major sensors used on the ER-1 are infrared proximity sensors. In most applications, a threshold value would be set to convert the infrared input value into a binary number. For example, the infrared input value is from 0 to 100 and the threshold value is 50, if the input value is higher than 50, then it is a binary one otherwise it is a zero. Therefore, a one-bit binary number can represent one infrared input source, if there are three infrared sensors, a three-bit binary number can represent them. With this three-bit value, the robot can perform some simple behaviors like collision avoidance.

Another heavily used sensor in the ER-1 is the position sensor. It can measure the relative position from the starting point. A coordinate system is built based on the initial starting point and the initial heading of the robot. It provides the system with the verification value to evaluate the result of each movement.

A video camera mounted on the robot provides the recognition software in ER-1 with video input. The camera takes a picture of an object first and uses it to train the recognition engine. It will analyze the patterns of the picture, then store these patterns for future recognition. This video camera provides the ER-1 with certain navigation abilities such as recognizing the way-point on its way to confirm the location of the robot. The camera has certain limitations. First, it has a narrow angle of view. The robot needs to rotate and take many pictures before it obtains a full view of its surrounding environment. Second, the camera needs to capture enough detail in order to recognize its target successfully. Because of this condition, the robot can not easily recognize more than one object at the same time. Excluding the camera used for recognition purposes, the ER-1 can add a second camera and use it for collision avoidance.

Based on the goal of the project and the equipment of ER-1 mentioned above, the encoding of the rules should not simply use the binary values from the sensors. A map-based encoding method is used to encode the chromosomes of the algorithm in this project. This map is called the simulation map in this thesis. Similar to a map with grids on it, a block in the middle of the map will have eight neighbor blocks surrounding it, shown in Figure 4.1. While the robot moves on the map, it will pass these blocks one by one. A block is marked as “visited” if the robot has been to it and “unvisited” if it has never been there, this map is updated while the robot moves from block to block. As shown in Figure 4.2, each block will use a binary number “1” to represent “visit” and “0” to represent “unvisited”. There are eight blocks surrounding the robot, therefore it is an eight-bit binary number. The combinations of this binary number is $2^8 = 256$. Before the robot starts its move to the next block, it gets this eight-bit binary number, which represents the visit status of the adjacent blocks. In Figure 4.1 this number starting from the upper left corner clockwise is 00110100. The robot may consider its next move based on this environment value. There are eight possible movement options the robot can select for the next step (east, southeast, south, southwest, west, northwest, north and northeast). These eight directions can be encoded into a three-bit binary value. If the eight-bit environment value is used as the index for the movement actions and each storage unit is a three-bit number, it will generate a binary number $256 * 3 = 768$ bits long.

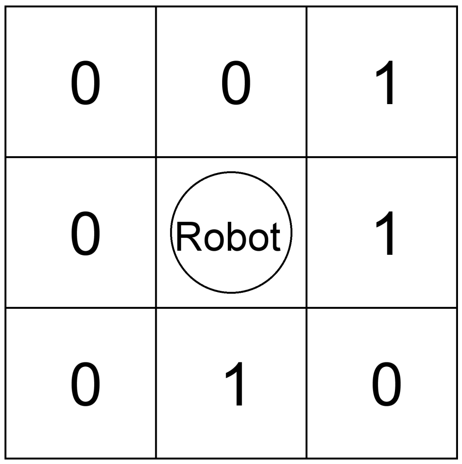

Figure 4.1: Environment value

Figure 4.3 shows a section of an entire encoded rule. It is a look-up table with eight-bit index and three-bit contents. The eight-bit number is the index(address) value from eight adjacent blocks through which we can access the three-bit encoded movement command. This binary string will be the chromosome used as the genotype of the rules. After a robot reads the eight-bit environment data from the map, it uses this data as an index to locate the three-bit movement command from the binary string, decodes it and moves to the position assigned by the rule. For example, the environment value from Figure 4.1 is 00110100. Using this number as an index to read the chromosome sequence in Figure 4.3, we get a number 101. This three-bit value represents “west” in the program hash table. Therefore, the controller will command the robot to move one block west in the map. In real testing, the robot will move 30 cm west at the same time.

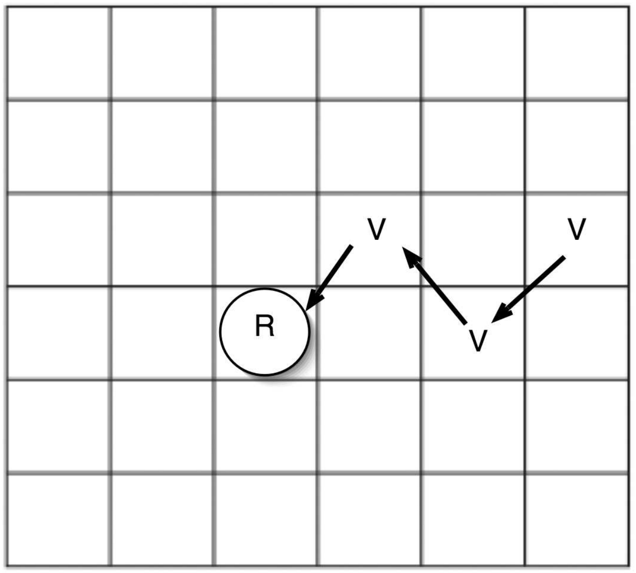

Figure 4.2: Simulation map R:Robot V:Visited

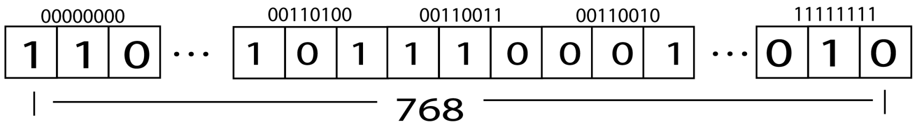

Figure 4.3: The encoding of the chromosome
Here a string with eight bit index and each address is a three bit string, the total length is 768

4.2.2 Edge and obstacle problem

The encoding of the chromosome considers the eight surrounding blocks adjacent to the center block which is the location of the robot. This method will not work while the robot is located at the edge of the map because there are only five blocks surrounding the robot with one wall at the side. In the worst situation, the robot will be located at the corner of the map, which has two walls and only three blocks nearby. There are two solutions for this problem. The first one is to develop a set of rules to handle these special cases, these rules will be activated when the robot runs to the wall-adjacent blocks. The second method is to evolve a set of rules to handle these cases using the same algorithm with another map to mark obstacles like edges and corners. These methods are used to handle obstacles also.

In the first solution, the rules are specified as followed:

If the robot is at the edge of the map, move one block away from the edge.

If the robot is at the corner of the map, move one block away from the corner in a diagonal direction.

Because of these extra rules, the robot can keep away from the edges and corners which are not covered by the general rules. The implementation of these rules should be adjusted due to differences of the problem between simulation and the physical world. The modifications of rules for the real environment are the following:

A1. If the movement is not a diagonal movement and it fails (north, south, east, west), move to the opposite of that direction (south, north, west, east).

B1. If the movement fails in the diagonal direction (northeast, southeast, northwest, southwest), the robot will move to the opposite of the initial direction. For example, if the attempt of moving northeast fails, it will try to move southwest.

B2. If rule B1 fails, with the same example above, it will try to move southeast.

B3. If B2 fails, the final attempt will move to the opposite direction of the third move, which will be northwest in this sample.

B4. If B3 fails too, the robot will stop.

By these rules, the robot may handle some edge problems in the real environment; however, it will still be in trouble in some cases like running into a loop or getting stuck in front of the obstacle. It is discussed in Chapter 5. There are several reasons leading to the failure of the edge rules, e.g. the irregular shape of the edge, the heading error of the robot and different surfaces of the wall. These rules should be considered more in future research.

In the second solution, we try to combine the rules for normal running and the rules for edge handling together. In Section 4.2.1, the simulation map has two status indicators, “visited” and “unvisited”. If we add one more status “obstacle”, each block on the map will need two bits to present its status and the environment data will increase from eight bits to 16 bits. This way is not practical because it will cause the chromosome to increase to over one hundred thousand bits. An independent obstacle map with the same format as the simulation map is added in our second solution. It has the same size as the simulation map, and each block on this map is marked “obstacle” or “non-obstacle”. Before the robot starts its move, it checks whether there are any obstacles in the adjacent blocks. If yes, it will use the obstacle map; otherwise, it will use the simulation map. The obstacle map will be updated when the robot detects obstacles. When the robot tries to move to a destination and cannot complete the movement because of a wall or other obstacle on its way, the destination block on the obstacle map will be marked “obstacle”.

We build two look-up tables, the first one uses environment data from the simulation map to look-up its action, the second one uses data from the obstacle map. They have similar sizes, both have an eight-bit index and three-bit storage space for each address. The length of the chromosome is doubled to contain these rules, the front part of the chromosome contains the look-up table for the simulation map, and the latter part of the chromosome contains the look-up table for the obstacle map. The robot controller will select the correct look-up table based on whether there are any known obstacles nearby.

4.2.3 Parameter settings

In order to get a set of good parameters for the algorithms, a test program was run with different parameters for five minutes each. Table 4.1 shows the comparison. In the final simulation program, the population size is set to 256, and the length of each chromosome is either 768 bits or 1536 bits which depends on the edge solution (Section 4.2.2). Single Point Crossover and Tournament Selection are used with a mutation rate of 0.1. Each generation,

Table 4.1: Comparison for different parameters

Test	Population	Elitism	mutation	selection	Generations	Best Fitness	Average Fitness
1	256	16	0.1	0.2	34	159	28
2	256	16	0.1	0.3	33	177	31
3	256	16	0.1	0.4	37	165	30
4	256	32	0.1	0.3	37	159	29
5	256	8	0.1	0.3	38	169	30
6	512	8	0.1	0.3	16	165	21
7	128	8	0.1	0.3	76	161	28
8	256	8	0.05	0.3	25	159	21
9	256	16	0.05	0.3	34	161	31

16 of the best individuals are cloned to the next generation. These parameters were selected because they helped the program evolve better results in the test. The genetic operators' parameters effect the speed of the process. In some cases it is critical to make progress by reducing the run time [Whi93], but the parameter values do not determine the performance of the experiment in this project. A faster running program may not generate better rules for the robot.

Simulation maps with different sizes are tested in the simulation. All maps are rectangular with different width and length ratios. The following map sizes are used in the experiment: 30 by 5, 30 by 10. Each unit of the map represents 30 centimeters in the real environment in later experiments. The first map size is based on the size of the real experiment environment and the second size is based on the size of a generic room.

4.2.4 Fitness function

The goal of the robot mission is to explore as much of the space as possible. Therefore, the more space it has visited/explored, the higher the fitness value it gets, and the fitness will be calculated based on the blocks of the map which are marked "visited". To encourage the robot to search the area along the wall, the fitness of visiting the boundary area is higher than for other locations. If an individual searches most of the space, it will have a high fitness value because it completed the task.

4.2.5 Simulation results

At the beginning of the simulation, the simulation map is initialized and the genes of the first generation are randomized. In the main loop of the evolution, each individual is placed at a starting point on the map, with each block of the map set to a starting point in turn. If we have a total number of N blocks in the map, the experiment runs N times such that the starting point will be different in different experiments. The evaluation of the current individual stops after the individual finishes six hundred moves. The second is set to prevent the simulation from falling into a loop, the value six hundred is twice the total number of blocks on the map and it is large enough for the robot to explore the map. After all individuals are evaluated, they are sorted by their fitnesses. The best 16 individuals are cloned into the next generation without modification. All other individuals will be selected using tournament selection and are exposed to crossover and mutation operators. Their children are sent to the next generation.

A successful simulation takes approximately 5 minutes to achieve acceptable results. Several results from the simulations are shown here. The following results are from chromosomes evolved from different map sizes in the simulation environment. The map sizes used here are size 30 by 5 and 30 by 10, these results show the robot starting from the center of the map. Almost all the rule-based controllers evolved from the simulation results can fully explore the map by different strategies, such as individual B and C.

Figure 4.4 and 4.7 shows movement tracks of the robot in a simulation with a map size 30 by 5. Map sizes of Figure 4.5 and Figure 4.6 are 30 by 10.

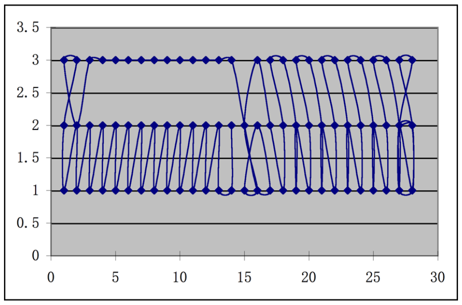
Figure 4.4: Individual A simulation

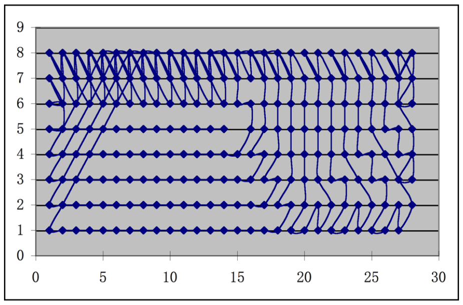
Figure 4.5: Individual B Simulation

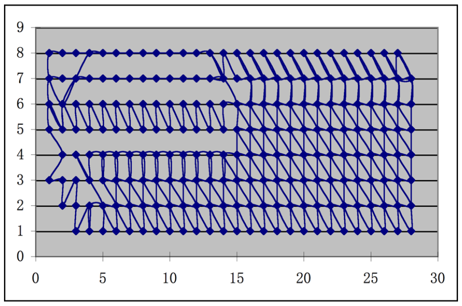
Figure 4.6: Individual C Simulation

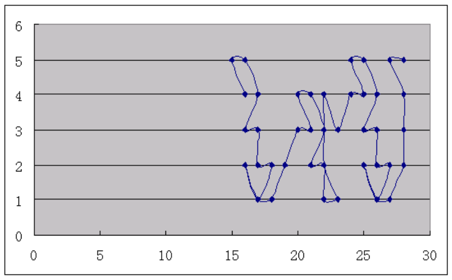
Figure 4.7: Individual D Simulation

4.3 Implementation

The second part of the software approach is implementing the rules evolved from the simulation on ER-1. The control system uses C++ and ERSP to connect hardware and software. The system uses the navigation command “Moveto” from the ERSP library to cause ER-1 to move to a target position. This command has built-in collision avoidance ability provided by the infrared sensors. If the way to the destination is blocked by an obstacle, the ER-1 will move back a little and try to move forward again to check if the obstacle disappears. It keeps trying until a timeout and sends the failure flag to the control program. There are some limitations about the “MoveTo” command: it can’t move to a point behind it and too close to it, for example, within 10 cm behind its back. The robot will just keep turning until timeout. Therefore, each step of the move must be large enough to prevent this problem. The command "Position" is applied after each movement command to record the current robot position. This value is used to decide whether the robot stops at the position as requested. However, when the movement number increases, the errors from the position sensors accumulate, and the actual position will be different from the position shown by the position sensors.

4.3.1 Maps

There are two maps used in the real robot implementation. The first one is called the blackboard map in this thesis and it has the same structure as the simulation map in the robot simulation. It records the history of the movement at an abstract level and the relative positions of the robot in the space. It also provides the environment data used by the robot to make its decisions. The second map, called the real map, is used to record the real movement of ER-1 provided by its position sensors. Because the robot may not arrive at the exact position assigned by the movement command, the distance between the proposed position and the actual position provided by its position sensors is calculated. If the distance is within the acceptable range, the ER-1 may judge the movement as successful; otherwise the movement fails. The acceptable distance is 10 cm which is based on the ER-1's performance. Based on the size of ER-1 and the range of infrared sensors, the distance between two blocks in the blackboard map is defined as 30 cm in the real map. The movement of the real robot is recorded on this map for analyzing results which are discussed in Chapter 5.

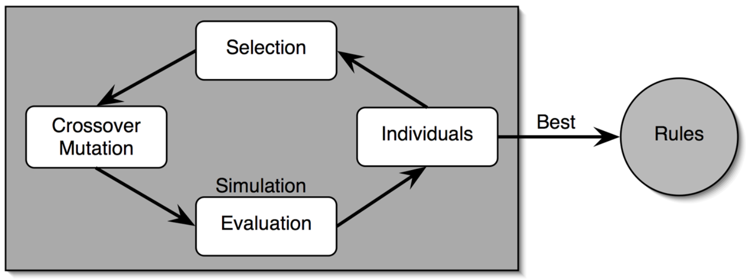

Figure 4.8: Software flow of simulation

4.3.2 Rules

The rules used to control ER-1 are determined by the chromosome generated by the simulation shown in Figure 4.8. The system reads the environment data from the blackboard map, uses the eight-bit genes as an index to look up the action, then applies the action on the robot and sends out the movement commands based on the current position. The infrared sensors prevent the robot from colliding with obstacles it meets. If an obstacle is in its way, the robot keeps a distance of about two feet from the obstacle and keeps trying to move forward until a timeout or the obstacle disappears. After the movement, the robot position is evaluated and used to determine the success of the movement. If the last movement failed, the robot updates its obstacle map or follows the rules for handling edge and obstacles to leave those obstacles (Section 4.2.2).

4.4 Improvement of the chromosome design

The length of the chromosome in the previous sections is much longer than ordinary chromosome design. To improve the experiment results, the length of the chromosome must be shortened. Symmetry design is used to achieve this enhancement. In Figure 4.9, the environment is different in the view of the robot if we use the previous chromosome encoding method, in fact, these situations can be regarded as the same situation if we consider symmetry. A binary number can represent this symmetrical situation with a left shift or right shift of the bits. For example, 00000001, 00000100, 00010000 and 01000000, can be generated by applying left shifting 0, 2, 4, 6 bits on binary number 00000001. In some cases, the symmetrical numbers are less than four, for example, 11111111, whose symmetrical numbers are itself. A hash table is built to record the mapping and shifting. Table 4.2 shows a part of this hash table. Therefore the encoding of the environment data is shortened from 256 to 70, and the length of the chromosome becomes 70*3=210, nearly 1/4 of the original chromosome design.

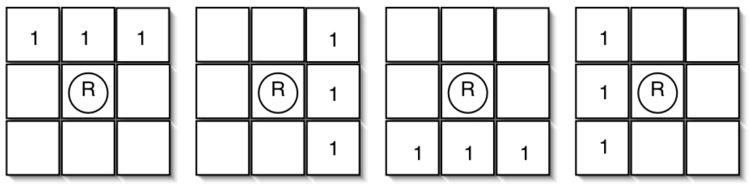

Figure 4.9: Symmetry environment

Table 4.2: Using symmetry to reduce the environment value

Original number	Mapping number	shift bits(by two)
0	0	0
1	1	0
2	2	0
3	3	0
4	1	1
5	4	0
6	5	0
7	6	0
8	2	1
9	7	0
10	8	0
11	9	0
12	3	1
13	10	0
14	11	0
15	12	0
16	1	2

To improve the performance of the algorithm, the fitness function is modified and a larger population of 1024 is used. The new fitness function adds a penalty for diagonal movement to reduce distance, a penalty for direction changing in movement to reduce unnecessary turning, and a penalty for revisiting the same position to save time. Figure 4.10 and Figure 4.11 show the progress of the best fitness and the average fitness while generation increases in the simulation. They are based on different solutions to handle edge/obstacles. Figure 4.10 is based on the first solution and Figure 4.11 is based on the second solution.

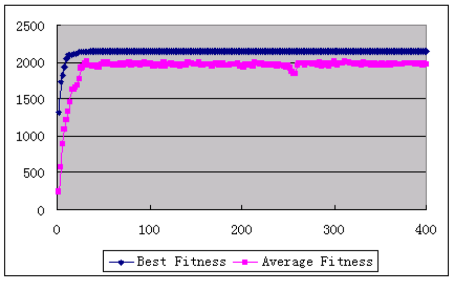

Figure 4.10: Fitness vs. Time(Solution 1)

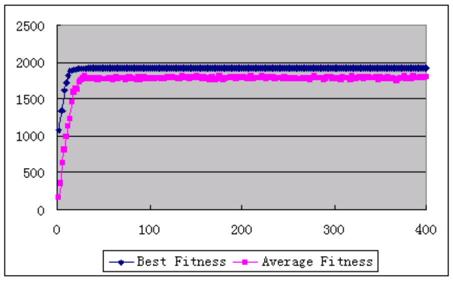

Figure 4.11: Fitness vs. Time(Solution 2)

Figure 4.12 shows the performance of the best individual in the simulation after 18 hours of evolution cycles. This individual uses the first solution rules to handle edge/obstacles. Figure 4.13 shows the performance of the best individual in the simulation after 18 hours of evolution cycles. This individual uses evolved rules to handle edge/obstacles. Both simulation results show very satisfactory results, and both figures are clear and clean. The turning and repeatedly visiting behaviors are far less than the previous results, also the movements are mainly in straight lines instead of diagonals. The detailed performance of these individuals in the real environment is discussed in section 5.3.

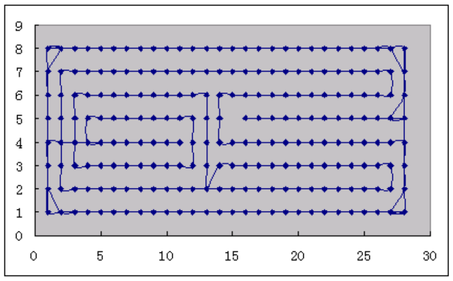

Figure 4.12: Individual E’s simulation

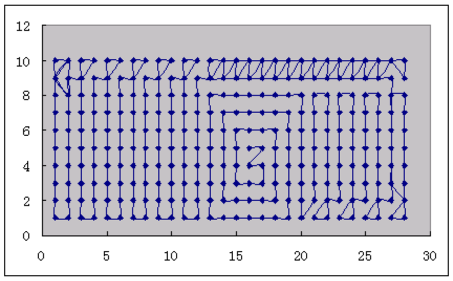

Figure 4.13: Individual F’s simulation

Chapter 5

Experiments

5.1 Experimental setup

The experiments are performed in a narrow straight corridor and in a wide lobby hall with moderate indoor light conditions. The size of the corridor is approximately 90 feet by 8 feet, and the hall size is approximately 50 feet by 40 feet. Their shapes are rectangular with several open exits and doors. There are no obvious obstacles inside the corridor, although there are columns and a fountain along the wall. There are two columns in the middle of the lobby hall that are 20 feet apart. The materials of the walls and exits are variable. They are glass, metal, wood as well as brick and concrete. These testing environments are not an ideal place for robot experiments, e.g. the glass door does not reflect enough infrared signal; however, these conditions make experiment results more practical. Figure 5.1 shows a picture of the corridor.

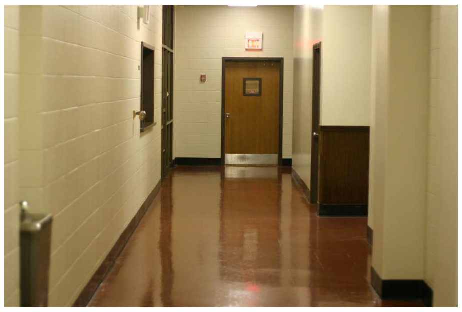

Figure 5.1: The testing environment

The goal of this project is to use a genetic algorithm to evolve the rules which control a robot exploring an unknown space. To evaluate the results, some criteria are selected. First, the robot is expected to explore as much space as possible. Second, the movements to complete the first task are as few as possible. Finally, the robot is expected to handle the edge/obstacle problem.

The robot is placed in the center of the corridor or the lobby hall with different heading either facing north or facing east at the beginning of each experiment. The speed of movement is approximately 10 cm/sec and the speed of rotation is approximately 30 degree/sec.

In simulation, two different simulation map sizes are used. One is a 30 by 5 grid and another is a 30 by 10 grid. The size of the first map is based on the real environment, and the second one is a generic size for comparison. The starting point of the robot is either at (15,3) or (15,5). In the real environment testing, the blackboard map size of the experiment is 100 by 100 and the starting point is (50,50). Such a map size is used in the real experiments because the experiment environment may be much larger than the simulation environment, and it needs enough storage space to record its movements. Both maps are the implementations for the blackboard model of the rule-based controller.

5.2 Experimental results

The following results are from the log files which record the movements of the robot. One file records the movement in the grid map; another file records the movements and positions within the real environment. The dots on these figures are the coordinates saved by the control computer. They are connected by lines based on the order of time. All tests are stopped when the robot moves repeatedly in a closing small circle or sticks before the corner or edge of the wall.

5.2.1 Result A

Figure 5.2 is a part of the simulation result of individual A. It is evolved from a 30 by 5 simulation map whose size is similar to the real environment. Figure 5.3 shows the controlling history of individual A in the blackboard map, and Figure 5.4 shows the record of the real robot movement in the corridor. This controller commanded the robot to do a zig-zag move along the wall at first, and after it detected an obstacle, which is attached on the wall, it switched its way to the other side of the corridor and preferred the zig-zag exploration again. In this test, the robot explored nearly 50% of the space with a zig-zag strategy as the main search method. It went as far as 15 meters and as wide as one meter in the corridor. This is the best result among all four results discussed here. The reason for the good result may rely on the similarity between the simulation map and real environment.

Figure 5.5 and Figure 5.6 show the controlling history and the record of robot movement in the lobby hall. The individual did a zig-zag move to the west at the beginning and met the column inside the lobby hall. It moved away from the obstacle and moved back to the starting point directly. Then it started to explore the south and finally fell into a looping movement, which is between (49, 48) and (48, 48) in Figure 5.5. The results shown here have a good matching with the simulation results in Figure 5.2. The robot ran inside an area which is three meters long and 1.5 meters wide. The distance that the robot moved in the lobby hall is shorter than the result in the corridor because of the obstacles. These results show that the exploring strategy of individual A is suitable for a narrow corridor but not for a wide lobby hall.

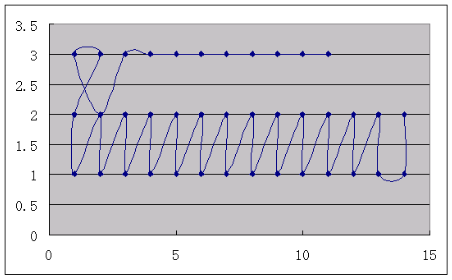

Figure 5.2: Part of Individual A's simulation

5.2.2 Result B

Figure 5.7 is part of the simulation result of individual B. It is evolved from a 30 by 10 simulation map. Figure 5.8 shows the controlling history of individual B in the blackboard map and Figure 5.9 shows the record of the real robot movement in the corridor. This

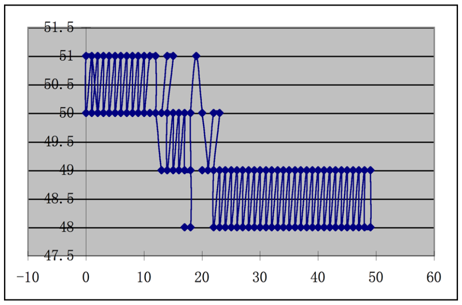

Figure 5.3: Individual A controlling history in blackboard map (corridor)

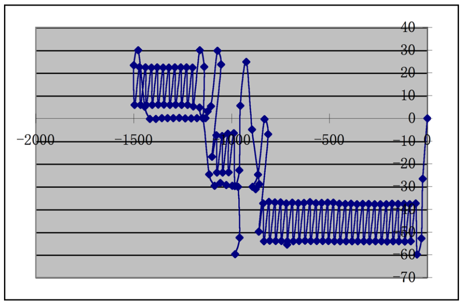

Figure 5.4: Individual A movement (corridor)

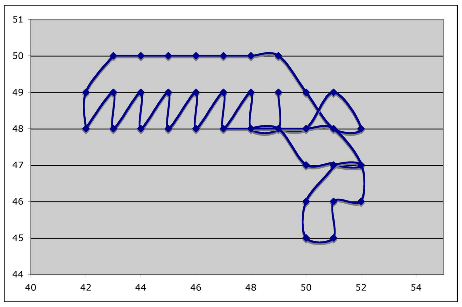

Figure 5.5: Individual A controlling history in blackboard map (lobby)

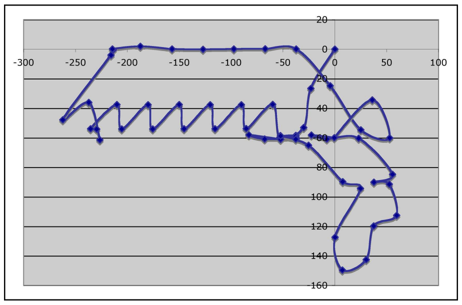

Figure 5.6: Individual A movement (lobby)

controller did a great job in simulation, and it used a strategy different from individual A to explore different parts of the map. But in real testing, individual B ran fine at the beginning then faced problems after moving four meters from the starting point. This individual did not move in the zig-zag track like individual A, but it ran through the space step by step also.

Figure 5.10 and 5.11 show the result of the controlling history and the record of robot movement in the lobby hall. This result shows excellent performance of individual B. It began with a straight movement to the west, then it turned to explore the north side of the lobby hall by progressively scanning the space. After four meters of movement, it detected the wall and turned to the south. In the corridor experiment, this individual got stuck before the wall at this stage, but it managed to escape from the wall of the lobby hall because the larger space in the lobby gave it a better chance to manipulate its movement. The robot then repeatedly moved south and north to scan the space, this strategy is efficient and sufficient to explore the space fully. This result shows that the exploring strategy of individual B is suitable for a wide lobby hall but not for a narrow corridor. This result also shows that a problem exists in the position sensor of ER-1. We can see the positions in the upper part of the map gradually increased from 400 to 500, actually their positions are all approximately 400, four hundred centimeters north from the starting point. These reading errors accumulated gradually while the experiment was in progress.

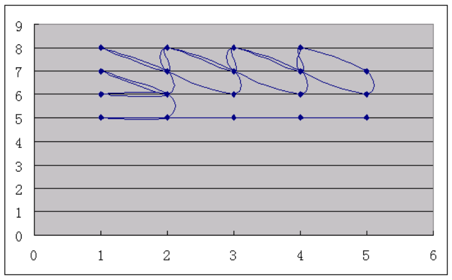

Figure 5.7: Part of Individual B's simulation

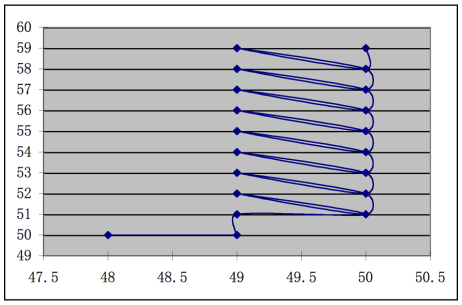

Figure 5.8: Individual B controlling history in blackboard map (corrdior)

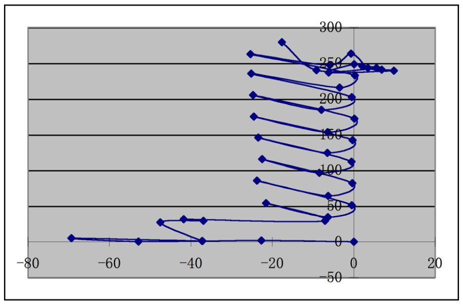

Figure 5.9: Individual B movement (corrdior)

5.2.3 Result C

Figure 5.12 is part of the simulation result of individual C. It is evolved from a 30 by 10 simulation map. Figure 5.13 shows the controlling history of individual C in the blackboard

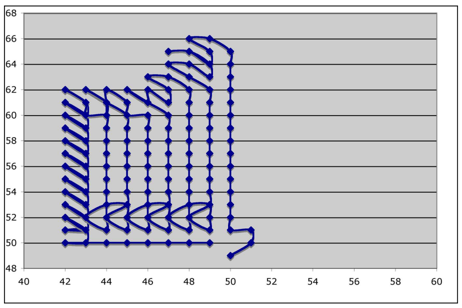

Figure 5.10: Individual B controlling history in blackboard map (lobby)

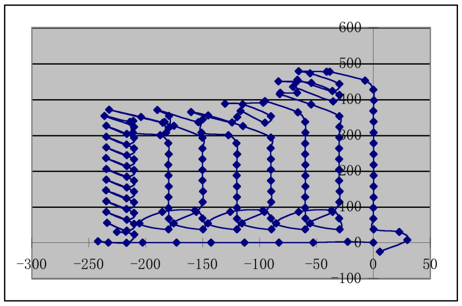

Figure 5.11: Individual B movement (lobby)

map, and Figure 5.14 shows the record of the real robot movement in the corridor. The zig-zag searching behavior is obvious from its simulation. The robot keeps this searching pattern until it detects something in its way. After several attempts to avoid the obstacle, it decides to move straight back to its starting point along its previous track. This behavior is interesting because it imitates the behavior of a living creature, which spends time on searching and saves time on its return. However, this individual did not search a large area. It only moved three meters, which is 1/5 of individual A.

Figure 5.15 and 5.16 show the result of the controlling history and the record of robot movement in the lobby hall. This experiment result shows that the individual moved to the west with a zig-zag searching behavior until it met the column in the lobby, then it moved southeast with the same searching pattern. It reached the entrance doors of the lobby hall and turned its course again to the west. It ran as far as eight meters before it arrived at the wall of the west boundary of the lobby hall. It moved south again to explore the southwest corner of the lobby hall. Finally, it ended up looping between (31, 35) and (30, 36) in its control map. This result has the longest movement path among all the results, the distance it traveled was more than 16 meters, and it successfully handled different walls and obstacles three times. This individual has relatively better results in the lobby hall than in the corridor.

The position sensor problem also exists here, the offset of its final position on Figure 5.16 is approximately one meter from its actual position. The farther it goes, the more errors the position sensor accumulates.

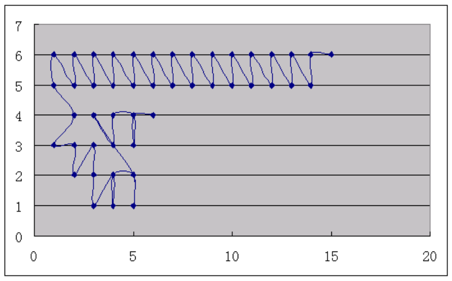

Figure 5.12: Part of Individual C's simulation

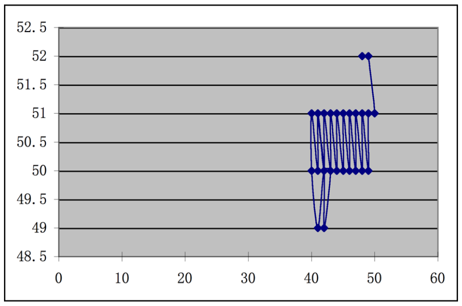

Figure 5.13: Individual C controlling history in blackboard map(corridor)

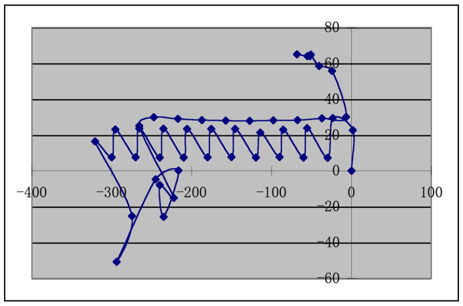

Figure 5.14: Individual C movement(corridor)

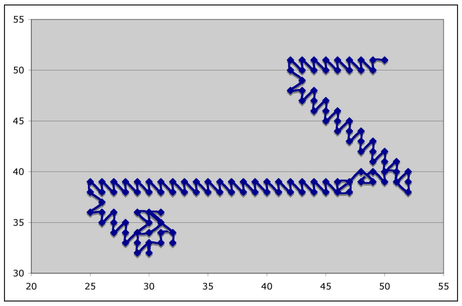

Figure 5.15: Individual C controlling history in blackboard map (lobby)

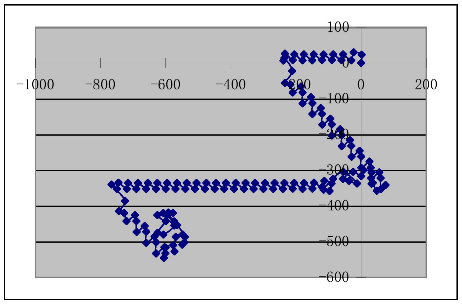

Figure 5.16: Individual C movement (lobby)

5.2.4 Result D

Individual D used the second solution method to handle edges and corners (Section 4.2.2). The length of the chromosome in this result is 768*2, twice as much as the others. This length makes the individuals much harder to improve in the simulation. Only half of the map is visited by the robot in simulation. Figure 5.17 shows part of the simulation results of individual D. Figure 5.18 shows the controlling history of individual D in the blackboard map, and Figure 5.19 shows the record of the real robot movement in the corridor. This result shows that the robot has explored a space 4 by 2 meters and at the same time handles the edges quite well, but it finally ends with looping in a cycle two meters north from its starting point in Figure 5.19.

Figure 5.20 and 5.21 show the result of the controlling history and the record of robot movement in the lobby hall. The searching pattern of the individual is quite different from previous individuals. It explored a three by three square meter area and met the obstacle, the column three meters east to it starting point. The robot manages to avoid it with the rules evolved in simulation, but it fell into looping among positions (57, 53), (56, 54) and (57, 54) in Figure 5.20 soon after that. This result shows that the rules evolved from simulation for edge and obstacle detection did work, but it is not as efficient as the pre-developed rules.

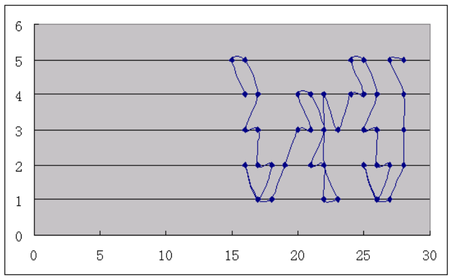

Figure 5.17: Part of Individual D's simulation

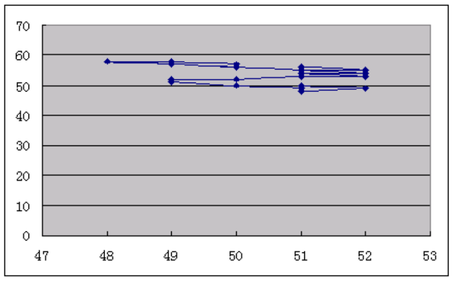

Figure 5.18: Individual D controlling history in blackboard map (corridor)

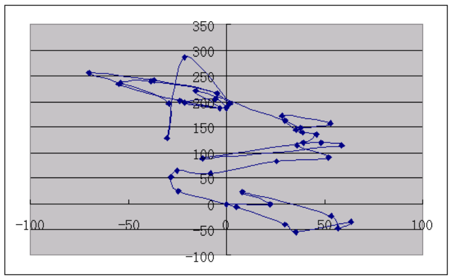

Figure 5.19: Individual D movement (corridor)

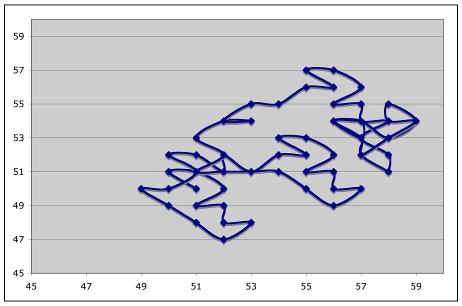

Figure 5.20: Individual D controlling history in blackboard map (lobby)

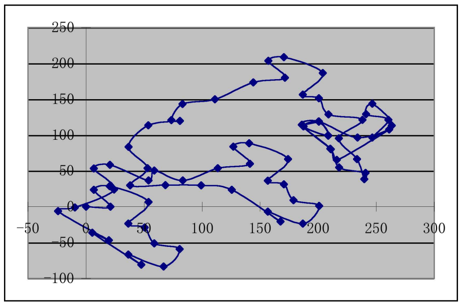

Figure 5.21: Individual D movement (lobby)

5.3 Results of Improved chromosome design

5.3.1 Individual E

Figure 4.12 is the simulation result of individual E. It is evolved from a 30 by 10 simulation map with the first edge/obstacle solution. Figure 5.22 shows the controlling history of individual E in the blackboard map, and Figure 5.23 shows the record of the real robot movement in the corridor. The robot moves east in a straight line at the beginning until it reaches the end of the corridor, then it starts to search the sideway. After a few trails, it gives up the sideway movement and moves back to the corridor it comes from, and finally sticks in front of the obstacles.

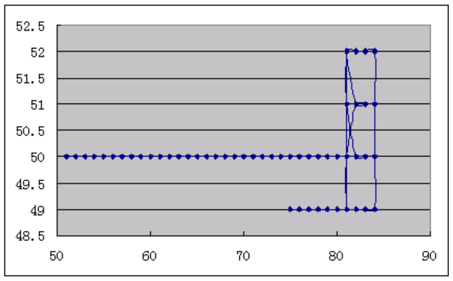

Figure 5.22: Individual E controlling history in blackboard map (corridor)

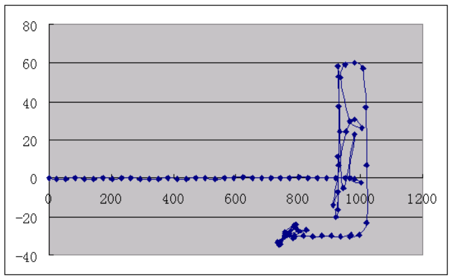

Figure 5.23: Individual E movement (corridor)

Figure 5.24 and 5.25 show the result of the controlling history and the record of robot movement in the lobby hall. The performance of the robot in the lobby hall is significant. The robot starts from the middle of the lobby; moves east first until it detects an obstacle, then it moves south and reaches the wall on the south. Each time when the robot meets an obstacle, it makes a sharp right turn. This robot moves along the border of the lobby hall and successfully explores the whole area of the lobby.

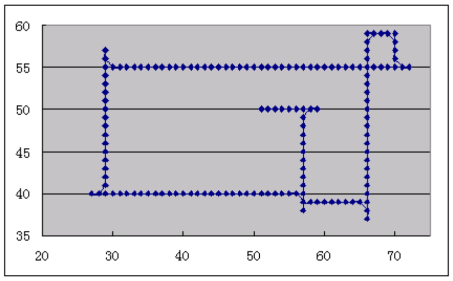

Figure 5.24: Individual E controlling history in blackboard map (lobby)

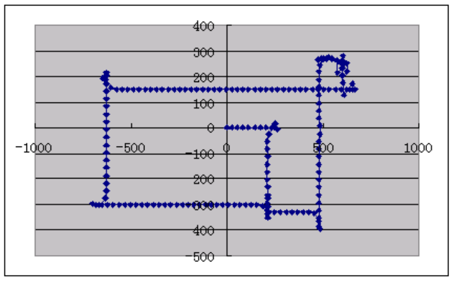

Figure 5.25: Individual E movement (lobby)

5.3.2 Individual F

Figure 4.13 is the simulation result of individual F. It is evolved from a 30 by 10 simulation map with the second edge/obstacle solution. Figure 5.26 shows the controlling history of individual F in the blackboard map, and Figure 5.27 shows the record of the real robot movement in the corridor. The track of the robot movements is in a kind of S shape. It shows that the wall of the corridor blocks the robots movement many times, and the robot is stuck at last. This result shows the limitation of the robots searching strategy in a narrow area.

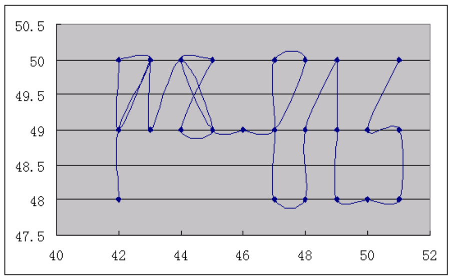
Figure 5.26: Individual F controlling history in blackboard map (corridor)

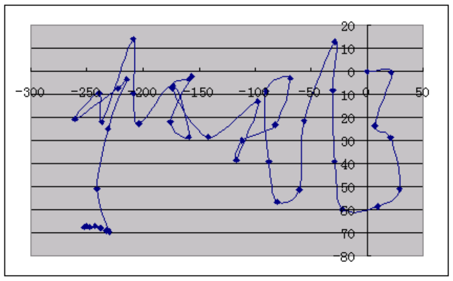
Figure 5.27: Individual F movement (corridor)

Figure 5.28 and 5.29 show the result of the controlling history and the movement track of individual F in the lobby hall. The performance of the robot in the lobby hall is satisfactory. The robot starts from the middle of the lobby, keeps drawing circles from its starting point with the radius gradually increasing. The track ends up with a spinning curve. An obstacle blocks the advancing robot and makes it turn to move to north, then it starts to explore the north side of the lobby and finally stops in front of the north wall of the lobby hall. This individual fully explores the center part and the north part of the lobby hall. If there is no obstacle near the center of the lobby, the robot should be able to explore more area in the lobby with its searching strategy.

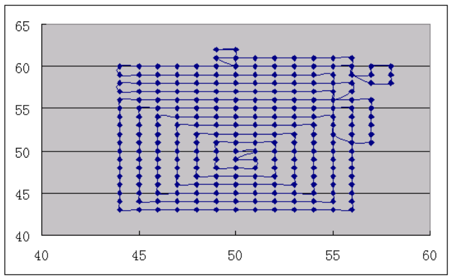
Figure 5.28: Individual F controlling history in blackboard map (lobby)

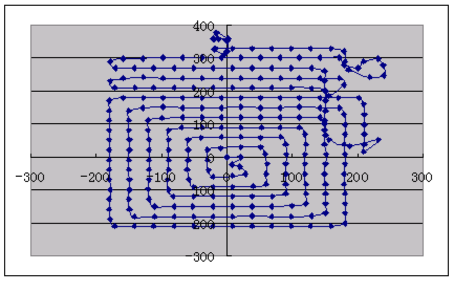
Figure 5.29: Individual F movement (lobby)

5.4 Result analysis

Based on the results in the previous sections, the best controller-individual, evolved from the evolutionary algorithm in the simulation has the ability to explore the space as requested. The strategies of exploration used in the simulation were effective, and these strategies showed parts of their functionalities in the real environment. By redesigning the chromosome and the fitness function, the new experiments show satisfying results. Individual E and F show great improvement in both the simulation and the real environment.

The robot, when tested in a different environment, shows different performance. Generally speaking, the individuals perform relatively better in the lobby hall than the corridor. The main reason is the narrow corridor causes the robot to spend too much time handling the walls and obstacles, instead of exploring the space. The rules to handle the edge problem come from two solutions. Both solutions solve the problem to some degree. They work better in the lobby hall and have limitations in the corridor. In most cases, the robot ended up with a looping movement during the exploration. No rule was developed or evolved to solve this problem currently. We need to consider this problem in future research. Another important issue is the accuracy of the position sensors in long runs. Many good position sensors will have this problem, because there is no method to calibrate them while running. There are some new methods to solve this problem, such as the NorthStar system from Evolution Robotics Inc. [Inc05], which uses two infrared points on the ceiling to locate the robot.

Chapter 6

Conclusions

6.1 Conclusions

This thesis discusses an experimental approach to develop a rule-based control robot to explore unknown space. Our rule-based control system is an efficient architecture for autonomous robots. This thesis introduces an attempt of developing the rules with Evolutionary Algorithms. A simulation environment was built to evaluate the performance of rules which were designed by the machine. Artificial evolution running in the simulation enhanced the rules step by step. The best evolved rule set was chosen and implemented on a real robot platform, ER-1. ER-1 is a three-wheel robot equipped with infrared sensors, a camera and position sensors. The central control program is stored inside the laptop computer which is carried by the ER-1. The same laptop was the simulation platform also.

We expect the Evolutionary Algorithm to evolve a number of controllers with different strategies to explore the space, because they are generated randomly and enhanced by the Evolutionary Algorithm. This expectation has been demonstrated by the results in Chapter 5. We also anticipate the real robot implementations will have similar behaviors with the simulation; however, the results show that there is some difference between the simulation results and experimental results. The explanation is that the simulation simplifies the testing environment and ignores the complicated conditions such as the shape of the testing environments, infrared reflection and errors accumulated by sensors. Overall, the strategies of exploration developed by the Evolutionary Algorithm are effective in the simulation, but only showed parts of their functionalities in the real environment.

6.2 Future directions

Evolutionary Algorithms are suitable tools for rule-based control robotics design. The most challenging work in this field is the use of a binary string to represent the relationship between the complex environment data received from the robot sensors and control actions sent by it. The robot design is limited by the hardware technology also.

To improve the experiment in the future, some modifications can be done. First, a larger testing field with a unique wall surface and better lighting conditions may improve the environment condition. Second, the rules to handle edges and corners of the testing environment can be better integrated with the normal rules to improve the performance. Third, if the size of the robot is reduced, the controller would be less complex and the accuracy of the movement would be improved. Finally, more details should be added into the simulation to match the real environment.

Bibliography

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies a comprehensive introduction. *Natural Computing*, 1:3–52, 2002.

[Dar59] Charles Darwin. *The origin of species.* Oxford University Press, Oxford, UK., 1859.

[dCT02] L. N. de Castro and J. Timmis. *Artificial Immune Systems: A Novel Paradigm to Pattern Recognition*, chapter 5, pages 67–84. University of Paisley, UK,, 2002.

[DM93] Floreano D. and F. Mondada. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Proceedings of the Third International Symposium on Experimental Robotics*, pages 501–513, Kyoto, Japan., 1993.

[HB91] F. Hoffmeister and T Bäck. Extended selection mechanisms in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 92–99, San Mateo, California, USA, 1991.

[Hof94] Frank Hoffmann. Soft computing techniques for the design of mobile robot techniques for the design of mobile robot behaviours. *Information Sciences*, 122(2-4):241–258, 1994.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, first edition, 1992.

[Hol01] Gordon S Hollingworth. *Fault Tolerance of Evolvable Hardware Through Diversification.* PhD thesis, The University of York, September 2001.

[HP75] Schwefel H-P. *Evolutions strategie und numerische Optimierung.* PhD thesis, Technical University of Berlin, Germany, 1975.

[HP87] Schwefel H-P. Collective phenomena in evolutionary systems. In *31st Annual Meeting International Society for General System Research*, pages 1025–1033, Budapest, Hungary, 1987.

[I71] Rechenberg I. *Evolutions strategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* PhD thesis, Technical University of Berlin, Department of Process Engineering, 1971.

[Inc04a] Evolution Robotics Inc. *ER-1 User Manual*, 2004.

[Inc04b] Evolution Robotics Inc. *ERSP 3.0 Overview*, 2004.

[Inc05] Evolution Robotics Inc. *NorthStar System*, 2005.

[KKI$^+$98] Didier Keymeulen, Kenji Konaka, Masaya Iwata, Yasuo Kuniyoshi, and Tetsuya Higuchi. Robot learning using gate-level evolvable hardware. In *The Sixth European Workshop on Learning Robots*, page 173, Brighton, UK, 1998. Springer-Verlag London, UK.

[Koz92] J. R. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, U.S.A., 1992.

[NFMM94] Stefano Nolfi, Dario Floreano, Orazio Miglino, and Francesco Mondada. How to evolve autonomous robots: Different approaches in evolutionary robotics. In *Proceedings on the Artificial Life IV Conference*, pages 190–197, Cambridge, MA, U.S.A., July 1994.

[ONT94] Miglino O., K. Nafasi, and C. Taylor. Selection for wandering behavior in a small robot. *Artificial Life*, 2(1):101–116, 1994.

[TKZ04] A.M. Tyrrell, R.A. Krohling, and Y. Zhou. A new evolutionary algorithm for the promotion of evolvable hardware. *IEE Proceedings of Computers and Digital Techniques*, 151(4):267–275, July 2004.

[Whi89] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Whi93] Darrell Whitley, editor. *Evolutionary Computation*, volume 1. MIT Press, 1993.

[XB97] H. Xum and H. Van Brussel. A behaviour-based blackboard architecture for reactive and efficent task execution of an autonomous robot. *Robotics and Autonomous Systems*, 22:115–132, 1997.

Appendix A

Source Code

A.1 Genetic Algorithm Codes

```
import java.util.Random;

public class GA{
    int bestfit;
    Random gen;
    public GA(){
        gen = new Random();
    }

    public boolean[][] randominit(int population, int length){
        boolean[][] genes= new boolean[population][length];
            for (int i=0; i<population; i++){
                for (int j=0; j< length; j++){
                    genes[i][j]=gen.nextBoolean();
                }
            }
        return genes;
    }

    /////   Sort gens by their fitness value
    public boolean[][] sorting(boolean[][] oldgenes, int[] fit,
        int length, int population){
        int tempfit;
        boolean[] tempgene = new boolean[length];
        boolean[][] newgenes= new boolean[population][length];
        int[] fitvalue =new int[population];  //fitness

        //  copy values to local variables
        for (int i=0; i< population; i++){
            fitvalue[i]=fit[i];
            for (int j=0; j<length; j++){
                newgenes[i][j]=oldgenes[i][j];
            }
        }
        // Sort them
        boolean[] tempgenes= new boolean[length];
        for (int i=0; i< population; i++){
            for (int j=0; j<population-1; j++){
                if (fitvalue[j]< fitvalue[j+1]){
                    //switch fitness
                    tempfit=fitvalue[j];
                    fitvalue[j]=fitvalue[j+1];
                    fitvalue[j+1]=tempfit;
                    //switch genes
                    for (int k=0;k<length;k++){
                        tempgene[k]=newgenes[j][k];
                        newgenes[j][k]=newgenes[j+1][k];
                        newgenes[j+1][k]=tempgene[k];
                    }
                }
            }
        }
        bestfit = fitvalue[0];
        return newgenes;
    }// end of sort

    ///// *****  selection, pick up a good guy to be a parent *******
    public int selection(int population,double selection){


        float f1=gen.nextFloat()*(float)population;
        float f2=gen.nextFloat()*(float)population;
        float ftemp;
        if (f1>f2) {
            ftemp=f1;
            f1=f2;
            f2=ftemp;
        }
        int i1=(int)f1;
        int i2=(int)f2;
        if (gen.nextFloat()>selection)
            return i1;
        else
            return i2;
    }//end of selection

    ///// *****       Crossover   ************************
```

```
    public boolean[][] crossover(boolean[][] oldgenes,
                int population, int length){
        boolean[][] newgenes= new boolean[population][length];
        int individualA;
        int individualB;
        for(int i=0; i<32; i++){
            for(int j=0; j <length; j++)
                newgenes[i][j]=oldgenes[i][j];
        }

        for(int j=32; j< population; j++){
            individualA=selection(population,0.3);
            individualB=selection(population,0.3);
            int point=gen.nextInt(length);
            for(int i= 0; i < point; i++){
                newgenes[j][i]=oldgenes[individualA][i];
            }
            for(int i =point; i<length; i++){
                newgenes[j][i]=oldgenes[individualB][i];
            }
        }
        return newgenes;
    }//  End of crossover

    public boolean[][] mutation(boolean[][] oldgenes,
        int population, int length, double mutate){
        boolean[][] newgenes= new boolean[population][length];
        int temp;
        newgenes= oldgenes;
        for (int i =32; i< population; i++){
            for (int j=0; j<(int)(mutate*length); j++){
                temp = gen.nextInt(length);
                newgenes[i][temp]= !newgenes[i][temp];
            }
        }
        return newgenes;
    }// End of mutation

    public int getBest(){
        return bestfit;
    }
}
```

A.2 Simulation Codes

A.2.1 Simulation with pre-set Edge/Obstacle rules

```
//   This program run with TestMain.java
//   To do the simulation
import java.util.Random;
import java.io.*;

public class SimERSP{

    final int wide=30;
    final int height=10;
    final int population=256;
    final int geneLength=768;
    final int afit=120;
    final int bfit=400;
    int x,y, door;
    int[] fitness = new int[population];
    Random gen;
    boolean[][] gene= new boolean[population][geneLength];
    boolean[] bestGene = new boolean[geneLength];
    boolean[][] map = new boolean[wide][height];
    boolean[][] bestmap = new boolean[wide][height];
    GA ga;
    PrintWriter outFile, outGene;

    public SimERSP(){
      gen = new Random();
      ga = new GA();
      initOthers();
      run();
    }

    public void initOthers(){
      for (int i =0; i<population; i++)
        fitness[i]=0;
      gene=ga.randominit(population, geneLength);
    }

    public void initMap(){
      for (int i=0; i < wide; i++){
        for (int j=0; j< height; j++){
          map[i][j]=false;
        }
      }
      map[x][y]=true;
      //System.out.println("map clean");
    }

    public void run(){
      int address,tempfit,round=0, highest=0, environment;
      double avefit=0;
      while(avefit<afit || ga.getBest()<bfit){
      //evolve loop
        door = 26;
        for (int i=0; i<population; i++){
        // adding a loop here
          tempfit=0;
          for (int sx=1; sx<wide-1; sx++){
            for (int sy=1; sy<height-1; sy++){
              x=sx;
              y=sy;
              initMap();
              //run loop
              for (int j =0; j< 600; j++){
                //get inforn from 8 directions
                environment=readEnvironment();
                address=environment*3;
                  if (gene[i][address+2]){
                    if (gene[i][address+1]){
                      if(gene[i][address]){
                        //case 111
                        x=x-1;
                        y=y+1;
                        if (fail(x,y)){
                          x=x+2;
                          y=y-2;
                          if (fail(x,y)){
                            y=y+2;
                            if (fail(x,y)){
                              x=x-2;
                              y=y-2;
                              if (fail(x,y)){
                                System.out.println("Error at 111");
                                return;
                              }
                            }
                          }
                        }
                      }else{
                        //case 110
                        y=y+1;
                        if(fail(x,y))
                        y=y-2;
                      }
                    }else{
                      if(gene[i][address]){
                        //case 101
                        x=x-1;
                        if (fail(x,y))
                          x=x+2;
                      }else{
                        //case 100
                        x=x-1;
                        y=y-1;
                        if (fail(x,y)){
                          x=x+2;
                          y=y+2;
                          if (fail(x,y)){
                            x=x-2;
                            if (fail(x,y)){
                              x=x+2;
                              y=y-2;
                              if(fail(x,y)){
                                System.out.println("Error at 100");
                                return;
                              }
                            }
                          }
                        }
                      }
                    }
                  }else{
                    if (gene[i][address+1]){
                      if(gene[i][address]){
                        //case 011
                        x=x+1;
                        if(fail(x,y))
                          x=x-2;
                      }else{
                        //case 010
                        x=x+1;
                        y=y+1;
                        if(fail(x,y)){
                          x=x-2;
                          y=y-2;
                          if(fail(x,y)){
                            x=x+2;
                            if(fail(x,y)){
                              x=x-2;
                              y=y+2;
                              if(fail(x,y)){
                                System.out.println("Error at 010");
                                return;
                              }
                            }
                          }
                        }
                      }
                    }else{
                      if(gene[i][address]){
                        //case 001
                        x=x+1;
                        y=y-1;
                        if(fail(x,y)){
                          x=x-2;
                          y=y+2;
                          if(fail(x,y)){
                            y=y-2;
                            if(fail(x,y)){
                              x=x+2;
                              y=y+2;
                              if(fail(x,y)){
                                System.out.println("Error at 001");
                                return;
                              }
                            }
                          }
                        }
                      }else{
                        //case 000
                        y=y-1;
                        if(fail(x,y))
                          y=y+2;
                      }
                    }
                  }
                  if(!fail(x,y)){
```

```
                    map[x][y]=true;
                 }else{
                    System.out.println("Error for running");
                    return;
                 }
                 if(y==(height-2) && (x == door
|| x==(door+1) || x==(door+2)) ){
                   j=999;
                   tempfit = tempfit +wide*height;
                 }
             }// run loop
             //calculation fitness
             for (int ii=0; ii < wide; ii++){
               if (map[ii][1])
                 tempfit++;
               if (map[ii][height-2])
                 tempfit++;
               for (int j=0; j< height; j++){
                 if (map[ii][j])
                   tempfit++;
               }
             }
             for (int ii=0; ii<height; ii++){
               if (map[1][ii])
                 tempfit++;
               if (map[wide-2][ii])
                 tempfit++;
             }
             if(tempfit>highest){
               highest=tempfit;
               for (int mm=0; mm<wide; mm++){
                 for (int nn=0; nn<height; nn++){
                   bestmap[mm][nn]=map[mm][nn];
                 }
               }
             }
           }//sx and sy
         }
         fitness[i]=tempfit/(wide*height);
       }//population loop
       avefit=0;
       for (int i=0; i<population; i++)
         avefit+=fitness[i];
       avefit=avefit/population;
       gene = ga.sorting(gene, fitness, geneLength, population);
```

```
      System.out.println(avefit + " " + ga.getBest());
      for (int wi=0; wi< geneLength; wi++){
        bestGene[wi]=gene[0][wi];
      }
      round++;
      gene = ga.crossover(gene, population, geneLength);
      gene = ga.mutation(gene, population , geneLength, 0.04);
    }
    //evolve loop
    try{
      outFile = new PrintWriter(new FileWriter("Map.txt"), true);
      for (int i=0; i < height; i++){
        for (int j=0; j< wide; j++){
          if(bestmap[j][i]){
            outFile.print("1 ");
          }else{
            outFile.print("0 ");
          }
        }
        outFile.println();
      }
      outGene = new PrintWriter(new FileWriter("Gene.txt"), true);
      for (int i=0; i< geneLength; i++){
        if (bestGene[i]){
          outGene.print(1);
        }else{
          outGene.print(0);
        }
      }
      outGene.println();
    }
    catch(IOException e){
      System.err.println("File error " + e.toString());
      System.exit(1);
    }
  }

  public int readEnvironment(){
    int results=0;
      if(map[x][y-1])
        results += 1;
      if(map[x+1][y-1])
        results += 2;
      if(map[x+1][y])
        results+=4;
        if(map[x+1][y+1])
          results +=8;
        if(map[x][y+1])
          results +=16;
        if(map[x-1][y+1])
          results +=32;
        if(map[x-1][y])
          results +=64;
        if(map[x-1][y-1])
          results +=128;
      return results;
    }

    public boolean fail(int inx, int iny){
       if (inx==0 || inx==(wide-1) || iny==0 || iny==(height-1) ){
           return true;
        }else{
            return false;
         }
     }
}
```

A.2.2 Simulation with evolved Edge/Obstacle rules

```
//   This program run with TestMain.java
//   To do the simulation
import java.util.Random;
import java.io.*;

public class SimERSP{

    final int wide=30;
    final int height=7;
    final int population=256;
    final int geneLength=768*2;
    final int afit=60;
    final int bfit=190;
    int x,y, door,generation=0;
    int[] fitness = new int[population];
    Random gen;
    boolean[][] gene= new boolean[population][geneLength];
    boolean[] bestGene = new boolean[geneLength];
    boolean[][] simmap = new boolean[wide][height];
```

```
boolean[][] obsmap = new boolean[wide][height];
boolean[][] enmap = new boolean[wide][height];
GA ga;
PrintWriter outFile, outGene;
public SimERSP(){
  gen = new Random();
  ga = new GA();
  initOthers();
  run();
}

public void initOthers(){
  for (int i =0; i<population; i++)
    fitness[i]=0;
  gene=ga.randominit(population, geneLength);
}

public void initMap(){
  for (int i=0; i < wide; i++){
    for (int j=0; j< height; j++){
      simmap[i][j]=false;
        obsmap[i][j]=false;
        enmap[i][j]=false;
    }
  }
         //init environemnt
  for (int i=0; i < wide; i++){
    enmap[i][0]=true;
    enmap[i][height-1]=true;
  }
  for (int i=0; i < height; i++){
    enmap[0][i]=true;
    enmap[wide-1][i]=true;
  }
  simmap[x][y]=true;
  //System.out.println("map clean");
}

public void run(){
  int address,tempfit,round=0, highest=0, environment;
  boolean flag;
  double avefit=0;
  while(avefit<afit || ga.getBest()<bfit){
  //evolve loop
    generation++;
```

```
System.out.print("Generation is " + generation + " ");
door = 26;
for (int i=0; i<population; i++){
// adding a loop here
  tempfit=0;
  for (int sx=1; sx<wide-1; sx++){
    for (int sy=1; sy<height-1; sy++){
      x=sx;
      y=sy;
      initMap();

      //run loop
      for (int j =0; j< 600; j++){
          //read flag
          flag=readFlag(x,y);
          if(flag){
            //get inforn from 8 directions
            environment=readOM();
            address=environment*3+geneLength/2;
          }else{// read SM map
            environment=readSM();
            address=environment*3;
          }//if-else of flag
          action(address, i);
          if (y==(height-2) &&
            (x == door || x==(door+1) || x==(door+2)) ){
            j=999;
            tempfit = tempfit +wide*height;
          }
      }// run loop
      //calculation fitness
      for (int ii=0; ii < wide; ii++){
        if (simmap[ii][1])
          tempfit++;
        if (simmap[ii][height-2])
          tempfit++;
        for (int j=0; j< height; j++){
          if (simmap[ii][j])
            tempfit++;
        }
      }
      for (int ii=0; ii<height; ii++){
        if (simmap[1][ii])
          tempfit++;
        if (simmap[wide-2][ii])
```

```
              tempfit++;
          }
          if (tempfit>highest){
             highest=tempfit;
          }
        }//sx and sy
      }
      fitness[i]=tempfit/(wide*height);
    }//population loop
    avefit=0;
    for (int i=0; i<population; i++)
    avefit+=fitness[i];
    avefit=avefit/population;
    gene = ga.sorting(gene, fitness, geneLength, population);
    //System.out.println("Generation: " + round
        + "  fitness: " + avefit + " Best: " + ga.getBest());
    System.out.println(avefit + " " + ga.getBest());
    for (int wi=0; wi< geneLength; wi++){
      bestGene[wi]=gene[0][wi];
    }
    round++;
    gene = ga.crossover(gene, population, geneLength);
    gene = ga.mutation(gene, population , geneLength, 0.1);
  }
  //evolve loop
  try{
    outGene = new PrintWriter(new FileWriter("Gene.txt"), true);
    for (int i=0; i< geneLength; i++){
      if (bestGene[i]){
        outGene.print(1);
      }else{
        outGene.print(0);
      }
    }
    outGene.println();
  }
  catch(IOException e){
    System.err.println("File error " + e.toString());
    System.exit(1);
  }
}

public int readOM(){
  int results=0;
    if(obsmap[x][y-1])
```

```
      results += 1;
    if(obsmap[x+1][y-1])
      results += 2;
    if(obsmap[x+1][y])
      results+=4;
    if(obsmap[x+1][y+1])
      results +=8;
    if(obsmap[x][y+1])
      results +=16;
    if(obsmap[x-1][y+1])
      results +=32;
    if(obsmap[x-1][y])
      results +=64;
    if(obsmap[x-1][y-1])
      results +=128;
  return results;
}

public int readSM(){
  int results=0;
    if(simmap[x][y-1])
      results += 1;
    if(simmap[x+1][y-1])
      results += 2;
    if(simmap[x+1][y])
      results+=4;
    if(simmap[x+1][y+1])
      results +=8;
    if(simmap[x][y+1])
      results +=16;
    if(simmap[x-1][y+1])
      results +=32;
    if(simmap[x-1][y])
      results +=64;
    if(simmap[x-1][y-1])
      results +=128;
  return results;
}

public boolean readFlag(int xx, int yy){
    return obsmap[xx-1][yy-1]|obsmap[xx][yy-1]|obsmap[xx+1][yy-1]|
           obsmap[xx-1][yy]|obsmap[xx+1][yy]|
           obsmap[xx-1][yy+1]|obsmap[xx][yy+1]|obsmap[xx+1][yy+1];
}
```

```
public boolean fail(int xx, int yy){
    return enmap[xx][yy];
}
public void action(int index, int ind){
  int lastX=x, lastY=y;
    if (gene[ind][index+2]){
        if (gene[ind][index+1]){
            if(gene[ind][index]){
                //case 111
                x=x-1;
                y=y+1;
            }else{
                //case 110
                y=y+1;
            }
        }else{
            if(gene[ind][index]){
                //case 101
                x=x-1;
            }else{
                //case 100
                x=x-1;
                y=y-1;
            }
        }
    }else{
        if (gene[ind][index+1]){
            if(gene[ind][index]){
                //case 011
                x=x+1;
            }else{
                //case 010
                x=x+1;
                y=y+1;
            }
        }else{
            if(gene[ind][index]){
                //case 001
                x=x+1;
                y=y-1;
            }else{
                //case 000
                y=y-1;
            }
        }
        }
        if(fail(x,y)){
            obsmap[x][y]=true;
            x=lastX;
            y=lastY;
        }else{
            if(!readFlag(x,y)){
                simmap[x][y]=true;
            }
        }
    }
}
```

A.3 Implementation Codes

A.3.1 Implementation with pre-set Edge/Obstacle rules

```
#include <iostream>
#include <evolution/Base.hpp>
#include <evolution/Task.hpp>
#include <evolution/Resource.hpp>
#include <evolution/tasks/TaskNavigation.hpp>
#include <evolution/tasks/TaskUtil.hpp>
#include <fstream>
#include <math.h>

using namespace Evolution;
using namespace std;

  const int geneLength=768;
  const int height=100;
  const int wide =100;
  const int step=30,dstep=28;

  bool gene[geneLength];
  bool runmap[wide][height];
  int x,y, sx=50, sy=50, px=0, py=0;
  ofstream logFile("log.txt");
  ofstream posFile("pos.txt");
  ofstream mapFile("runmap.txt");

int readgene(){
  ifstream infile("gene.txt", ios::in);
  if(!infile){
    std::cout << " open failed " << std::endl;
    return -1;
  }
  char ch;
  int index=0;
  while(infile.get(ch)){
    if(ch=='1'){
      gene[index]=1;
      index++;
    }else if (ch=='0'){
      gene[index]=0;
      index++;
    }
  }
}

void initmap(){
  for (int i=0; i < wide; i++){
    for (int j=0; j< height; j++)
      runmap[i][j]=0;
  }
  runmap[x][y]=1;
}

bool moveTo(int toX, int toY){
  bool value=0;

  if (! logFile) // Always test file open
    {
        cout << "Error opening output file track.txt" << endl;
        return 0;
    }
    // We want to use centimeter as the distance unit, degrees
    // as the angle unit, and seconds as the time unit.
    Units::set_default_units(UNIT_DISTANCE, "centimeter");
    Units::set_default_units(UNIT_ANGLE,    "degrees");
    Units::set_default_units(UNIT_TIME,     "seconds");

    // Load resource task library, which contains the
    // DriveMoveDelta task.
    TaskRegistry::load_library("libevotasknavigation");
  TaskRegistry::load_library("libevotaskutil");
  TaskRegistry::load_library("libevotaskresource");
///////////////////    Get positions
```

```
  TaskFunctor* get_pos = TaskRegistry::find_task("Evolution.GetPosition");
  TaskContext *c = TaskContext::task_args(0);
    TaskValuePtr result  (get_pos->run (c));

////////////////    Computer the heading and distance
  DoubleArray *da = result->get_double_array();
  double currentX=da->operator[](0);
  double currentY=da->operator[](1);
  logFile << "to position " << toX << " " <<toY << " now "
        << currentX << " " <<currentY <<endl;
  posFile << currentX << " " << currentY << endl;
  cout << "to position " << toX << " " <<toY << " now "
     << currentX << " " <<currentY <<endl;

  TaskContextPtr context(TaskContext::task_args(0));
    // Create a Parallel object using the context.
    Parallel parallel(context.get());

    TaskFunctor* moveto =
  TaskRegistry::find_task("Evolution.MoveTo");

  double p11[] = {toX, toY};
  DoubleArray  point1 = DoubleArray(p11, 2);
  Evolution::TaskArg args[] = { 0, point1 };
  Evolution::Task* task1 = parallel.add_task(moveto, 2, args);
  TaskFunctor* waitwait =
        TaskRegistry::find_task("Evolution.Wait");
  Evolution::TaskArg wait_args[] = { 10 };
  Task* task2 = parallel.add_task(waitwait, 1, wait_args);

  Task* completetask;
  parallel.wait_for_first_complete_task (&completetask);

  if(completetask==task1){
    value=1;
    cout << "move done" << endl;
    logFile << "move done" << endl;
  }else{
    value=0;
  }
  ///////////////////    Get positions
    TaskValuePtr resultpos2  (get_pos->run (c));

////////////////    Computer the heading and distance
  DoubleArray *db = resultpos2->get_double_array();
  currentX=db->operator[](0);
  currentY=db->operator[](1);
  double diff = abs(toX-currentX) + abs(toY-currentY);
  logFile<<"Stop at "<<currentX<<" "<<currentY<<" diff "<<diff<<endl;
  cout <<"Stop at "<<currentX<<" "<< currentY<<" diff "<<diff<<endl;
  if (value ==0 && diff<15){
    value=1;
  }
  if(!value){
    cout << "move failed make a turn" << endl;
    logFile << "move failed make a turn" << endl;
    TaskFunctor* irturn =
TaskRegistry::find_task("Evolution.RangeSensorSweep");
    Evolution::TaskArg argturn[] = { 25, 170, 20, "2IR1" };
    TaskContext *tt = TaskContext::task_args(4, argturn);
    TaskValuePtr result3  (irturn->run (tt));
  }
// Shut down the task manager prior to exit.
  return value;
}

bool moveSt(int dx, int dy){
  bool success=0;
  if (dx==0){
    success = moveTo(px, (py+dy));
    if(!success){
      success = moveTo(px,(py-dy));
      if(success){
        py=py-dy;
        y-=(dy/abs(dy));
      }else{
        cout << "failed to move !" << endl;
        logFile << "failed to move !" << endl;
      }
    }else{
      py=py+dy;
      y+=(dy/abs(dy));
    }
  }else{
    success = moveTo((px+dx), py);
    if(!success){
      success = moveTo((px-dx),py);
      if(success){
        px=px-dx;
        x-=(dx/abs(dx));
      }else{
        cout << "failed to move !" << endl;
        logFile << "failed to move !" << endl;
      }
    }else{
      px=px+dx;
      x+=(dx/abs(dx));
    }
  }
  return success;
}

bool moveDi(int dx, int dy){
  bool success=0;
  success = moveTo((px+dx),(py+dy));
  if(success){
    px+=dx;
    py+=dy;
    x+=(dx/abs(dx));
    y+=(dy/abs(dy));
  }else{
    success=moveTo((px-dx), (py-dy));
    if(success){
      px-=dx;
      py-=dy;
      x-=(dx/abs(dx));
      y-=(dy/abs(dy));
    }else{
      success=moveTo((px-dx),(py+dy));
      if(success){
        px-=dx;
        py+=dy;
        x-=(dx/abs(dx));
        y+=(dy/abs(dy));
      }else{
        success=moveTo((px+dx),(py-dy));
        if(success){
          px+=dx;
          py-=dy;
          x+=(dx/abs(dx));
          y-=(dy/abs(dy));
        }else{
          cout << "failed to move !" << endl;
          logFile << "failed to move Di!" << endl;
        }
      }
    }
  }
  return success;
}

void moveN(){
  moveSt(0, -step);
}
void moveNE(){
  moveDi(step,-step);
}
void moveE(){
  moveSt(step,0);
}
void moveSE(){
  moveDi(step,step);
}
void moveS(){
  moveSt(0, step);
}
void moveSW(){
  moveDi(-step, step);
}
void moveW(){
  moveSt(-step, 0);
}
void moveNW(){
  moveDi(-step,-step);
}

int readEnvironment(){
  int results=0;
  if(runmap[x][y-1])
    results += 1;
  if(runmap[x+1][y-1])
    results += 2;
  if(runmap[x+1][y])
    results+=4;
  if(runmap[x+1][y+1])
    results +=8;
  if(runmap[x][y+1])
    results +=16;
  if(runmap[x-1][y+1])
```

```
    results +=32;
  if(runmap[x-1][y])
    results +=64;
  if(runmap[x-1][y-1])
    results +=128;
  return results;
}

void run(){
  int address;
  x=sx;
  y=sy;
  initmap();

  for(int i=0; i<600; i++){
    int environment=readEnvironment();
    address = environment*3;
    logFile << environment << " "<< gene[address+2]
           << gene[address+1] << gene[address] << endl;
    if (gene[address+2]){
      if (gene[address+1]){
        if(gene[address]){
            //case 111
            moveSW();
        }else{
            //case 110
            moveS();
        }
      }else{
        if(gene[address]){
            //case 101
            moveW();
        }else{
            //case 100
            moveNW();
        }
      }
    }else{
      if (gene[address+1]){
        if(gene[address]){
            //case 011
            moveE();
        }else{
            //case 010
            moveSE();
        }
      }else{
        if(gene[address]){
            //case 001
            moveNE();
        }else{
            //case 000
            moveN();
        }
      }
    }
    mapFile << x << " " << y << endl;
    runmap[x][y]=1;
  }
}

int main(int argc, char **argv)
{
  readgene();
  Evolution::TaskManager* manager =
        Evolution::TaskManager::get_task_manager();
  run();
  TaskManager::get_task_manager()->shutdown();
  logFile.close();
  mapFile.close();
  return 0;
}
```

A.3.2 Implementation with evolved Edge/Obstacle rules

```
#include <iostream>
#include <evolution/Base.hpp>
#include <evolution/Task.hpp>
#include <evolution/Resource.hpp>
#include <evolution/tasks/TaskNavigation.hpp>
#include <evolution/tasks/TaskUtil.hpp>
#include <fstream>
#include <math.h>

using namespace Evolution;
using namespace std;
  const int geneLength=768*2;
  const int height=100;
  const int wide =100;
  const int step=30,dstep=28;
  bool gene[geneLength];
  bool runmap[wide][height];
  bool obsmap[wide][height];
  int x,y, sx=50, sy=50, px=0, py=0;
  ofstream logFile("log.txt");
  ofstream posFile("pos.txt");
  ofstream mapFile("runmap.txt");

int readgene(){
  ifstream infile("gene.txt", ios::in);
  if(!infile){
    std::cout << " open failed " << std::endl;
    return -1;
  }
  char ch;
  int index=0;
  while(infile.get(ch)){
    if(ch=='1'){
      gene[index]=1;
      index++;
    }else if (ch=='0'){
      gene[index]=0;
      index++;
    }
  }
  cout << gene[767] << endl;
}

void initmap(){
  for (int i=0; i < wide; i++){
      for (int j=0; j< height; j++){
        runmap[i][j]=0;
        obsmap[i][j]=0;
      }
  }
  runmap[x][y]=1;
}

bool moveTo(int toX, int toY){
  bool value=0;
  if (! logFile) // Always test file open
    {
        cout << "Error opening output file track.txt" << endl;
        return 0;
    }
    // We want to use inches as the distance unit, degrees
    // as the angle unit, and seconds as the time unit.
    Units::set_default_units(UNIT_DISTANCE, "centimeter");
    Units::set_default_units(UNIT_ANGLE,    "degrees");
    Units::set_default_units(UNIT_TIME,     "seconds");

    // Load resource task library, which contains the
    // DriveMoveDelta task.
    TaskRegistry::load_library("libevotasknavigation");
  TaskRegistry::load_library("libevotaskutil");
  TaskRegistry::load_library("libevotaskresource");
///////////////////    Get positions
  TaskFunctor* get_pos = TaskRegistry::find_task("Evolution.GetPosition");
  TaskContext *c = TaskContext::task_args(0);
    TaskValuePtr result  (get_pos->run (c));

/////////////////    Computer the heading and distance
  DoubleArray *da = result->get_double_array();
  double currentX=da->operator[](0);
  double currentY=da->operator[](1);
  logFile << "to position " << toX << " " <<toY
      << " now " << currentX << " " <<currentY <<endl;
  posFile << currentX << " " << currentY << endl;
  cout << "to position " << toX << " " <<toY
      << " now " << currentX << " " <<currentY <<endl;

  TaskContextPtr context(TaskContext::task_args(0));
    // Create a Parallel object using the context.
    Parallel parallel(context.get());

    // The first task is to drive forward 20 inches.  We
    // will do this using the DriveMoveDelta task.
    TaskFunctor* moveto =
        TaskRegistry::find_task("Evolution.MoveTo");
  double p11[] = {toX, toY,15.0, 30.0, 10.0 };
  DoubleArray  point1 = DoubleArray(p11, 5);
  Evolution::TaskArg args[] = { 0, point1 };
  Evolution::Task* task1 = parallel.add_task(moveto, 2, args);
  TaskFunctor* waitwait =
        TaskRegistry::find_task("Evolution.Wait");
  Evolution::TaskArg wait_args[] = { 7 };
  Task* task2 = parallel.add_task(waitwait, 1, wait_args);
  Task* completetask;
  parallel.wait_for_first_complete_task (&completetask);

  if(completetask==task1){
    value=1;
    cout << "move done" << endl;
    logFile << "move done" << endl;
  }else{
    value=0;
  }
    TaskValuePtr resultpos2  (get_pos->run (c));
/////////////////    Computer the heading and distance
  DoubleArray *db = resultpos2->get_double_array();
  currentX=db->operator[](0);
  currentY=db->operator[](1);
  double diff = abs(toX-currentX) + abs(toY-currentY);
  logFile <<"Stop at "<<currentX<<" "<<currentY<<" diff "<<diff<<endl;
  cout<<"Stop at "<<currentX<<" "<<currentY<<" diff "<<diff<<endl;
  if (value ==0 && diff<15){
    value=1;
  }
  if(!value){
    cout << "move failed make a turn" << endl;
    logFile << "move failed make a turn" << endl;
    TaskFunctor* irturn
= TaskRegistry::find_task("Evolution.RangeSensorSweep");
    Evolution::TaskArg argturn[] = { 25, 170, 20, "2IR1" };
    TaskContext *tt = TaskContext::task_args(4, argturn);
    TaskValuePtr result3  (irturn->run (tt));
  }
// Shut down the task manager prior to exit.
  return value;
}

bool moveDi(int dx, int dy){
  bool success=0;
  success = moveTo((px+dx),(py+dy));
  if(success){
    px+=dx;
    py+=dy;
  }
  return success;
}

bool moveN(){
  y-=1;
  return moveDi(0, -step);
}
bool moveNE(){
  x+=1;
  y-=1;
  return moveDi(step,-step);
}
bool moveE(){
  x+=1;
  return moveDi(step,0);
}
bool moveSE(){
  x+=1;
  y+=1;
  return moveDi(step,step);
}
bool moveS(){
  y+=1;
  return moveDi(0, step);
}
bool moveSW(){
  x-=1;
  y+=1;
  return moveDi(-step, step);
}
bool moveW(){
  x-=1;
  return moveDi(-step, 0);
}
bool moveNW(){
  x-=1;
  y-=1;
  return moveDi(-step,-step);
}

int readRM(){
  int results=0;
  if(runmap[x][y-1])
    results += 1;
  if(runmap[x+1][y-1])
    results += 2;
  if(runmap[x+1][y])
    results+=4;
  if(runmap[x+1][y+1])
    results +=8;
  if(runmap[x][y+1])
    results +=16;
  if(runmap[x-1][y+1])
    results +=32;
  if(runmap[x-1][y])
    results +=64;
  if(runmap[x-1][y-1])
    results +=128;
  return results;
}

int readOM(){
  int results=0;
  if(obsmap[x][y-1])
    results += 1;
  if(obsmap[x+1][y-1])
    results += 2;
  if(obsmap[x+1][y])
    results+=4;
  if(obsmap[x+1][y+1])
    results +=8;
  if(obsmap[x][y+1])
    results +=16;
  if(obsmap[x-1][y+1])
    results +=32;
  if(obsmap[x-1][y])
    results +=64;
  if(obsmap[x-1][y-1])
    results +=128;
  return results;
}

bool readFlag(int xx, int yy){
    return obsmap[xx-1][yy-1]|obsmap[xx][yy-1]|obsmap[xx+1][yy-1]|
        obsmap[xx-1][yy]|obsmap[xx+1][yy]|
        obsmap[xx-1][yy+1]|obsmap[xx][yy+1]|obsmap[xx+1][yy+1];
}

void action(int index){
  bool success=0;
  int lastX=x, lastY=y;
  if (gene[index+2]){
      if (gene[index+1]){
        if(gene[index]){
            //case 111
```

```
                success=moveSW();
            }else{
                //case 110
                success=moveS();
            }
          }else{
            if(gene[index]){
                //case 101
                success=moveW();
            }else{
                //case 100
                success=moveNW();
            }
          }
        }else{
          if (gene[index+1]){
            if(gene[index]){
                 //case 011
                 success=moveE();
            }else{
                 //case 010
                 success=moveSE();
            }
          }else{
            if(gene[index]){
                 //case 001
                 success=moveNE();
            }else{
                 //case 000
                 success=moveN();
            }
          }
        }
        if(success){
          if(!readFlag(x,y))
               runmap[x][y]=1;
        }else{
              obsmap[x][y]=1;
              x=lastX;
              y=lastY;
        }
    }

    void run(){
      int address, environment;
```

```
  bool flag;
  x=sx;
  y=sy;
  initmap();
  for(int i=0; i<1000; i++){
    flag=readFlag(x,y);
    //cout << "movement " << i << endl;
     if(flag){
         environment=readOM();
         address=environment*3+geneLength/2;
         logFile << environment << " OM "<< gene[address+2]
             << gene[address+1] << gene[address] << endl;
     }else{// read SM map
        environment=readRM();
        address=environment*3;
        logFile << environment << " RM "<< gene[address+2]
            << gene[address+1] << gene[address] << endl;
    }//if-else of flag
    //cout << " to action" << endl;
    action(address);
    //cout << " action done" << endl;
    mapFile << x << " " << y << endl;
    //runmap[x][y]=1;
  }
}

int main(int argc, char **argv)
{
  readgene();
  Evolution::TaskManager* manager =
    Evolution::TaskManager::get_task_manager ();
  run();
  TaskManager::get_task_manager()->shutdown();
  logFile.close();
  mapFile.close();
  return 0;
}
```

Appendix B

Source code of improved design

B.1 Simulation

```
//   This program run with TestMain.java   the rules of this one is man-made
//   To do the simulation
import java.util.Random;
import java.io.*;

public class SimERSP{

    final int wide=30;
    final int height=10;
    final int population=1024;
    final int highPopulation = population*2;
    final int geneLength=210;
    double mutaterate=0.01;
        final int timelimit=10000;
        final int updatelimit=20;
    int x,y, oldshift=0,currentshift=0, generation=0;
    int[] fitness = new int[highPopulation];
    boolean[][] gene= new boolean[highPopulation][geneLength];
    int[] reduce = new int[256];
    int[] shift = new int[256];
    boolean[] bestGene = new boolean[geneLength];
    boolean[][] map = new boolean[wide][height];
    boolean[][] bestmap = new boolean[wide][height];
    long startTime=0, currentTime=0,mytime=0;
    GA ga;
    BufferedReader reduceFile, shiftFile;
    PrintWriter outFile, outGene;

    public SimERSP(){
      //gen = new Random();
      ga = new GA();
      initOthers();
      run();
    }
////////////////////////////////////////////////////////////////////////
    public void initOthers(){
      for (int i =0; i<highPopulation; i++)
        fitness[i]=0;
      gene=ga.randominit(highPopulation, geneLength);
      startTime=System.currentTimeMillis();
      try{
        reduceFile = new BufferedReader(new FileReader("reduce.txt"));
        String line;
        String[] temp;
        while ((line = reduceFile.readLine() ) !=null){
          temp = line.split(" ");
          int index = Integer.parseInt(temp[0]);
          int content = Integer.parseInt(temp[1]);
          reduce[index]=content;
        }
        shiftFile = new BufferedReader(new FileReader("shift.txt"));

        while ((line = shiftFile.readLine()) !=null){
          temp = line.split(" ");
          int index = Integer.parseInt(temp[0]);
          int content = Integer.parseInt(temp[1]);
          shift[index]=content;
        }
      }catch(IOException e){
        System.err.println("Error in file");
      }
    //  for(int i =0; i<256; i++){
    //    System.out.println(i + " " + shift[i]);
    //  }
    }
    //////////////////////////////////////////////////
    public void initMap(){
      for (int i=0; i < wide; i++){
        for (int j=0; j< height; j++){
          map[i][j]=false;
                }
      }
      map[x][y]=true;
      //System.out.println("map clean");
    }
  ///////////////////////////////////////////////////////////////
    public void run(){
```

```
int address,round=0, highest=0, action, noupdate=0;
int tempfit=0;
double avefit=0;

try{
  outFile = new PrintWriter(new FileWriter("log.txt"), true);
while(mytime < timelimit){
  generation++;
  currentTime=System.currentTimeMillis();
  mytime=(currentTime-startTime)/1000;
  if(generation%2 ==0){
    System.out.print("Generation is " + generation
+ " Time " + mytime + " ");
outFile.print("Generation "+generation+" Time "+mytime+" ");
  }
  for (int i=0; i<highPopulation; i++){// for each individual
    tempfit=0;
        x=15;
        y=5;
        initMap();
        map[x][y]=true;
        //run loop
        noupdate=0;
        for (int j =0; j< 600; j++){
          //get inforn from 8 directions
          action=readEnvironment(i);
          if (oldshift !=currentshift){
            tempfit=tempfit-1;
          }
          if ((action%2)==1){
              tempfit=tempfit-1;
          }
          switch(action){
             case 0:  //north
               y=y+1;
               if(fail(x,y))
                 y=y-2;
               break;
             case 1: //northeast
               y=y+1;
               x=x+1;
               if (fail(x,y)){
                 x=x-2;
                 y=y-2;
                 if (fail(x,y)){
```

```
     x=x+2;
     if (fail(x,y)){
       x=x-2;
       y=y+2;
       if (fail(x,y)){
        System.out.println("Error at northeast");
            break;
       }
      }
     }
  }
  break;
case 2://East
   x=x+1;
   if (fail(x,y))
      x=x-2;
   break;
case 3:  //Southeast
   x=x+1;
   y=y-1;
   if (fail(x,y)){
     x=x-2;
     y=y+2;
     if(fail(x,y)){
        y=y-2;
        if(fail(x,y)){
          x=x+2;
          y=y+2;
          if (fail(x,y)){
            System.out.println("Error at southeast");
               break;
          }
        }
       }
     }
     break;
  case 4:  //south
      y=y-1;
  if (fail(x,y))
     y=y+2;
     break;
   case 5: //southwest
     y=y-1;
     x=x-1;
   if (fail(x,y)){
     x=x+2;
     y=y+2;
     if (fail(x,y)){
         x=x-2;
         if (fail(x,y)){
            x=x+2;
            y=y-2;
         if (fail(x,y)){
             System.out.println("Error at southwest");
                  break;
             }
         }
       }
     }
   break;
 case 6: //west
     x=x-1;
     if(fail(x,y))
       x=x+2;
       break;
case 7: //northwest
     x=x-1;
     y=y+1;
     if (fail(x,y)){
       x=x+2;
       y=y-2;
       if(fail(x,y)){
          y=y+2;
          if (fail(x,y)){
            x=x-2;
            y=y-2;
            if (fail(x,y)){
              System.out.println("Error at northwest");
              break;
             }
            }
          }
       }
       break;
   }//switch
 if(map[x][y]){
    tempfit=tempfit-1;
    noupdate++;
 }else{
    tempfit=tempfit+10;
```

```
                        map[x][y]=true;
                        noupdate=0;
                     }
                     if (noupdate>updatelimit){
                        j=600;
                     }
                }      // run loop
                fitness[i]=tempfit;
        }//population loop
        gene=ga.sorting(gene,fitness,geneLength,highPopulation,population);
        avefit=0;
        for (int i=0; i<population; i++){
          avefit+=fitness[i];
        }
        avefit=avefit/population;
        int[] somevalues = ga.getSome();
        if(generation%2 ==0){
          System.out.println( somevalues[0] + "\t" + somevalues[1] + " "
+ somevalues[2] + " "+somevalues[3]+" "+somevalues[4]+"\t" + avefit);
          outFile.println( somevalues[0] + "\t" + somevalues[1] + " "
+somevalues[2]+" "+somevalues[3]+" "+somevalues[4]+"\t"+avefit);
        }
        for (int wi=0; wi< geneLength; wi++){
          bestGene[wi]=gene[0][wi];
        }
        round++;
        gene = ga.crossover(gene, population, highPopulation, geneLength);
        gene = ga.mutation(gene,  highPopulation, geneLength, mutaterate);
      }
      //evolve loop
        outGene = new PrintWriter(new FileWriter("Gene.txt"), true);
        for (int i=0; i< geneLength; i++){
          if (bestGene[i]){
            outGene.print(1);
          }else{
            outGene.print(0);
          }
        }
        outGene.println();
      }
      catch(IOException e){
        System.err.println("File error " + e.toString());
        System.exit(1);
      }
    }
    public int readEnvironment(int individual){
      int results=0;
        oldshift = currentshift;
     if(map[x][y+1])
          results += 1;
        if(map[x+1][y+1])
          results += 2;
        if(map[x+1][y])
          results+=4;
        if(map[x+1][y-1])
          results +=8;
        if(map[x][y-1])
          results +=16;
        if(map[x-1][y-1])
          results +=32;
        if(map[x-1][y])
          results +=64;
        if(map[x-1][y+1])
          results +=128;
        int reducevalue = reduce[results];
        int shiftvalue = shift[results];
        boolean action1 = gene[individual][reducevalue*3];
        boolean action2 = gene[individual][reducevalue*3+1];
        boolean action3 = gene[individual][reducevalue*3+2];
        int action=0;
        if (action3)
          action+=4;
        if (action2)
          action+=2;
        if (action1)
          action+=1;
        action+=(2*shiftvalue);
        action=action%8;
        currentshift = action;
      return action;
    }
    public boolean fail(int inx, int iny){
          if (inx==0 || inx==(wide-1) || iny==0 || iny==(height-1) ){
            return true;
          }else{
            return false;
         }
    }
}
```

B.2 Implementation

```
#include <iostream>
#include <evolution/Base.hpp>
#include <evolution/Task.hpp>
#include <evolution/Resource.hpp>
#include <evolution/tasks/TaskNavigation.hpp>
#include <evolution/tasks/TaskUtil.hpp>
#include <fstream>
#include <math.h>
#include <stdlib.h>

using namespace Evolution;
using namespace std;

  const int geneLength=210;
  const int height=100;
  const int wide =100;
  const int step=30,dstep=28;
  bool gene[geneLength];
  int reduce[256];
  int shift[256];
  bool runmap[wide][height];
  int x,y, sx=50, sy=50, px=0, py=0, currentshift=0, oldshift=0;;
  ofstream logFile("log.txt");
  ofstream posFile("pos.txt");
  ofstream mapFile("runmap.txt");

int readgene(){
  ifstream infile("gene.txt", ios::in);
  if(!infile){
    std::cout << " open failed " << std::endl;
    return -1;
  }
  char ch;
  int index=0;
  while(infile.get(ch)){
    if(ch=='1'){
      gene[index]=1;
      index++;
    }else if (ch=='0'){
      gene[index]=0;
      index++;
    }
  }
  ifstream inreduce("reduce.txt", ios::in);
  if(!inreduce){
    std::cout << " open failed " << std::endl;
    return -1;
  }
  string buf;
  int counter=0;
  while(!inreduce.eof()){
    getline(inreduce, buf);
    const char *chars = buf.c_str();
    reduce[counter]=atoi(chars);
    counter++;
  }
  ifstream inshift("shift.txt", ios::in);
  if(!inshift){
    std::cout << " open failed " << std::endl;
    return -1;
  }
  counter=0;
  while(!inshift.eof()){
    getline(inshift, buf);
    const char *chars = buf.c_str();
    shift[counter]=atoi(chars);
    counter++;
  }
}
void initmap(){
  for (int i=0; i < wide; i++){
    for (int j=0; j< height; j++){
      runmap[i][j]=0;
    }
  }
  runmap[x][y]=1;
}

bool moveTo(int toX, int toY){
  bool value=0;

  if (! logFile) // Always test file open
    {
        cout << "Error opening output file track.txt" << endl;
        return 0;
    }
    // We want to use centimeter as the distance unit, degrees
    // as the angle unit, and seconds as the time unit.
    Units::set_default_units(UNIT_DISTANCE, "centimeter");
    Units::set_default_units(UNIT_ANGLE,    "degrees");
    Units::set_default_units(UNIT_TIME,     "seconds");
    // Load resource task library, which contains the
    // DriveMoveDelta task.
    TaskRegistry::load_library("libevotasknavigation");
  TaskRegistry::load_library("libevotaskutil");
  TaskRegistry::load_library("libevotaskresource");
///////////////////    Get positions
  TaskFunctor* get_pos
= TaskRegistry::find_task("Evolution.GetPosition");
  TaskContext *c = TaskContext::task_args(0);
    TaskValuePtr result  (get_pos->run (c));

/////////////////    Computer the heading and distance
  DoubleArray *da = result->get_double_array();
  double currentX=da->operator[](0);
  double currentY=da->operator[](1);
  logFile << "to position " << toX << " " <<toY <<
    " now " << currentX << " " <<currentY <<endl;
  posFile << currentX << " " << currentY << endl;
  cout << "to position " << toX << " " <<toY << " now "
    << currentX << " " <<currentY <<endl;
  TaskContextPtr context(TaskContext::task_args(0));
    // Create a Parallel object using the context.
    Parallel parallel(context.get());
    TaskFunctor* moveto =
        TaskRegistry::find_task("Evolution.MoveTo");

  double p11[] = {toX, toY};
  DoubleArray  point1 = DoubleArray(p11, 2);
  Evolution::TaskArg args[] = { 0, point1 };
  Evolution::Task* task1 = parallel.add_task(moveto, 2, args);
  TaskFunctor* waitwait =
        TaskRegistry::find_task("Evolution.Wait");
  Evolution::TaskArg wait_args[] = {6};
  Task* task2 = parallel.add_task(waitwait, 1, wait_args);
  Task* completetask;
  parallel.wait_for_first_complete_task (&completetask);

  if(completetask==task1){
    value=1;
    cout << "move done" << endl;
    logFile << "move done" << endl;
  }else if(completetask==task2){
    value=0;
  }else{
    value=1;
    cout << "move done 3" << endl;
    logFile << "move done 3" << endl;
  }
    TaskValuePtr resultpos2  (get_pos->run (c));

/////////////////    Computer the heading and distance
  DoubleArray *db = resultpos2->get_double_array();
  currentX=db->operator[](0);
  currentY=db->operator[](1);
  double diff = abs(toX-currentX) + abs(toY-currentY);
  logFile << "Stop at " << currentX << " " << currentY
    << " diff " << diff  <<endl;
  cout << "Stop at " << currentX << " " << currentY
    << " diff " << diff  <<endl;
  if (value ==0 && diff<15){
    value=1;
  }
  if(!value){
    cout << "move failed make a turn" << endl;
    logFile << "move failed make a turn" << endl;
    TaskFunctor* irturn
= TaskRegistry::find_task("Evolution.RangeSensorSweep");
    Evolution::TaskArg argturn[] = { 25, 170, 20, "2IR1" };
    TaskContext *tt = TaskContext::task_args(4, argturn);
    TaskValuePtr result3  (irturn->run (tt));
  }
// Shut down the task manager prior to exit.
  return value;
}
bool moveSt(int dx, int dy){
  bool success=0;
  if (dx==0){
    success = moveTo(px, (py+dy));
    if(!success){
      success = moveTo(px,(py-dy));
      if(success){
        py=py-dy;
        y-=(dy/abs(dy));
      }else{
        cout << "failed to move !" << endl;
        logFile << "failed to move !" << endl;
      }
    }else{
      py=py+dy;
      y+=(dy/abs(dy));
    }
  }else{
    success = moveTo((px+dx), py);
    if(!success){
      success = moveTo((px-dx),py);
      if(success){
        px=px-dx;
        x-=(dx/abs(dx));
      }else{
        cout << "failed to move !" << endl;
        logFile << "failed to move !" << endl;
      }
    }else{
      px=px+dx;
      x+=(dx/abs(dx));
    }
  }
  return success;
}
bool moveDi(int dx, int dy){
  bool success=0;
  success = moveTo((px+dx),(py+dy));
  if(success){
    px+=dx;
    py+=dy;
    x+=(dx/abs(dx));
    y+=(dy/abs(dy));
  }else{
    success=moveTo((px-dx), (py-dy));
    if(success){
      px-=dx;
      py-=dy;
      x-=(dx/abs(dx));
      y-=(dy/abs(dy));
    }else{
      success=moveTo((px-dx),(py+dy));
      if(success){
        px-=dx;
        py+=dy;
        x-=(dx/abs(dx));
        y+=(dy/abs(dy));
      }else{
        success=moveTo((px+dx),(py-dy));
        if(success){
          px+=dx;
          py-=dy;
          x+=(dx/abs(dx));
          y-=(dy/abs(dy));
        }else{
          cout << "failed to move !" << endl;
          logFile << "failed to move Di!" << endl;
        }
      }
    }
  }
  return success;
}
void moveN(){
  moveSt(0, step);
}
void moveNE(){
  moveDi(step,step);
}
void moveE(){
  moveSt(step,0);
}
void moveSE(){
  moveDi(step,-step);
}
void moveS(){
  moveSt(0, -step);
}
void moveSW(){
  moveDi(-step, -step);
}
void moveW(){
  moveSt(-step, 0);
}
void moveNW(){
  moveDi(-step,step);
}

int readEnvironment(){
  int results=0;
  if(runmap[x][y+1])
    results += 1;
  if(runmap[x+1][y+1])
    results += 2;
  if(runmap[x+1][y])
    results+=4;
  if(runmap[x+1][y-1])
    results +=8;
  if(runmap[x][y-1])
    results +=16;
  if(runmap[x-1][y-1])
    results +=32;
  if(runmap[x-1][y])
    results +=64;
  if(runmap[x-1][y+1])
    results +=128;
  int reducevalue = reduce[results];
  int shiftvalue = shift[results];
  bool action1 = gene[reducevalue*3];
  bool action2 = gene[reducevalue*3+1];
  bool action3 = gene[reducevalue*3+2];
  int action=0;
  if (action3)
    action+=4;
  if (action2)
    action+=2;
  if (action1)
    action+=1;
  action+=(2*shiftvalue);
  action = action%8;
    currentshift = action;
  return action;
}

void run(){
  int address;
  x=sx;
  y=sy;
  initmap();
  for(int i=0; i<1000; i++){
    int address=readEnvironment();
    logFile << "envior "<<address  << endl;
    switch(address){
      case 0:  //north
              moveN();
              break;
          case 1: //northeast
```

```
              moveNE();
                break;
            case 2://East
                moveE();
                break;
            case 3:  //Southeast
                moveSE();
                break;
            case 4:  //south
                moveS();
                break;
            case 5: //southwest
                moveSW();
                break;
            case 6: //west
                moveW();
                break;
            case 7: //northwest
                moveNW();
                break;
    }
    mapFile << x << " " << y << endl;
    runmap[x][y]=1;
  }
}

int main(int argc, char **argv)
{
  readgene();
  Evolution::TaskManager* manager
= Evolution::TaskManager::get_task_manager ();
  run();
  TaskManager::get_task_manager()->shutdown();
  logFile.close();
  mapFile.close();
    return 0;
}
```