Intelligent Automated Guided Vehicle (AGV) with Genetic Algorithm Decision Making Capabilities

Hendrik Gideon Lubbe

Dissertation for the degree of

Magister Technologiae Engineering: Electrical

at the

School of Electrical and Computer Systems Engineering

Faculty of Engineering, Information and Communication Technology

at the

Central University of Technology, Free State

January 2007

Supervisor:	Mr. BJ Kotze, M. Tech. (Eng)
Co-Supervisor:	Prof. F. Aghdasi Ph.D.

Expressions of thanks

Firstly, I want to thank my Heavenly Father for giving me this talent and the people that have supported me. Secondly, thanks to my wife for her motivation. There were a lot of individuals at the Central University of Technology that believed and supported me but special thanks to Ben Kotze who helped me through the years.

I want to take this opportunity to say that I consider evolution nothing more than a theory.

Acknowledgements

I, Hendrik Gideon Lubbe, hereby declare that this research project submitted for the degree MAGISTER TECHNOLOGIAE ENGINEERING: ELECTRICAL, is my own independent work that has not been submitted before to any institution by me or to my knowledge by anyone else as part of any qualification.

....	....
SIGNATURE OF STUDENT	**DATE**

Opsomming

Die algehele doel van die eksperiment is om 'n intelligente masjien te maak wat kan leer uit ondervinding, dus ‘n nuwe metode moes ontwerp word. Dit is moontlik gemaak, deur 'n program te ontwikkel wat 'n ander program genereer. Deur die gegenereerde program konstant te verander om dit te verbeter, word 'n masjien die vermoë gegee om by sy omgewing aan te pas en dus te leer uit ondervinding.

Die gegenereerde program moes 'n spesifieke taak verrig. In die eksperiment is 'n program gegenereer vir 'n gesimuleerde PIC mikrobeheerder aanboord 'n gesimuleerde robot. Die doel was dat die robot so na as moonlik aan 'n spesifieke punt binne in 'n doolhof moes kom. Alhoewel slegs die afstand tussen die laaste posisie van die robot en die eindbestemming gebruik word as indikasie van hoe goed die robot opgetree het, moes die robot nogtans die vermoë toon om hindernisse te oorkom.

Die program het eksperimente uitgevoer deur na willekeur 'n paar instruksies in die huidige gegenereerde program te verander. Die gegenereerde program is dan geëvalueer deur die reaksies van die robot te simuleer. As die verandering aan die gegenereerde program veroorsaak dat die robot nader aan die eindbestemming kom, word die veranderde gegenereerde program gehou om later gebruik te word. As die verandering 'n minder gewenste resultaat lewer, word die veranderde gegenereerde program verwyder en die oorspronklike onveranderde gegenereerde program gehou vir toekomstige gebruik. Die proses is 'n honderd-duisend keer herhaal voordat die gegenereerde program as 'n resultaat beskou is.

Omdat 'n geringe kans bestaan het dat die instruksie wat gekies is, voordelig sou wees vir die program, was dit nodig om baie veranderinge te maak voordat die ideale instruksie en dus die ideale resultaat verkry kon word. Na elke verandering moes die program geëvalueer word deur die robot se reaksies te simuleer.

Die aantal veranderinge wat nodig was, kon baie verminder word deur instruksies wat moontlik voordelig is 'n hoër kans te gee om gekies te word as moontlik minder voordelige instruksies.

Omdat die Ewekansige-funksie so baie gebruik word in die eksperiment, het die resultate baie van mekaar verskil. Die probleem is oorkom deur die eksperimente baie te herhaal. 'n Enkele program is gegenereer deur 'n honderd-duisend keer die instruksies te verander.

Die nuwe metode is vergelyk met Genetiese Algoritmes, waar Genetiese Algoritmes gebruik was om programme te genereer deur gebruik te maak van dieselfde bronne as vir die nuwe metode. Die nuwe metode het gemaak dat die robot baie vinniger aanpas by sy omgewing as wat Genetiese Algoritmes kon.

'n Fisiese robot, ooreenstemmend met die gesimuleerde robot, is gebou om te bewys dat die gegenereerde programme op 'n fisiese robot sal werk.

Daar was baie verskille tussen die gegenereerde programme en die normale manier waarop 'n mens 'n program sou skryf. Vir diè rede, gee die resultate nie net nuwe maniere waarop 'n program geskryf kan word nie, maar kan moontlik programme ontwikkel wat voorheen nie deur menslike programmeerders gedoen kon word nie.

Abstract

The ultimate goal regarding this research was to make an intelligent learning machine, thus a new method had to be developed. This was to be made possible by creating a programme that generates another programme. By constantly changing the generated programme to improve itself, the machines are given the ability to adapt to there surroundings and, thus, learn from experience.

This generated programme had to perform a specific task. For this experiment the programme was generated for a simulated PIC microcontroller aboard a simulated robot. The goal was to get the robot as close to a specific position inside a simulated maze as possible. The robot therefore had to show the ability to avoid obstacles, although only the distance to the destination was given as an indication of how well the generated programme was performing.

The programme performed experiments by randomly changing a number of instructions in the current generated programme. The generated programme was evaluated by simulating the reactions of the robot. If the change to the generated programme resulted in getting the robot closer to the destination, then the changed generated programme was kept for future use. If the change resulted in a less desired reaction, then the newly generated programme was removed and the unchanged programme was kept for future use. This process was repeated for a total of one hundred thousand times before the generated program was considered valid.

Because there was a very slim chance that the instruction chosen will be advantageous to the programme, it will take many changes to get the desired instruction and, thus, the desired result. After each change an evaluation was made through simulation. The amount of necessary changes to the programme is greatly reduced by giving seemingly desirable instructions a higher chance of being chosen than the other seemingly unsatisfactory instructions.

Due to the extensive use of the random function in this experiment, the results differ from one another. To overcome this barrier, many individual programmes had to be generated by simulating and changing an instruction in the generated programme a hundred thousand times.

This method was compared against Genetic Algorithms, which were used to generate a programme for the same simulated robot. The new method made the robot adapt much faster to its surroundings than the Genetic Algorithms.

A physical robot, similar to the virtual one, was build to prove that the programmes generated could be used on a physical robot.

There were quite a number of differences between the generated programmes and the way in which a human would generally construct the programme. Therefore, this method not only gives programmers a new perspective, but could also possibly do what human programmers have not been able to achieve in the past.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMME SEGMENTS

1 Introduction

For thousands of years, man has attempted to imitate the human body by using mechanisms. One example of this is the mechanical arms the ancient Egyptians used to attach to their gods. The Greeks and eighteenth century Europeans were known to have made moving puppets and statues [1, p.2]. This makes it apparent that the idea of a robot, the word meaning "compulsive labour" has been around for quite some time [60, p.10].

Maybe the two most interesting phenomena found in nature are intelligence and the ability of organisms to adapt to their surroundings.

In the seventies, robots revolutionised many industries. More recently, the use of computers has made a significant difference in many fields. Numerically-controlled milling machines are not seen as robots, the distinction is in the complexity of the programme [10, p.3]. The term *robotic* has been used whenever a device changes its actions in response to an instruction or a change in its environment [5, p.1, 4].

The goal of a robot, with the ability to learn, is to let the robot be able to deal with unforeseen circumstances, and incomplete information [16]. In the end the robot must be able to do more than what it was programmed to do [9]. Up until 1992 only a small percentage of automation systems could learn through experience, thus there is space for improvement [14, p.19]. A current autonomous mobile robot's ability to avoid obstacles is much worse than a human's ability to steer a vehicle when it comes to speed, safety and complexity [15]. It is possible that robots will become as important to us as computers are today [6, p.1].

With this technology it may be possible to program a machine so that it can find new ways of doing what it is intended to do. These may be things that a person has not even thought of or considered possible. It also makes it possible for a machine's "mind" to adapt to different situations.

In this dissertation alternative methods to Genetic Algorithms, as well as a new alternative method for generating a programme, will be discussed, analysed and evaluated against Genetic Algorithms [8]. It might also be applied for the improvement in other types of robots or even just computers.

For this experiment, the final result must be an Automated Guided Vehicle (AGV) that can change its programme to adapt to its surroundings [4]. Because the AGV has to learn, it does not have to be able to do anything when it is first started up. At the end the AGV must be able to move from one specified point to another. Between these two points there may be obstacles and the robot must be able to avoid them. The AGV must also be able to follow instructions from a human operator.

This dissertation has two hypotheses. Firstly, determine if it is possible to generate a programme with an algorithm and give the features of the generated programme. Secondly, it should prove how effective Single-Chromosome-Evolution-Algorithms are compared to standard Genetic-Algorithms when used to generate a programme.

1.1 Hypothetical Solution

A simulation is made for the controller of an AGV. The AGV can then learn to adapt to a certain kind of surroundings before it is built in full.

The programme for the “Simulated-Functions-of-the-Microcontroller”, which acts as the controller for a virtual AGV, will be generated using two methods. These two methods are incorporated to evaluate the performance of the two methods against each other. The first is to use Genetic Algorithms and the second is to use “Single-Chromosome-Evolution-Algorithm”. Single-Chromosome-Evolution-Algorithms are explained in chapter four.

The position of the destination is altered in accordance with an adjustment to an artificial input, thus “teaching” the AGV to go to a specified destination for a specified input instruction.

The linear distance between the current position of the AGV and the destination functions as the current fitness of the Generated Programme.

It is possible that, in the past, people have focused too heavily on how genes and cells work. In this dissertation, the emphasis will be more on the mating habits and breeding methods of bigger organisms, with the aim of incorporating that into Genetic Algorithms.

The desired outcome is to have a real AGV, which is controlled by a simulated controller. This controller will have the same instruction set as a PIC micro-controller. At first the AGV will be a Simulated AGV in a simulated world. Single-Chromosome-Evolution-Algorithms, or Genetic-Algorithms, will generate the programme for the simulated controller, which drives the Simulated AGV. Results will be compared to evaluate which method performs the best.

If results are favourable, the Generated Programme will be tested on the Physical AGV. Using the simulation of the AGV to determine the programme is similar to thinking about what can be accomplished in a situation. Implementing the programme with the use of a Physical AGV resembles doing something after thinking about it.

An umbilical cord will connect the computer to the Physical AGV so that the simulated controller can still do the "thinking".

1.2 Layout of Dissertation

Chapter two gives the basic layout of a robot. Examples of different robots and components of a robot are given, with the focus being placed on the controller of a robot. Artificial Intelligence techniques could be in the form of a controller or they could act on the controller in some way. In section 2.7.5, Genetic Algorithms, which are Artificial Intelligence techniques, are explained.

Chapter three was included to indicate how genetics work in nature. The emphasis was on how multi-cellular organisms, such as chimpanzees, act; what the result of this action is and possible explanations for the outcome. This information is used as an explanation for possible reasons as to why Genetic Algorithms act the way they do. Some of these methods can also possibly be used in future experiments.

In chapter four, a new programme generating method, called Single-Chromosome-Evolution-Algorithms, is developed. Being the alternative to Genetic Algorithms, its ability to perform lay in its simplicity. Like Genetic Algorithms, Single-Chromosome-Evolution-Algorithms used mutation and fitness and were implemented using exactly the same simulation as Genetic Algorithms.

Although some hardware could have been designed to generate software, all programmes were generated with the use of software. A hardware version of the Automated Guided Vehicle (AGV), the robot that has been chosen for this experiment, has been built for the purpose of testing the programmes. Chapter five shows the layout of this AGV, how it has been built and why it has been built in that way. The emphasis was on making it small, inexpensive and simple.

A total of three programmes were written to make this experiment possible. The reason and placement for each is explained in chapter six.

The Virtual Programme generates a programme known as the Simulated Programme. The Simulated Programme is later executed on an AGV. Chapter seven gives details on how a simulation of the AGV in a virtual maze is used to determine the fitness level of a Simulated Programme. These fitness levels are used by either Genetic Algorithms or by Single-Chromosome-Evolution-Algorithms to generate the new generation of Simulated Programmes.

In most experiments, only one destination is used. This is generally in the opposite corner when compared to the original position of the AGV in the maze. When the split option is activated, the AGV has to learn how to react to an input from a human operator. The value of a single bit, simulating an input from a user, is generated randomly. The value of the bit determines to which one of the two destinations the AGV should go. In all experiments, the fitness level is equivalent to the linear distance between the AGV and the relevant destination.

Mutation is used by both Genetic Algorithms and Single-Chromosome-Evolution-Algorithms. Chapter eight indicates that it takes time and is not the best option to use only mutation to generate a Simulated Programme. Although all instructions, using any operand, could be generated by this new mutation method, instructions that are used for obstacle avoidance are inserted more extensively.

In chapter nine, the programme that controls the Physical AGV from the laptop is shown, while chapter ten shows the working of the programme on the Physical

AGV's PIC microcontroller. These programmes communicate with each other via an umbilical cord. This combination makes it possible to determine how well a generated programme will act when used on a Physical AGV.

The Physical AGV is controlled by an exact copy of the virtual controller, which controls the virtual AGV. The controller, known as the Simulated-Functions-of-the-Microcontroller, has almost the same layout and instruction set as a PIC microcontroller and is explained in chapter eleven.

All the resulting levels of fitness of all relevant experiments are given in chapter twelve. The influences the different options have on the fitness values are calculated. Examples of Simulated Programmes that displayed obstacle avoidance capabilities are analysed. In section 12.8 the conclusion of the whole experiment is given and, thereafter, future recommendations are made.

2 Robots

The following chapter will illustrate the basic layout of most robots as well as different types of robots. First the history of robots is investigated.

Mechanical engineering, Electrical engineering, Computer science, Mathematics and Physics are all combined to form one discipline called robotics [1, p.10][5, p.1,1]. The difference between ordinary machines and robots is that robots collect information, process this data and then function accordingly as if they were intelligent. [5, p.2,13] This is called a closed feedback loop and is described in **figure 2.1**.

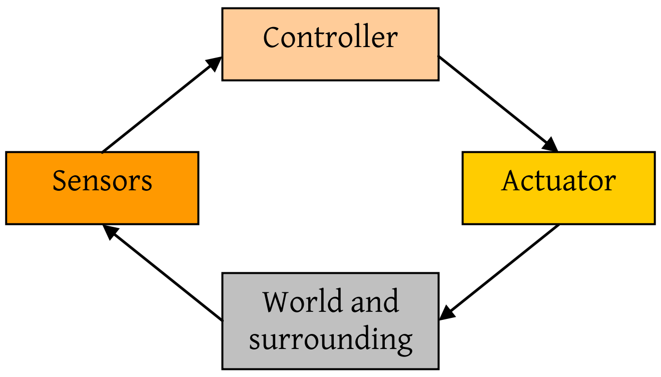


Figure 2.1 Representation of a Closed Feedback Loop for Robots

Sensors installed in the robot itself can collect information from its immediate surroundings. A transducer, in collaboration with an electronic circuit, functions as a sensor [1, p.547]. Some sensors can detect things that a human cannot. Radar for example, is a type of sensor. Various sensors were not necessarily invented for robotics [50]. For a machine, determining and understanding what is being sensed is more difficult than sensing something [34]. The actuator, on the other hand, makes movement possible [5, pp.3,1-3,15 & pp.10,1-10,28]. If the robot does not have a body it is not a robot, but some artificial intelligence system [13, p.7].

Some techniques, such as the optimisation technique of Genetic Algorithms, cannot be directly implemented between the inputs from the sensors and the output to the actuator of the AGV. A controller has to interface these two parts as shown in **figure 2.2.** [34]. There is another internal part that can be added, this is the memory of the AGV and usually forms part of the controller. This dissertation concentrates on the characteristics of the PIC microcontroller as the controller for the AGV. However, this dissertation also includes other types of controllers.

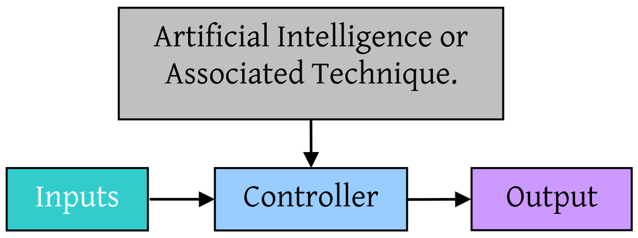


Figure 2.2 Insertion of Artificial Intelligence Techniques on a Robot

The simulated controller interacts with the simulated motors and simulated sensors, or with the real motors and real sensors, of the AGV. The controller forms part of the closed feedback loop described in **figure 2.1** and it is the part that does the “thinking” for the robot.

2.1 Buggies

Buggies are very simple in design. Some buggies make use of an umbilical cord to connect to a computer. They have two main wheels in order to propel and steer the buggy. Balancing the buggy is accomplished by a castor wheel or peg. Forwards or backwards motion is accomplished by letting the wheels turn in the same direction. If the wheels are made to move in opposite directions the buggy will be able to turn. Sensors are not always included but can be implemented to give feedback to the computer that is controlling the buggy [2, pp.112-113] [3, pp.48-51].

2.2 AGV

Automated Guided Vehicles (AGV) are used in many factories today to convey a number of things. "There are effectively four key parts to an AGV system. The vehicle, its route – (the path decided upon by the factory or building planners) – the AGV's controller, and its guidance system." [4]

The guidance is, for example, a painted line on the ground where the AGV senses this line and moves on it as though it were a rail. Another guidance system consists of wires imbedded in the floor of the building. Modern AGV can move away from their guide path. Laser guidance on AGV uses lasers and reflectors to determine its position. With this information it follows a path.

Another way is using sensors to avoid obstacles in its path. Although the AGV is three-dimensional, its paths can be described in two dimensions [1, p.401]. A forklift can even be utilised as an AGV [4].

Because of its many uses and simplicity, an AGV is going to be used for the purpose of this experiment. The focus is going be on decision-making capabilities, as well as the robot's capability to adapt to specified conditions.

2.3 ROV

Remotely Operated Vehicles (ROV) are usually used in the inspection and maintenance of offshore oil-wells, laying and inspecting of communication cables, in geological geophysical surveys of the ocean floor and as underwater salvage machines [1, p.45]. More often than not, a human controller controls the ROV from a ship at the surface. A cable is used to link the controller to the ROV. The controller sends signals to the ROV, and the ROV gives the controller feedback from its sensors, these sensors mostly include a video camera [20].

A closed-loop is formed in the following way:

1. The ROV sends information to the controller (which, in this case, is a human).
2. The controller sends signals to the ROV in order for it to be able to move something.
3. This change in situation is picked up by the sensors and transmitted to the controller.
4. Again the controller reacts, and, in this way, a closed-loop control is formed [3][20].

2.4 AUV

An Autonomous Underwater Vehicle (AUV) also has a closed-loop similar to the ROV except that the controller is now, for example, an electronic circuit. The electronic circuit makes the umbilical cord unnecessary for this type of robot. In the future AUV could be used for recovering “Black Boxes” from aircraft wreckages. They could even be made with the ability to communicate with the “black boxes”. The aim is to design an AUV such that it will get information from the “black-box” therefore making the need to return with the original “black box” redundant [20].

2.5 Humanoids

Humanoids are robots that are made to look and act as though they were humans. The problem with this type of robot is that it is difficult to make due to its complexity. The Oxford English Dictionary defined a robot as a "mechanical device that looks like a human being and works in a similar way." This is basically the oldest form of a robot that was concentrated on. The first of these robots did not have a lot, if any, decision-making capabilities. Companies like Honda have already built a robot that can walk like a human [22].

2.6 Controllers

Some sort of controller usually controls a robot; an example of this arrangement is an AGV. The controller can be a human, but in most cases it is a processor of some kind. It could even be a mechanical controller [23, pp138-139]. The controller is the unit that determines the actions of the robot considering the situation the robot is in. Hence this is the part that is doing the thinking and is therefore the brain of the robot.

2.6.1 Logic Gates as Controllers

If the AGV does not need any memory, then the controller could be built up by just two sets of gates, and each set's output is connected to a motor [1]. This, however, is only viable if two outputs are used. The inputs to the controller are connected to the inputs to the sets of gates.

Figure 2.3 is not an actual controller that can be used, but it is drawn to illustrate an idea. A separate programme that uses algorithms can be used to determine the connections between the gates and the types of gates that have to be used. A circuit's effectiveness is evaluated through the way of simulation. In an experiment conducted on a robot, a chromosome of an evolutionary algorithm determined the type of logic element used, if specific connections existed between some logic elements and other logic elements, and if specific connections existed between logic elements and sensors [17].

If the circuit needs to remember anything then memory could be created by connecting some of the outputs to the inputs of the same circuit [24, p.405-414]. The connections and type of gates are determined in the same way as in the first paragraph, but with the additional feedback. Microprocessors and memory are basically built up out of logical gates. Consequently, if this method were very successful, it would be able to build up a computer made especially for a certain application. It could even be a more successful computer or circuit than the ones that have been built so far.

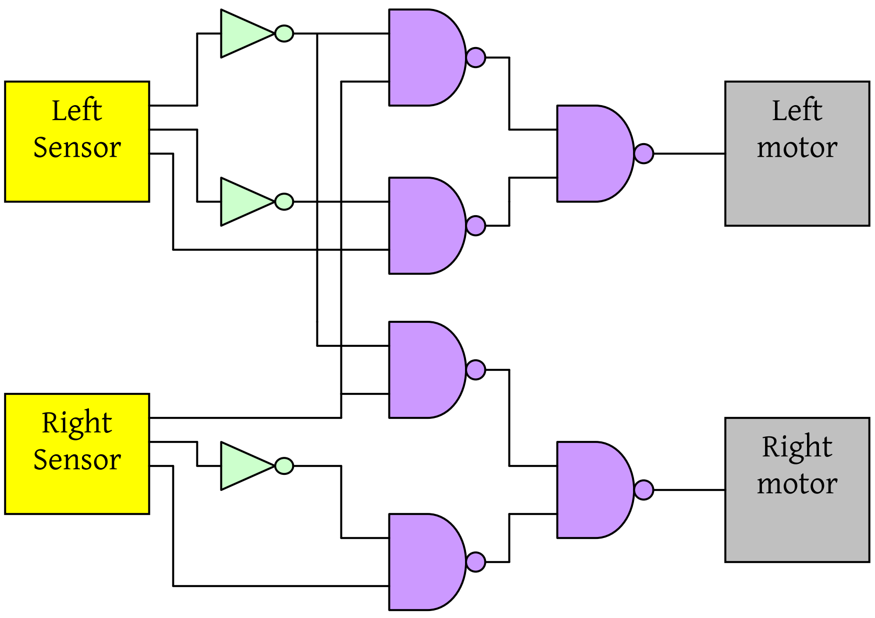


Figure 2.3 Logic Gates Could Function as a Controller for a Robot

If a human used this method to simulate or build up a processor it would be necessary to first program the internal structure of the processor. All the relevant internal circuits of a normal processor will have to be programmed.

The biggest possible problem is that the amount of variables could make this method difficult to work with.

2.6.2 PAL as the Controller of a Robot

If logic gates were used as a controller, where only the connections are changed and all of this is placed in a single IC, it would be analogous to a PAL. A PAL, which stands for Programmable Array Logic, is made up of AND-gates and OR-gates, where the AND-gates' outputs are connected to the inputs of the OR-gates, as shown in **figure 2.4**. The inputs to the AND-gates can come from any input, and can be made to be inverted if need be. Altering the internal connections connected to the inputs of the AND-gates program the PAL. Unfortunately the connections between the AND-gates and the OR-gates cannot be changed [37, p.374-379]. It is possible that some algorithm can determine the programme for this integrated circuit.

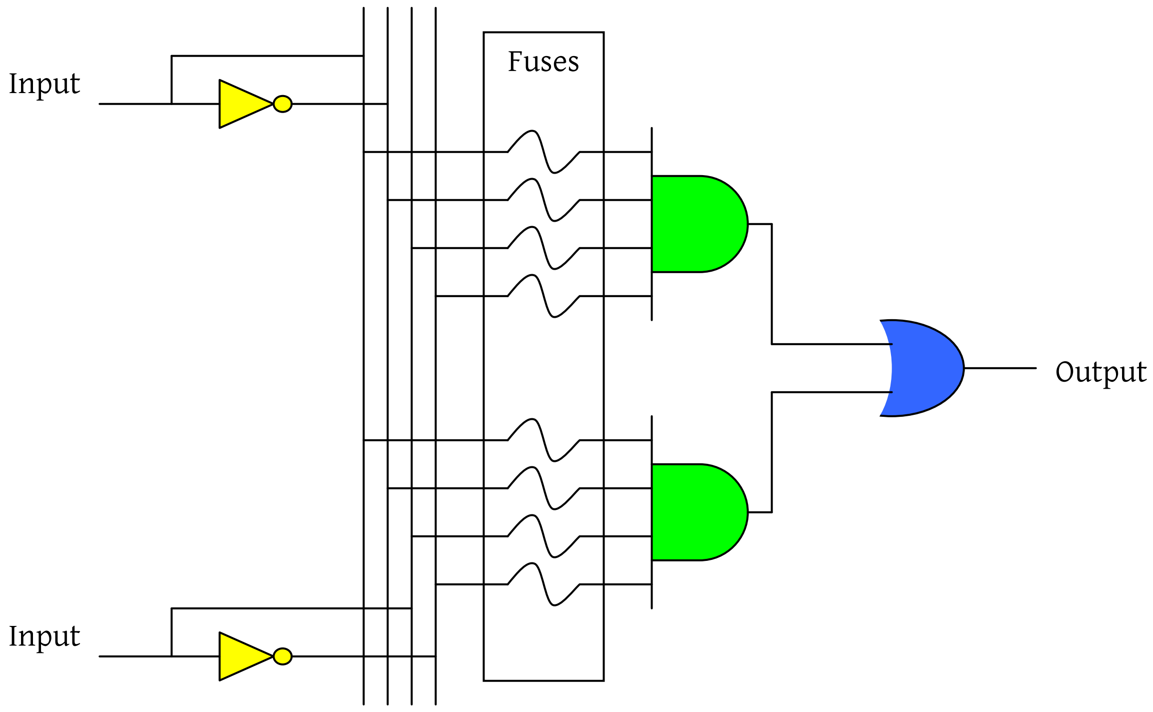


Figure 2.4 Example of a PAL Layout

2.6.3 PLC Used as Controller

Off-the-shelf Programmable Logic Controllers (PLCs) are used to control elements of an automation system [14, p.18], and they are designed to replace relay logic. Programming is accomplished with ladder diagrams as shown in **figure 2.5**.

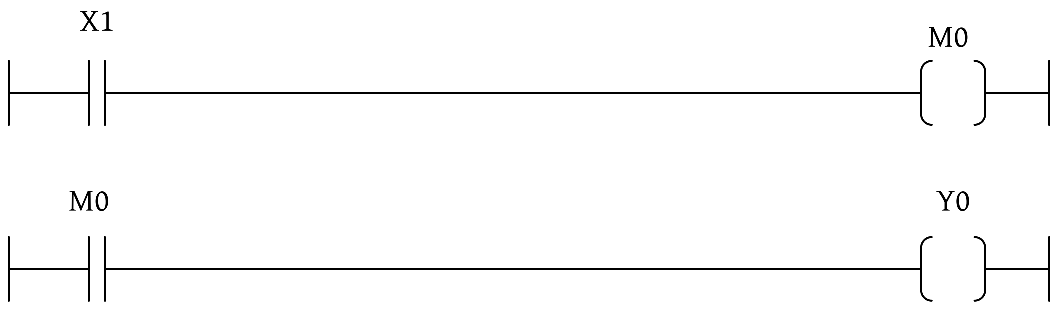


Figure 2.5 PLC's Ladder Diagram

The symbols have the following meaning:

- X1 is an input and acts as a switch. There are two types of switches, *normally open* and *normally closed* switches. More than one switch can be connected to a coil.
- Y0 is an output from the integrated circuit.
- M0 is either a memory coil or memory switch. They are connected to each other and form a virtual relay.

By closing all the switches in a ladder, the coil will be energised and this will alter the switches that are connected to that coil [25, pp.1-6].

A kind of logic gate can be built up in this way. Envisage that there are two switches connected to a coil. Both switches have to be closed to energise the coil. It is similar to an AND-gate, where both inputs have to be positive in order to create a positive output. The inverse can be simulated be using a normally closed switch.

The entire programming process can be accomplished by using an optimisation technique and has already been implemented in this way.

2.6.4 Using Only Memory as the Controller

A memory chip can be used to replace the logic controller. The address part acts as the input and the data part acts as the output, as shown in **figure 2.6**. For any input to the controller, any output can be prearranged. If the information in the memory is not changed then the outputs will always remain constant for a specific instantaneous input. The advantage, however is that it can be changed at will. By using extra circuits, or an optimisation technique for generating the programme, the information can be changed to adapt to the robot's surroundings. This is identical to the Nervous Nets that will be explained later in this chapter. Of course, although memory is used, the controller will act the same as logic gates without feedback would. Thus, it will have no memory of what happened to the robot despite the fact that it is made up of memory [26].

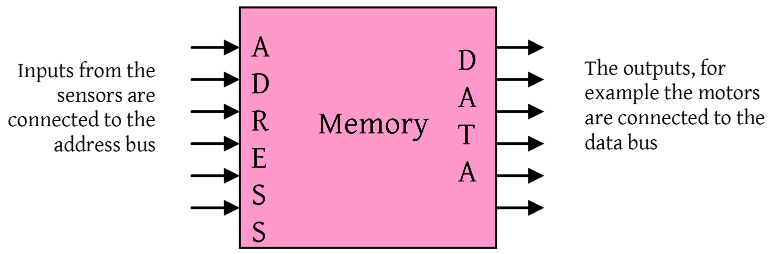

Inputs from the sensors are connected to the address bus
A D R E S S
Memory
D A T A
The outputs, for example the motors are connected to the data bus

Figure 2.6 Memory on its Own Could Function as a controller

As will be seen in the section 2.7.2, the memory in a memory chip can, and has been, changed by a certain technique to make the robot more intelligent or more adapted to its surroundings.

2.6.5 Memory with Feedback

As was seen under section 2.6.1, as soon as a circuit is given feedback, it possesses memory. The result is a controller that will not only react to the instantaneous input, but will be able to react to a combination of previous inputs, and if need be the current input to the controller.

The problem with the memory output is that it is possible that not all the outputs will change at the same time. A method of synchronisation is necessary so that the output will not be mistaken for something it is not. In order to do this a latch can be used. The output of the latch is then connected to a part of the memory's address bus. The input to the latch is connected to part of the data bus as shown in **figure 2.7**. This method is similar to the logic gates that have been connected back to themselves. The big difference is the latch, which takes a certain amount of time before it gives an output and also synchronises the outputs given.

In an additional experiment which has been constructed for this dissertation, on Artificial Intelligence, a memory chip can be connected as described above. Only one address bit has been left out as an input to the system. This one bit is used to select between one of two addresses and is the least significant bit of the following address. The remainder of the address information for the following address is contained in the data bus. If this input bit is zero, the total address will make up one address. If this bit is set to one, while the rest of the address stays constant, it will point to another address. For example, if the address held in the current data part is the following 7 bits: 0101011 and the input bit is made one then the following 8 bit address will be: 01010111, but if the input is made zero then the following 8 bit address will be: 01010110. With this method it is evident that each address is pointing to two addresses. All that is needed is the knowledge of what combination of ones and zeros makes up the rest of the following address. All addresses could contain the same information in the data part, and thus more than one address could be made to have the same couple of following addresses.

Extra circuitry is needed to determine the information in the data bus. Some of this information is part of the following address, as explained. For reading applications, an indication if one is allowed to change the input bit into one, and if one is allowed to change the bit into a zero, is all contained in the data of the memory. This information is built up by the experience of the robot, thus, if the selection bit was a one, a zero or both in earlier identical situations. As a result not all addresses have the option of indicating to two addresses. It is possible that an address will select only one address and that the selection bit can only be a zero for that situation.

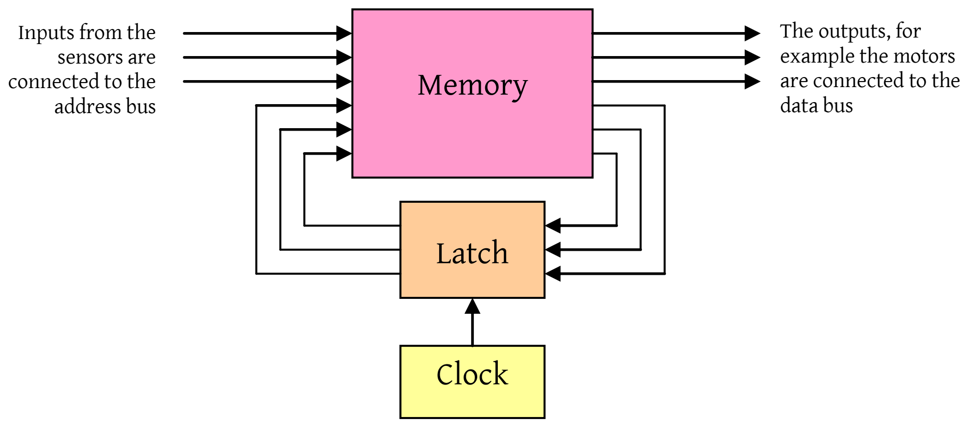

Inputs from the sensors are connected to the address bus
Memory
The outputs, for example the motors are connected to the data bus
Latch
Clock

Figure 2.7 Memory Feedback Circuit

In conclusion, an optimisation technique can be used to change the memory to form almost any circuitry function.

2.6.6 Computer or Microcontroller

The most common method is to use a computer or microcontroller of some sort as the controller for the robots. This section evaluates the type of controllers that can be used.

The PIC microcontroller made by Microchip, when compared to a personal computer, is cheap, small, and does not have any moving parts [18, p.5]. For these reasons hobbyists like to use the PIC microcontroller [13, p.77]. It is unfortunately limited in operational speed; it has a small memory, and does not have functions, such as “random”, that are frequently used in these methods.

The 80X86 CPU has a lot of instructions, memory peripherals, and operational speed. Another advantage to the personal computer (PC) is the user interface it provides and the wide variety of software available [14, p.18]. The disadvantage is that it consumes a large area and the moving parts in the form of a storage system, for example hard disk, make it susceptible to bumps which can cause serious damage.

Because the Z80 CPU needs external memory, it is larger than the PIC microcontroller and not really more useful.

The Motorola 68HCXX is used frequently in robotics [13, p.77]. The Motorola MC68000 is not suitable for this application although it has a more feasible architecture than the PIC microcontroller. With all its peripherals it is bigger than the PIC microcontroller and slower than the 80X86 CPU.

The 8051 is used for a wide variety of things in the automotive industry [13, p.77].

2.7 Artificial Intelligence and Associated Techniques

The aim of the study of Artificial Intelligence is to develop a system that acts intelligently. Most of the time this work is part of computer science but can also include the field of psychology, linguistics and mathematics. Because intelligence is such a vague concept it is broken down into the following aspects: understanding language, ability to learn, reasoning, solving problems and more [6, p.290-291].

The only thing that should be represented to an autonomous system is a goal [16]. For a robot to be able to learn it must be given an indication on how well its performance was, thus how close it got to the goal [55]. The following section gives an overview of some Artificial Intelligent systems and techniques.

2.7.1 Fuzzy Logic

In normal Boolean logic, an input or output can only be a one, representing true, or a zero, representing false, where as with fuzzy logic there are degrees of being true. The degree of being false is just the complementary value of how true something is [16].

If a specific aspect of a man has to be described it can be said that it is a tall man. The information "man" can be subdivided, for example into tall and short men, as illustrated by **figure 2.8**. All men are represented by the square. The circle defines all tall men.

Figure 2.9 shows two circles. The second circle shows men that are fat. The overlapping of the two circles is for men that are tall as well as fat. The question is at what length a man becomes tall. One can see that there is a point were one would not know if a man could be described as tall or not. Therefore the circles do not have sharp edges, as shown. There is a gradual change from when a man is described as not being tall to when a man is described as tall.

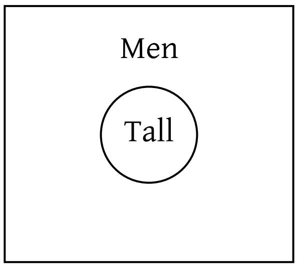


Figure 2.8 An Explanation of Tall Men in Fuzzy Logic

In logics, tall would be illustrated as having a value of one and, for not tall, a value of zero. Fuzzy logic has more values and the values are between one and zero. The length of a man is placed in a formula and the result is between one and zero. So the taller the man is the more the man description will move into the circle. There is still a point at which the man is so tall he could only be described as being tall, at this point the result of the formula is one. The same goes for not being tall.

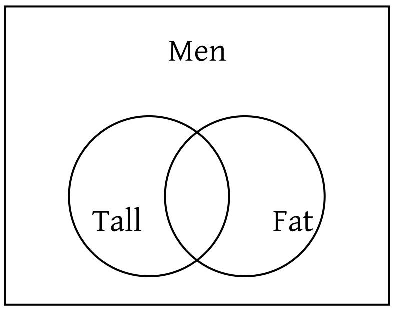


Figure 2.9 An Explanation of Tall and Fat Men in Fuzzy Logic

As with normal logics, fuzzy logic also has the functions of *and*, *or* and *not*. An example of the *and* function is when a man is tall and fat as shown in **figure 2.9.** If one looks again at the figure one will see that the area outside the circle and inside the square is for men that are not tall. Points inside any of the two circles, including the intersection, are for men that are either tall, fat, or both.

To get the inverse value the value calculated can just be subtracted from one. If the *and* is incorporated, then the lower of the two values calculated is taken, and the higher of the two is taken for the *or* function [27, p.1-48].

2.7.2 Nervous Nets

The controller for this method is similar to the memory only controller described in section 2.6.4. The method used to change the memory's contents is not accomplished with Genetic Algorithms. Firstly putting random data into the memory generates the initial generation. Then it is tested in much the same way as determining fitness for Genetic Algorithms. This is not determined by simulation but by real, live situations. During the course of the test the information is changed and assessed to get the best results. The advantage is that the control unit is much smaller than that of the Genetic Algorithms. [26]

2.7.3 Neural Networks

In a human brain there are a vast number of cells known as neurons. These neurons are used to build up patterns of knowledge in our brains. Neural Networks try to accomplish the same behaviour by simulating the functions of brain cells [28]. Almost all approximation techniques which are in the form of a network can be seen as a neural network [29]. Signal enhancement, noise cancellation, classification of input patterns, system identification, prediction and control are examples of functions that the neural and adaptive systems perform. These functions are used in modems, image-processing, image-recognition systems, speech recognition, front-end signal processors, and biomedical instruments [7, p.2]. Types of Neural Networks, as well as one way of teaching perceptron, are briefly discussed in the following section.

2.7.3.1 Schematic Net

A person would need a map in order to travel from town to town. There could be more than one correct route to a specific town. This map will show the towns one has to go through to get to a specific town. This is the basic principle of schematic nets as can been seen in **figure 2.10** [30, pp.15-21 & 63-79].

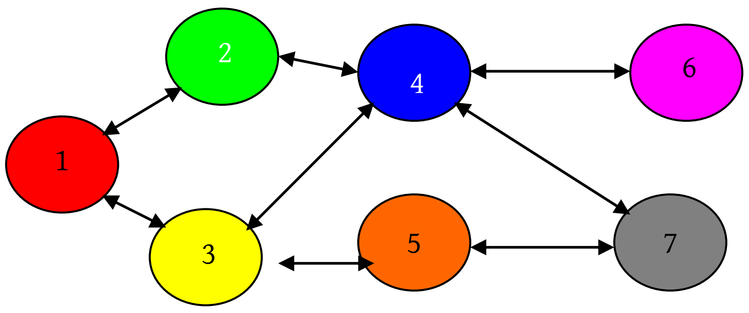


Figure 2.10 A Map of Towns

This map, to a certain extent, shows the way a person thinks. In a person's brain, decisions are made by saying to oneself, if this happens, then one of several results could occur. In **figure 2.10**, if situation one occurs, then situation two or situation three could occur. If situation three occurred, then situation four or five could occur, but if situation two took place then only situation four could occur.

An alternative way of illustrating this is with the search tree shown in **figure 2.11**. Loops, such as going from situation one to two, to four to three and back to one, have been left out. This search tree is what is used in Neural Nets. In much the same way as a goal is needed to determine the fitness in Genetic Algorithms, a goal is needed here. Different methods have been used to get a route from the current situation to the goal.

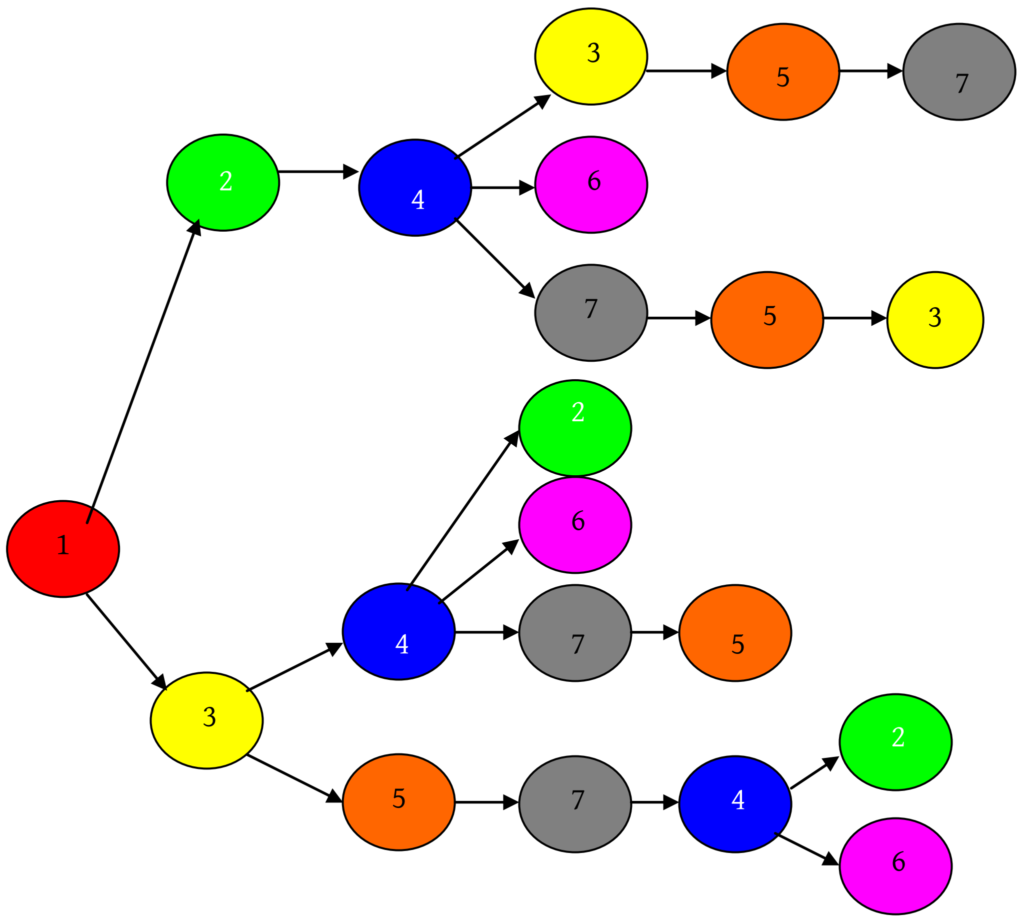


Figure 2.11 Search Tree Representation of a Map of Towns

2.7.3.2 Single Layer Perceptrons

This is a computation that is the closest simulation to the activities of a brain cell. Each input is multiplied by some value that is called the weight [29]. All the results are added together as shown in **figure 2.12**. It is then placed in a function, usually a function that is similar to the sine or cosine functions. This produces a single result from a single perceptron. This result can be the next input to another layer of perceptrons.

Normally perceptrons have a logic input, thus they have a one or a zero as an input. The output of the perceptron is then also a one or a zero. A threshold determines the output. If the result of the formula is more than the threshold, then the output is made one, otherwise it is made a zero [30, p.443 - 489].

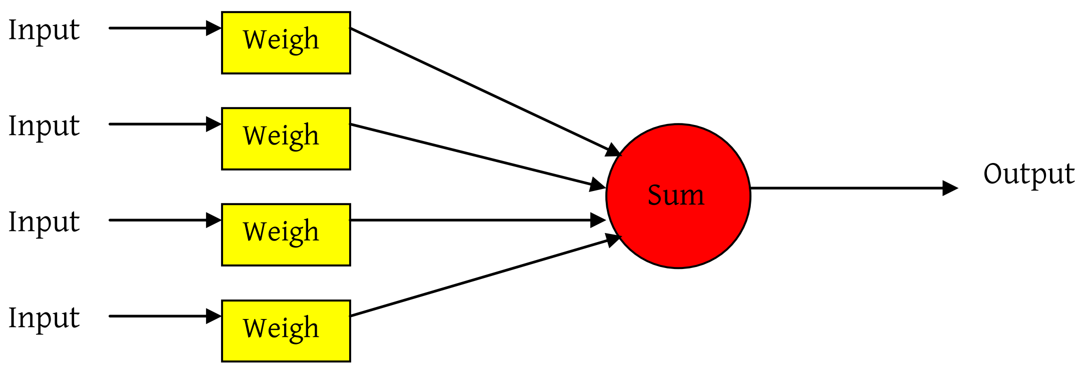


Figure 2.12 Simulated Neuron

Imagine that the threshold is 0.9 and all the weights are 1. If any of the inputs are made to be one, then the sum is more than 0.9 and the result is an output of one.

The only time the output will be zero is if all the inputs are zero. As can be seen, this is similar to an OR-gate.

If the threshold is made 3.9, then the output will be zero for all inputs unless all the inputs are one. This is again similar to an AND-gate. A gradual change from one function to the next is thus possible by changing the weights.

The threshold in the formula could always be made to be the same. Changing the overall threshold is accomplished by a dummy input and weight where the input is always one and the weight is changed in the normal way.

Change the sketch in **figure 2.12** to have one input, and with the additional threshold input, the simulated neuron will be similar to the one in **figure 2.13**. The resulting formula is $Y_1 = (W * X_1) + b$, which is a linear expression, thereby making the model linear [7, p.8]. Despite the fact that a single neuron is linear, a Neural Network can model a non-linear system [16].

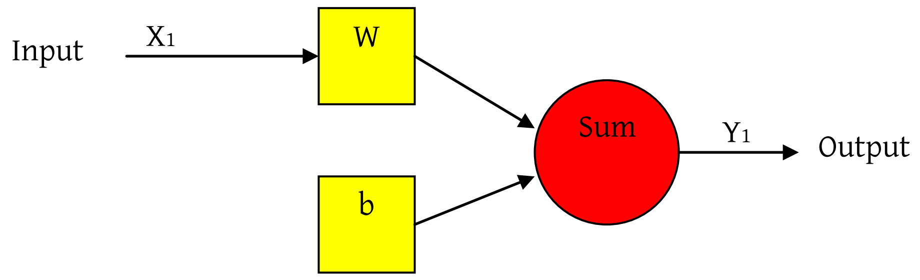


Figure 2.13 Illustrating the Linear Response of a Simulated Neuron

2.7.3.3 Multi-Layer Perceptrons

Normally, in logics, the outputs from a number of AND-gates are connected to the input of an OR-gate. Thus, it is necessary to have more than one layer here and usually, three layers are used [29].

As in logics, if one needs more outputs more circuits are required. The network in **figure 2.14** has three layers and every output of one layer is connected to all the inputs of the next layer. The circles represent the summations and the lines the values that have to be added together. Each line also represents the input and the weight that has to be multiplied with the input. The amount of perceptrons in each layer does not have to be the same.

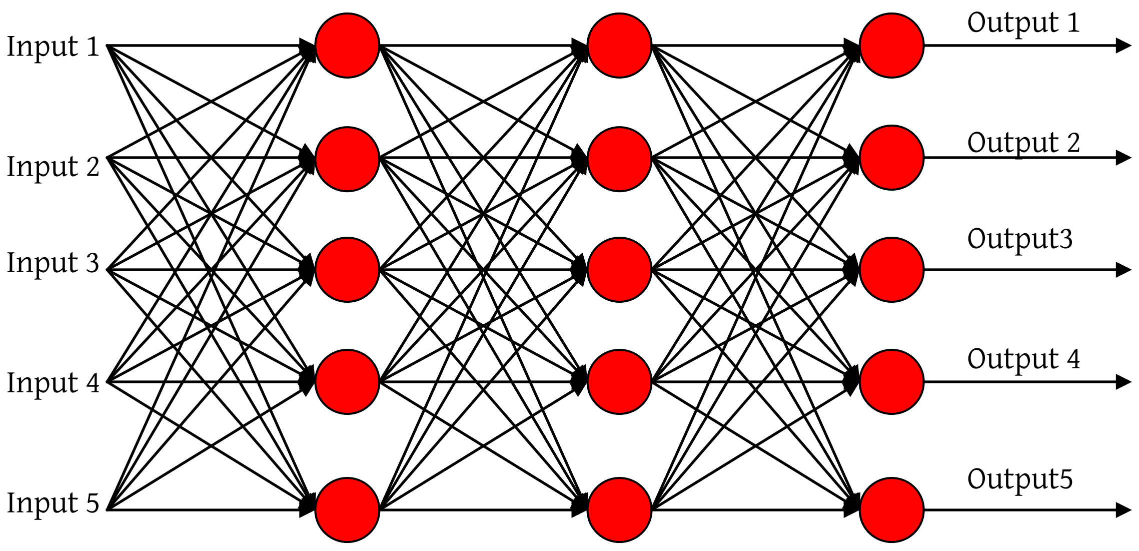


Figure 2.14 Multi-Layer Perceptron

As with the logic gates described earlier in this chapter, feedback networks could be created to generate memory for a neural net [11, pp.701-711].

2.7.3.4 Teaching the Perceptron

Teaching the perceptrons is accomplished by changing the weights [16]. The effect is shown in the section 2.7.3.2. A method that can be used to change the weight is to only change the weights of the inputs that have an input of one. Another option is to use the back propagation procedure [29].

Look again at **figure 2.13**. Because the input to output is linear, a line, as shown in **figure 2.15,** could represent it. The formula for this would be $Y_1 = (W * X_1) + b$, as was shown. Each black dot represents what the desired input to output relationship must be. Thus, the goal is to adjust the position of the line to minimize the overall distance between the line and all the dots. This is accomplished by changing the values of W and b [7, p.17].

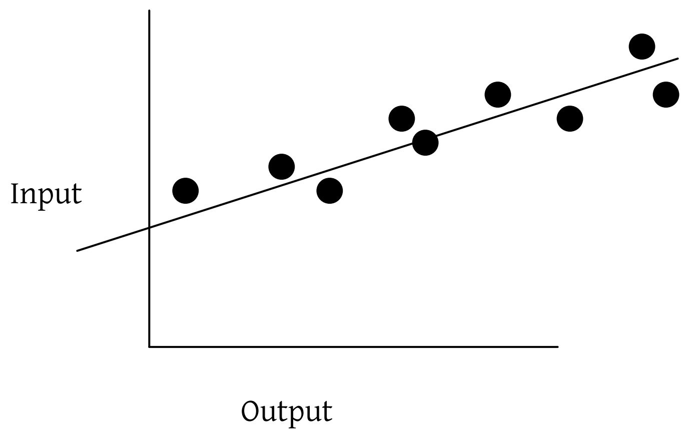


Figure 2.15 Graph Showing the Input to Output Relationship of a Perceptron

If there are two inputs to the perceptron then the formula is changed to $Y = (X_1 * W_1) + (X_2 * W_2) + b$. This is represented by a flat plain with the desired dots all around it in a three dimensional graph [7, p.41].

2.7.4 Intelligent Hybrid System

Systems like Genetic Algorithms, Fuzzy Logic and Neural Nets may be good in one aspect but bad in another. Because these aspects are not the same for all the options, a combination of the above methods can be used for various functions. If one looks at the decision-making capabilities of man, then one will see that they are governed by neural impulses and genetics that have evolved over millions of years.

There are different ways of implementing these combinations. One way is to fuse two or more methods. An example of this is where Genetic Algorithms change the weights of Neural Nets. Another way is a system where a black-board is incorporated. All the methods are used to work on one problem [31].

2.7.5 Genetic Algorithms

Genetic Algorithms can be defined as many things, including being a potential basis of machine learning [8]. "Generic Algorithms (GA) can be used as a direct analogy of natural evolution. Through the genetic evolution method an optimal solution can be found and represented by the final winner of the genetic game" [32, p.6].

"In the biological world, the fundamental unit of information in the living system is the gene. In general a gene is defined as a portion of a chromosome that determines or affects a single character or phenotype (visible property), for example eye colour. It comprises of a segment of deoxyribonucleic acid (DNA), commonly packaged into structures called chromosomes" [32, p.1].

"GA presumes that the potential solution of any problem is an individual and can be represented by a set of parameters. These parameters are regarded as the genes of a chromosome and can be structured by a string of values in binary form. A positive value, generally known as a fitness value, is used to reflect the degree of "goodness" of the chromosome for the problem, which would be highly related with its objective value.

Throughout a genetic evolution, the fitter the chromosome the more of a tendency it has to yield good-quality offspring, which means a better solution to any problem. In a practical GA application, a population pool of chromosomes has to be installed and these, initially, can be set randomly. The size of this population varies from one problem to another, although a number of guidelines are given. In each cycle of genetic operation, termed as an evolving process, a subsequent generation is created from the chromosomes in the current population. This can only succeed if a group of these chromosomes, generally called "parents" or the collection term "mating pool", is selected via a specific selection routine. The genes of the parents are mixed and recombined for the production of offspring in the next generation. It is expected that, from this process of evolution (manipulation of genes), the "better" chromosomes will create a larger number of offspring, and thus have a higher chance of surviving in the subsequent generation, emulating the "survival-of-the-fittest" mechanism in nature" [32, pp.6-8].

Instead of setting parameters in advance, neural and adaptive systems use external information in order to set their parameters automatically as seen in **figure 2.16**. Performance feedback makes the system "aware" of how close the result is to the desired goal [7, pp.2-3]. Adaptive systems do not always build up information from scratch; sometimes it is implemented on existing trajectories to change it a little for more accurate results [10, p.359]. **Figure 2.17** shows the flow of a Genetic Algorithm process [16].

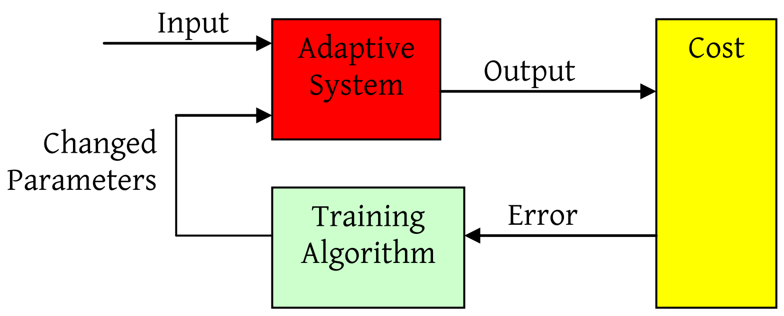


Figure 2.16 Adaptive System's Layout

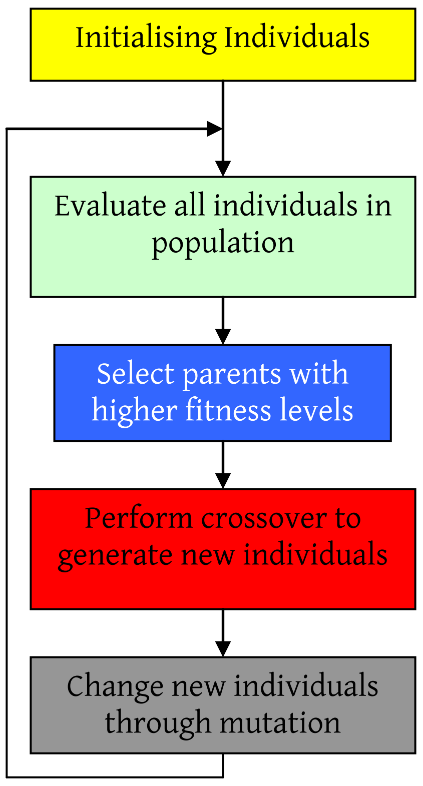

Initialising Individuals
Evaluate all individuals in population
Select parents with higher fitness levels
Perform crossover to generate new individuals
Change new individuals through mutation

Figure 2.17 Flow Diagram of Genetic Algorithms Processes

2.7.5.1 Selection of Parents

The most common technique being used for proportionate selection is a scheme called *Roulette Wheel Selection*. A member of the population is chosen. If the fitness of the member is more than the number chosen, then it is used for mating. This ensures that the higher the level of fitness is, the more likely the individual will be to reproduce [32, p.8]. After the parents have been used they are removed from the population.

2.7.5.2 Generation of Offspring

In Generic Algorithms two methods can be used simultaneously to produce a new chromosome. The one, and most widely used, is *Crossover* and the second is *Mutation*. Crossover is basically the mating of chromosomes. This is accomplished by connecting the first half of one chromosome to the second half of another chromosome, as shown in **figure 2.18**. This produces a third chromosome. The point where the cut has been made is called the crossover point and is determined by a random number [32, p.9]. More than one such crossover point can be used for generating one chromosome. The result is that there is an alteration between the parents' information each time a crossover point is reached. The generation of a Two Crossover Point offspring is shown in **figure 2.19**.

If one desires the result as shown in **figure 2.19** while using only one crossover point**,** it can be accomplished by using the method shown in **figure 2.20**. The red chromosome is a third chromosome in the population and an extra generation is needed. A half-brother and half-sister mating would yield exactly the same result, as in **figure 2.18**. More crossover points can be created in the same way if need be.

In some cases it is easy to see why multiple crossover points are used instead of single crossover points. With multiple crossover points fewer chromosomes are needed, but, more importantly, fewer generations are needed. Also, the chance that the programme would choose the correct chromosomes is slim. Of course this is similar to inbreeding, which leads to problems, as will be explained in the section "Inbreeding Problems using Genetic Algorithms" later in this chapter.

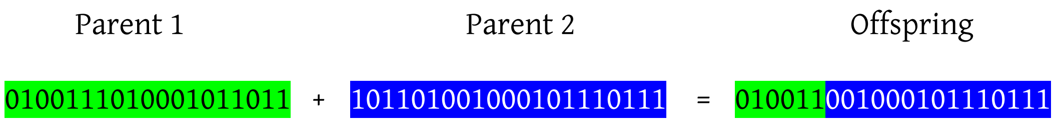


Figure 2.18 Single Crossover Performed on a 19 Bit "Program"

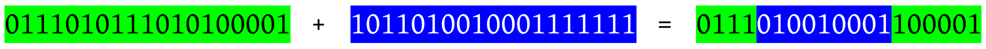


Figure 2.19 Two Crossover Points Performed on a 19 Bit "Program"

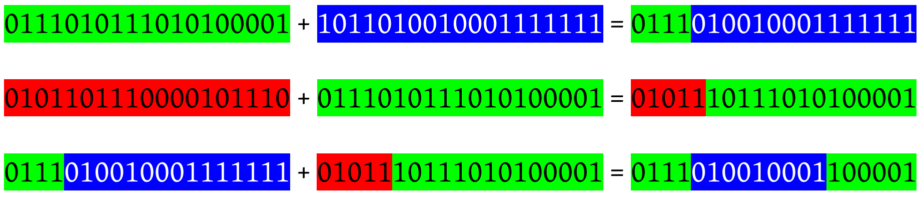


Figure 2.20 Single Crossover used to Generate Multiple Crossovers

Mutation is any alteration in a chromosome, as shown in **figure 2.21** [32, p.6]. In Genetic Algorithms this is mostly brought into being after the crossover has been executed. Normally the changes made must be kept as small as possible because, majority of the time mutation has a negative effect on the fitness of a chromosome. Subsequently, the bigger the mutation the more negative the effect would be [56, p.71]. It is possible to use mutation on its own without using crossover to produce offspring [30, pp.513-516].

0111010111010100001 => 0111100111010100001

Figure 2.21 Mutation by Changing Bits Randomly

2.7.5.3 Fitness

The fitness of the population's individuals determines the direction in which the population is developing. Determining how close the chromosome is to its goal is how the fitness of a chromosome is determined. Fitness is normally determined through simulation, but real life situations can also be used.

Not only does the AGV have to be simulated, but the surroundings as well. A list of rules is then set out, and these determine the level of fitness.

Majority of the people that have worked with Genetic Algorithms have their own set of rules. The application for which the Genetic Algorithms are used will also dictate these rules.

By giving the programme too few rules, it is possible that the evolution will not even start to develop. An example of this is when a chromosome has to be very specific to do something, such as to take the first step. If the fitness is determined by only the distance to the destination, then the fitness could stay unchanged although the chromosome might better itself, and, consequently, could make it very hard just to start moving.

In this application the processor has to send a certain output to the motors. These motors can be connected to an extended output. This means that the chromosome (programme in this case) has to go through many hardware configurations to get to the correct output.

Giving the programme too many rules is similar to programming the AGV. For example, to say that it has to execute a certain test before giving it a higher fitness rating is like telling it, it must execute a certain "if .. then" command.

Sushill gave higher fitness levels to robots that moved along an obstacle's boundary [33]. One may question why he did that. All that is required of the robot is to move to the required destination in the least amount of steps without bumping into anything. This is like programming the robot to move along a boundary because you know that that method will work. The reason for this can be as described in **figure 2.22**.

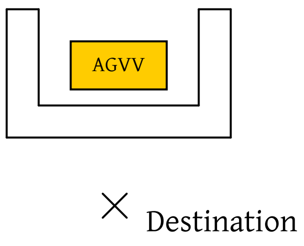


Figure 2.22 Local Maximum for AGV

2.7.5.4 Previous Implementation of Genetic Algorithms

The most used implementation of Genetic Algorithms in robotics is in a hybrid system were the Genetic Algorithms are used in collaboration with some sort of neural net. In some applications the Genetic Algorithms were used to change the weights of the perceptrons [34]. Researchers have tried to train Neural Networks for decision making problems using Genetic Algorithms [54]. Because any hidden node can react the same way as any other hidden node, there are a lot of configurations that will lead to exactly the same result. Using crossover could lead to a copying of some information while another part is ignored. Some people call this the *Competing Conventions Problem*, whereas others call it *Permutation Problems* [54].

Genetic Algorithms can determine the layout of the controller's logic gates and associated connections, as shown in the section 2.6.1. In one experiment two robot arms were simulated, the aim of the experiment was to make the two arms avoid bumping into each other. The speed of each link for a certain time is given by a formula that contains a graphical output [35]. The form of the graphic is determined by Genetic Algorithms, thus Genetic Algorithms change some of the variables in the formulas.

Many changes and additions have been made to the traditional Genetic Algorithms in order to make them perform better. One such method is to incorporate sexual preferences. In this experiment the individual searches for another individual with similar characteristics. Choosing a mate became just as important as obtaining food or avoiding predators. The addition made the population break up in more than one species [36].

One way of overcoming the problems of all the chromosomes concentrating on one local maximum is to give more diverse chromosomes higher fitness levels. The difference between two chromosomes is calculated and an additional fitness level is determined according to this difference [30, pp.519-523].

Great results where achieved by giving conventional Robot Control Theory a head start. It was accomplished by giving the robot initial knowledge about the system [38, p.364].

2.7.5.5 Inbreeding Problems Using Genetic Algorithms

A result from many applications of Genetic Algorithms is quite a number of identical chromosomes. Similar chromosomes might have developed as a result of the following situation. A parent has two children. The first one has the first half of the parent's chromosome and the second one has the second part of that particular parent. If these two mate it could give an offspring identical to the "grandparent". If these parts (for example from the start to the crossover point) of the original parent (grandparent) were big enough, then most "grandchildren" would be the same as the "grandparent". This process is shown in **Figure 2.23**. Of course, if two identical chromosomes had to mate, and no mutation took place, it would give an offspring that was identical to the parent. This process, known in nature as inbreeding, would force evolution to stand still.

0100110111 + 1010110000 => 0100110000

Figure 2.23 The Grandson is a Replica of the Grandfather

A method that has been used to overcome this problem is to make sure that there is a reasonable difference between the parents. Another way is by forcing mutation onto an offspring that is exactly the same as another chromosome. In Genetic Algorithms, mutation renders, in most cases, a lower fitness level. Majority of the time, in nature, mutation generates deformities.

One way of ensuring the next generation is by keeping a part of the old generation. The part that is kept is the fitter part of the original population. The reasoning behind this is "Why throw away something that is already working?" With this method, a chromosome can survive through many generations if its fitness level is strong enough. This situation changes if the average offspring is getting too strong in which case it will not survive even if it survived quite a few generations.

With this method, the chromosomes that survived can be made to have equal opportunity to mate [32, p.8].

2.8 Advantages and Disadvantages of Simulations

Brooks and Mataric [9] said that many people weaken the detail in a simulation, and even if it was as correct as possible, it would not be very accurate. They went on to say that global information is not available to a real robot, whereas it could be available to the simulated one. In simulation it is sometimes difficult to determine how the robot is learning. In some instances the simulation is harder to work with than the real situation [9].

Despite this opinion, simulations are commonly used to test programmes for industrial robots before being used on a real robot. This saves on the time the robot would not have been in service. This is known as offline programming [10, p.15]. It seems that there is a big advantage to simulating a robot and its work-cell before building it [6, pp.338-340], and this dissertation is not the first example where simulations are used in collaboration with evolutionary algorithms for controlling robots [17].

2.9 Summary

Figure 2.24 show some of the options to a robot. A robot is built out of actuators, controllers and sensors. The sensors pick up information from the world around them. This information is processed by the controller and a message is sent by the controller to the actuators who then change the orientation of the robot relative to the robots' surroundings. This change is picked up by the sensor and the process is repeated.

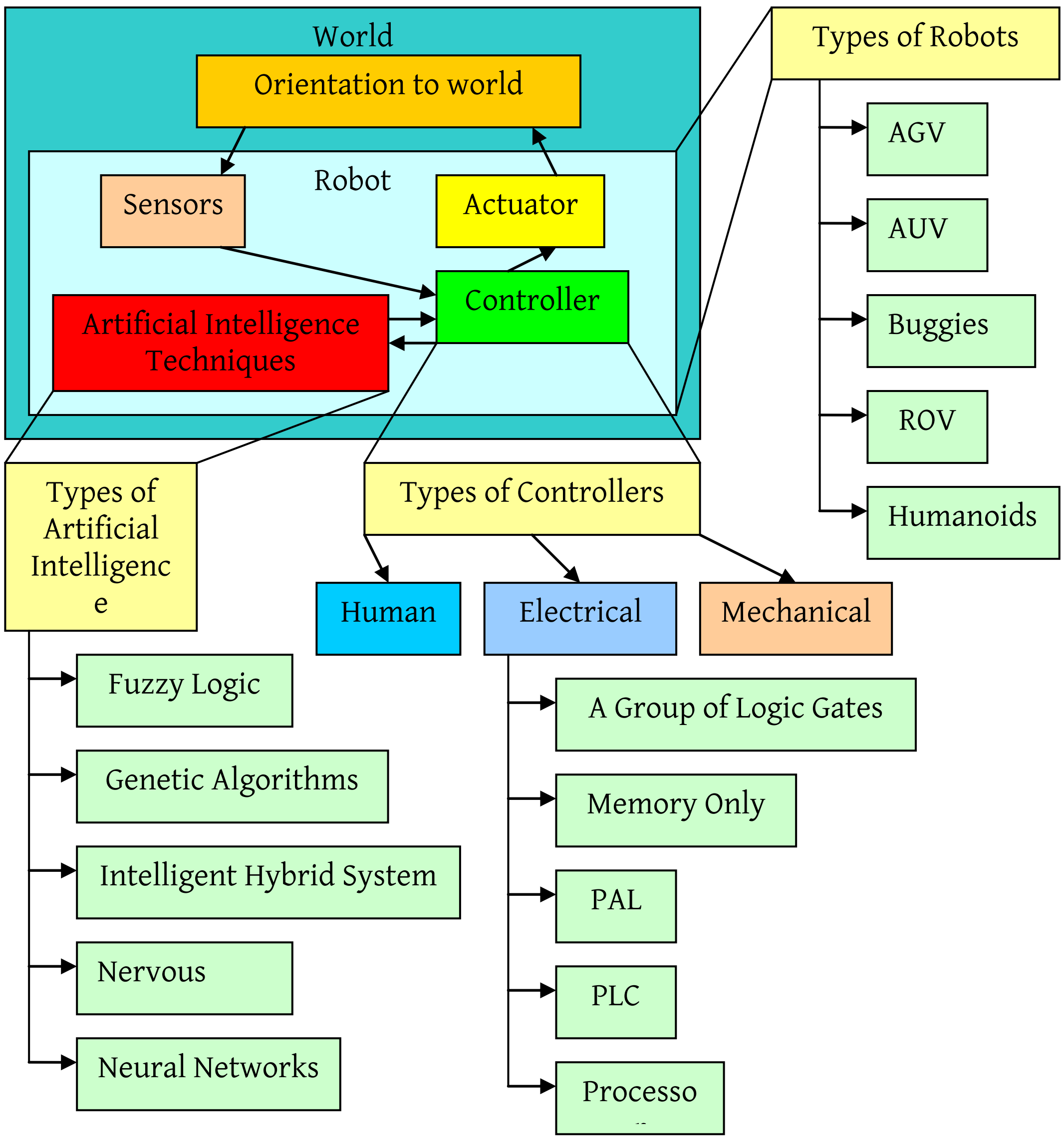


Figure 2.24 Layout of Typical Robot

Listed below are some possible forms of a robot:

- Buggies
- AGV: or Automated Guided Vehicles are used to convey things
- ROV: or Remotely Operated Vehicles are operated by a human and are mostly used offshore.
- AUV: or Autonomous Underwater Vehicles are the same as ROV, except that they use an electronic circuit as the controller.
- Humanoids are robots that are suppose to look like humans.

Controllers for a robot can be one of the following:

- Human
- Mechanical
- Electrical

The electrical controller seems to be the most widely used option and can be any one, or a combination of the following:

- A group of logic gates.
- PAL
- PLC

Memory only is an example is RAM, where the sensors are connected to the address part and the actuators are connected to the data part.

Memory with feedback is the same as Memory only, except that some of the outputs are looped back to the input. Timing must be considered for this to work.

Computers or Microcontrollers are the most common type used.

Enabling the robot to act more intelligently, artificial intelligence techniques are incorporated in conjunction, or alone, with the following as possible choices:

- Fuzzy logic
- Nervous Nets
- Neural Networks
- Intelligent Hybrid System
- Genetic Algorithms

Genetic Algorithms are an adaptive algorithm that was inspired by evolution in nature. It uses the following processes, also to be found in nature:

- Concerning the *selection of parents* the higher the fitness level of an individual is the better the chances of being chosen as a parent.
- *Generation of offspring* by using crossover and mutation.
- *Fitness level.*

In this dissertation, Genetic Algorithms generate a programme for a Simulated AGV. Fitness is determined through simulation. The fitness level is directly proportional to the distance between the last position of the AGV and the destination.

3 Genetics in Nature

A lot of knowledge can be obtained by observing nature. Many of the principles used in nature may also be used by Genetic Algorithms. However, it is important to note that although these principles might work in nature, it does not necessarily mean that they will work in Genetic Algorithms. Studying nature could result in the discovery of new methods or a new way of thinking about an old method. This chapter focuses on the reason behind certain things happening to Genetic Algorithms. It also illustrates what possible future experiments could be performed.

3.1 *Man's Closest Relative*

In nature, Chimpanzees have been proven to be the closest relatives to man. Scientists believe that a chimpanzee's DNA is 96 to 99 percent the same as human DNA [39, p.74]. As a result, the difference in DNA is 4 percent or less. Although there are a lot of similarities between human behaviour and chimpanzee's behaviour, the chimpanzee has learnt to use primitive tools while humans have gone to the moon and built the super computer [40, pp.20&36]. Even birds have learnt to use tools [41, p.20]. It is thus possible that a small change in the chromosome used in Genetic Algorithms could lead to a big change in the fitness of the individual.

3.2 *Small Groups in Nature*

Gorillas form relatively small groups. This would normally create problems relating to inbreeding, but fortunately males move over from one group to another [39, p.74, 136] [42, p.121-124]. George Schaller calls these males' *Lone Males* [42, pp.121-124]. Different groups also mingle for a while and separate again at a later stage.

The smaller the population in Genetic Algorithms, the less complex the programme is and the faster it will go through generations. Small populations lead to similar chromosomes being generated. This problem is overcome by deliberately changing a small part of the chromosome to form different chromosomes. Most of the time, these mutations give lower fitness levels and can be seen as problems generated by inbreeding [45, p.69][46].

Mimicking nature by making a copy of a randomly chosen individual and later using it in a totally new simulation could represent the *Lone Male.* Simulating the same space in time is accomplished by inserting this *Lone Male* after the same number of generations has past as the number of generations that had past when it was "extracted" from the other group. The smaller the number of *Lone Males* that are inserted into a specific population, the grater the difference will be between the *Lone Male's* chromosome and the other individuals in the population. Thus two populations exchanging *Lone Male's* will probably generate better results than one population that has twice as many individuals and is not using *Lone Male's.*

3.3 Similarities between Nature and Genetic Algorithms

A method that is used in Genetic Algorithms is to remove the whole generation and replace it with offspring. This, to some extent, is also done in nature. The Octopus female dies shortly after she has given birth [43, p.12]. The same thing happens to the Praying Mantis where the female eats the male after copulation has occurred. She may even begin to eat him before they are done [21].

3.4 Why Use Mutation

Von Ditfurth said, “The tempo of evolution is dependent on mutations.” He also goes on to say that if the surroundings do not change then less mutations are needed [44, pp.227-228]. The results in this dissertation show that mutation is much more important than crossover for generation of programmes.

3.5 Inbreeding

Inbreeding is a term used when the parents of the organism are closely related to each other. Organisms are considered closely related when they are related within five generations. Farmers use inbreeding to get a specific characteristic from a particular organism. An example of this is say a person that breeds with exotic birds. A breeder may like a specific bird and want to “make a copy” so to speak. This could be accomplished by taking the child of the bird and mating it with the parent, in other words, the bird that comes from this is the child; as well as the grandchild, of the first bird. That is because his father is his mother’s child. One could continue to take this bird and mate it with the original bird for the same result. It is important to note, however that not all bird species’ can use this method without yielding complications. Also, this method does not necessarily produce a better one, just a replica of the original. It is not advisable to use this method in Genetic Algorithms because a better result is needed.

Although this method has certain advantages, inbreeding mostly leads to unwanted results. When one looks at cows and pigs, the disadvantages that come from inbreeding are reduced reproduction, increased mortality, poorer growth rate and more. Van Rensburg said that this might occur because there are hidden deformities, which form part of the family bloodline [45, pp.69-73]. When these organisms mate, the deformities are added together and it becomes more obvious. Section 2.7.5.4 will demonstrate that mutation has to be incorporated in inbreeding when used in Genetic Algorithms or else evolution will stop [45, pp.69-73] [46]. Mutation may also be why inbreeding causes deformities.

Out-breeding occurs when two unrelated organisms mate. This is the opposite of inbreeding and can be used to neutralise inbreeding.

3.6 Cross-Breeding

Crossbreeding is when two different breeds mate. This process creates an offspring with a higher fitness level than that of the parents. An example of this is where two races are crossbred to form new individuals. If these individuals, although possibly unrelated, mate, then the offspring's fitness level will be less than that of the crossbred parents.

With every new breed brought into the population the offspring gets fitter. A method called "Rotational crossing" is widely used by pig farmers in many countries. With this method more than one breed is systematically introduced into the population [45, p.69-73] [46].

In Genetic Algorithms, it will thus be expected that with every introduction of a totally different chromosome with similar fitness levels to that of the average fitness level of the population, the total fitness level of the population will increase.

3.7 Convergence

The Bluefin Tuna and the Mackerel Shark have features that are very much the same. This is because they had to adapt to situations that are similar [41, p.39]. The expected results of different generations of Genetic Algorithms will be similar if the conditions to which the Genetic Algorithms are subjected to are similar.

3.8 Good Competition

In some Genetic Algorithm experiments, a chromosome with a high level of fitness, which has been previously generated, has been placed into a new population. The results for this were very good, but is it not expected. If the chromosome were to be placed in a new population, then it would generally have the highest fitness level. There will be a very high possibility that it will be used as a parent. Luckily, all of the old generation chromosomes were removed before the next generation. If this removal did not occur the chromosome could have survived for many generations, making the descendents more like him with each generation.

With the method that was used, a few descendents would have been produced using the newcomer. The offspring with the biggest part of the newcomer would probably have had the highest fitness level and would therefore have very good chance of being selected as a parent. Consequently, this population will become more and more like the population where the newcomer was generated and can, in some ways, be regarded as just applying more generations to the first population.

This is much like the war between the Aztecs and the Spaniards, where 400 Spaniards killed off a million people because they had more advanced technology. Carl Sagan said that if there was to be a war between two alien species, coming from different planets where the one specie must have developed more that the other, then the one would wipe the other out completely. This example was taken because the two species, like the two populations, had developed totally independently of one another [47, pp.342-338].

In conclusion, it is inadvisable to insert an individual with a much higher fitness level into a population with a lower average fitness level.

3.9 Summary

It is possible that with each different individual, with similar fitness levels, brought into the population, the total fitness level of the simulation will improve. This will sort out, amongst other things, inbreeding problems that might exist. The problem, however, is that, because of convergence, this new chromosome might not be that different from the other chromosomes in the population. Inserting a chromosome with a higher fitness level than the fitness level of the rest of the population, might change the whole population into similar versions of this new chromosome, thus not accomplishing anything of significance.

A small change in the chromosome could lead to big changes in fitness level.

4 Single-Chromosome-Evolution-Algorithms

This technique is a combination of Nervous Nets and Genetic Algorithms. Single-Chromosome-Evolution-Algorithms are comparable to Genetic-Algorithms, where one evaluated parent is paired with one of an infinite number of newly generated, non-evaluated parents. These infinite, newly generated, non-evaluated parents are not real, but imaginary. Because it is newly generated, it is random.

This is the most straightforward Algorithm of the three used in this dissertation and that might be the reason that it has not been used before. The idea is born out of the basic need for a self-generating programme. It was hoped that the following two advantages would be good enough to make up for the disadvantages of the less complex algorithm:

- Only one simulation has to be executed for a generation and this saves time. Simulation is, by far, the most time-consuming part of this experiment.
- Less recourse is needed because only two chromosomes have to be saved.

It might be possible to use a simulator with a smaller amount of data memory. Single-Chromosome-Evolution-Algorithms can use exactly the same methods for determining fitness levels, as was used by Genetic Algorithms.

4.1 Generation of a New Chromosome

Because the population is made up of a single chromosome, the generation of a new chromosome is also the generation of a new population. By implementing mutation on a chromosome, a new chromosome is generated. Since this dissertation deals with the generation of a new programme, a whole instruction has to be changed, as will be explained in the section "Special Mutation" in chapter seven. Changing an instruction in a programme is similar to changing a single gene in a chromosome.

There are two memory positions that contain two chromosomes. At the start, the two chromosomes are totally different because they are randomly generated, but, as will be seen, the chromosomes will become similar after the first generation of a new chromosome. An illustration examines the first few generations of the evolution. The original fitness value for both chromosomes is made zero, as shown in **figure 4.1**.

	Chromosome	Fitness
Chromosome 1	1010101010	0
Chromosome 2	0110011001	0

Figure 4.1 First Population Chromosomes were Randomly Generated

Only chromosome 1 is evaluated through simulation. From there the name Single-Chromosome-Evolution-Algorithms is derived. The result from the simulation, as in the case of Genetic Algorithms, is fitness. As shown in **figure 4.2,** the fitness level of chromosome 1 has changed to two.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1010101010	0	=>	1010101010	2
Chromosome 2	0110011001	0	=>	0110011001	0

Figure 4.2 Only One Chromosome has to be Evaluated

If the evaluated chromosome's fitness level, chromosome 1 in this case, is better or the same as the other chromosome's fitness level, then the evaluated chromosome is copied into the other chromosome. After completion, both memory spaces contain a copy of the chromosome with the higher fitness level. The level of fitness does not

have to be determined for the new copy of the chromosome because the new chromosome is a copy, and, thus, the earlier determined fitness level can be used. In short, the fitness level is also copied. The reason why only one chromosome has to be evaluated for each generation is because the previous determined fitness level, which was the highest of all the previously determined levels, acts as the current fitness level for chromosome 2.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1010101010	2	=>	1010101010	2
Chromosome 2	0110011001	0	=>	1010101010	2

Figure 4.3 Copy Better Chromosome into Both Memory Spaces

Changes are only made to the chromosome that has to be evaluated, as in **figure 4.4**. As already stated, these changes are similar to normal mutation used in Genetic-Algorithms. The result is a newly generated chromosome, and a new population is generated.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1010101010	2	=>	1001101010	2

Chromosome 2	1010101010	2	=>	1010101010	2

Figure 4.4 Random Changes Made to Chromosome that have to be Evaluated

Continuing with the second generation, another evaluation of one of the chromosomes has to be conducted. As seen in **figure 4.5,** the fitness level of chromosome 1 has changed from two to one after the evaluation. The new fitness level is due to the change made to the chromosome as highlighted in red. The resulting fitness level is generally worse than previous levels, but as will be evident, the change is only remembered when the change produces a better fitness level. From now on, the fitness level of the generated programme will not be less than two, thus a fitness of two has to be beaten before the resulting chromosome changes.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1001101010	2	=>	1001101010	1
Chromosome 2	1010101010	2	=>	1010101010	2

Figure 4.5 Change Coursed a Lower Fitness Level

In **figure 4.5,** the new chromosome has a lower fitness level value than what has previously been obtained. The change was for the worst and, by coping the original chromosome back into the evaluated chromosome, it ignores the changes made to it as, shown in **figure 4.6**.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1001101010	1	=>	1010101010	2

Chromosome 2	1010101010	2	=>	1010101010	2

Figure 4.6 Copy Old Chromosome Back into Both Memory Spaces

On the other hand, if the fitness level obtained after the evaluation is better than the other, thus the changed chromosome is better, ensuring that changed chromosome is kept for the next generation. **Figure 4.7** shows how this is accomplished by copying the evaluated chromosome into the other one.

	Chromosome	Fitness		Chromosome	Fitness
Chromosome 1	1001101010	3	=>	1001101010	3
Chromosome 2	1010101010	2	=>	1001101010	3

Figure 4.7 Copy Changed Chromosome as well as the Fitness Value

In **figure 4.7**, the same result would have been accomplished if the "new" fitness level was two or higher.

4.2 Size of Mutation

Sometimes the changing of only one gene would not be enough. If a computer programme has to be generated, in some cases adding only one of two instructions would give a lower fitness level, but because these instructions work together they can give a higher fitness level if both instructions are added to the programme simultaneously. The next example will illustrate this.

Programme segment 4.1 displays the part of a programme where the AGV walks forwards until it hits an obstacle. It could be walking in the direction of the destination or, in some way, it is accomplishing a fitness level of, for example, 20. A single instruction is added, as in **programme segment 4.2**.

Because of the insertion of the instruction "GOTO Turn", the programme will probably never execute the "GOTO Forwards" instruction, altering the AGV to turn in its own tracks [18, p.154]. The result is a lower fitness level. The Single-Chromosome-Evolution-Algorithm's procedure will see the adding of this instruction as weakening the chromosome, ignore the changes made, and go back to the previous chromosome.

```
NOP
NOP
GOTO   Forwards
```

Programme Segment 4.1 Single Instruction Added.

```
NOP
GOTO   Turn
GOTO   Forwards
```

Programme Segment 4.2 Worst Programme.

In **programme segment 4.3** the "GOTO Turn" instruction is again added to the programme shown in **programme segment 4.1**, except that it is not added alone but in collaboration with "BTFSS", changing the programme into **programme segment 4.3**.

```
BTFSS      STATUS, Carry
GOTO    Turn
```

```
        GOTO     Forwards
```

Programme Segment 4.3 Two Instructions Added Simultaneously.

In **programme segment 4.3,** the AGV will go forwards until it reaches an obstacle, it will then turn until the obstacle is not in front of the AGV any more and it can move forwards again. The result is a higher fitness level than any previously determined ones.

The disadvantage of this method is, if the instruction or gene is eight bits in length and if only one combination of ones and zeros will give a higher fitness level, then the chance that that instruction will be chosen will be one out of $2^8 = 256$, while, if two genes are changed at the same time and only one combination will give a higher fitness level, then one would have a one out of $2^{16} = 65536$ chance of the ones and zeros being the correct combination. This amount will increase rapidly with each additional gene. Consequently, the amount of genes to be replaced at any given time should be kept as small as possible. A high level of mutation in Genetic Algorithms is normally seen as an unwanted situation [32, p.6] [56, p.71].

4.3 Making It Possible to Compare Single-Chromosome-Evolution-Algorithms with Genetic Algorithms

When looking at the Virtual Programme, it was observed that the evaluation of a single Simulated Programme could take a few minutes, while the time it took to generate a new population while using any method, was too fast for a human to detect. Thus, evaluation is, by far, the most time-consuming component of the generation of a Simulated Programme.

If a population has ten individuals, then ten evaluations have to be executed for one generation. If ten generations are needed, then a hundred evaluations have to be executed.

Comparing Single-Chromosome-Evolution-Algorithms to Genetic Algorithms, a hundred generations of Single-Chromosome-Evolution-Algorithms have to be executed for only ten generations of Genetic-Algorithms. Counting the amount of evaluations in either Genetic-Algorithms or in Single-Chromosome-Evolution-Algorithms makes it possible to compare the spread of these two methods.

4.4 Summary

There are three consecutive parts forming a continuous loop that make up Single-Chromosome-Evolution Algorithms:

- Evaluate the one chromosome through simulation the result is a fitness level. Exactly the same methods can be used as those used in Genetic-Algorithms.
- Copy the chromosome with the higher fitness level of the two into the memory space occupied by the chromosome with the lower fitness level. Fitness levels as well as the resulting chromosomes are made to be two exact copies of the chromosome that possessed the higher fitness level.
- Mutation changes are only made to the chromosome that is going to be evaluated. After completion, the chromosomes will not be the same and this makes it possible to compare the two chromosomes.

Changing more than one gene into a usable combination could be much more advantageous than changing only one gene into a usable combination. Unfortunately it is so much more unlikely that the programme will change more than one gene into a usable combination than it will be to change one gene into a usable combination that it is, in most cases, not recommended.

Because evaluations take up the most time, the time it would take to accomplish something has to do with the amount of evaluations executed. In order to compare Genetic-Algorithms to Single-Chromosome-Evolution-Algorithms, the amount of evaluations is counted in each case.

5 Physical Layout of the AGV

Via the umbilical cord power and information is sent to the robot, also known as an Automated Guiding Vehicle (AGV), as shown in **figure 5.1**. In this way, the computer does the computations for the Physical AGV and there are no batteries that are liable to run flat.

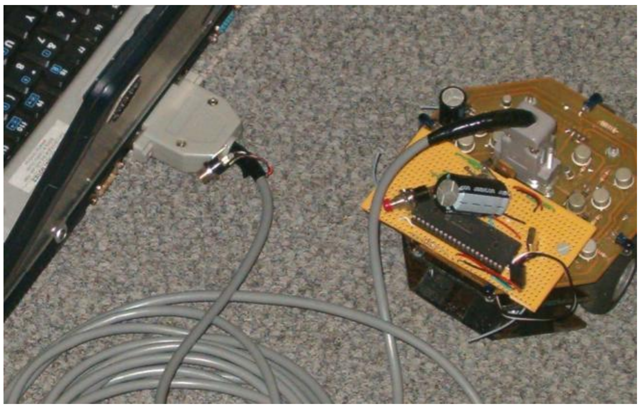


Figure 5.1 Photo of the Physical AGV

5.1 Considerations for the AGV

There are a few considerations that one must take into account when it comes to the Physical AGV. One must remember that the Physical AGV and the Simulated AGV must, as far as possible, react in the same way, because it is easy to transform the Simulated AGV entirely, whereas it is difficult and more expensive to make any change to the Physical AGV. In most cases, the Physical AGV influences the form and design of the Simulated AGV, but in some cases it is the other way around. The following considerations are to be discussed later in these chapters:

- Size of Physical AGV
- Speed and strength of Physical AGV

5.1.1 Size and Shape of the Physical AGV

The smaller the AGV, the better it will be at going though small spaces and it will, of course, be more portable. If being used in a maze, the maze's size will be determined by the size of the AGV, thus the bigger the Physical AGV is the less space it will have to move in. If a small AGV is needed, everything on the AGV must be as small as possible.

If the AGV were square and it turned around its own axel, the corners of the AGV would have bumped into an obstacle if it were to close to the AGV. This problem is avoided by making the AGV octagonal in shape, as shown in **figure 5.2**. It could have been made round, but making round electrical tracks and fitting rectangular components on such a PCB is more complex.

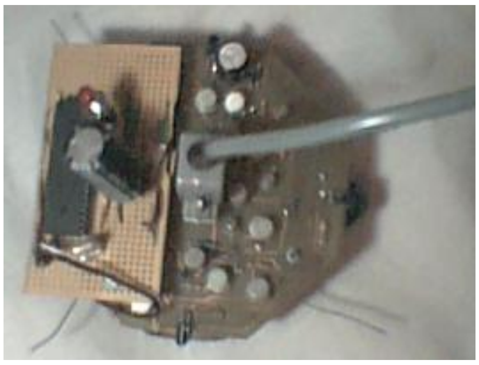


Figure 5.2 Photo Showing Octagonal Shape of Physical AGV

Once these methods are proven to work, the AGV could be made in a wide variety of sizes and thus it might be possible to use it in a wide variety of fields.

5.1.2 Motors of Physical AGV

The perception exists that the faster the AGV is, the better it will perform. Although speed does have its advantages, when it comes to this application it is not entirely true. Take, for example, when the AGV is at a position ten centimetres from a wall. The time it takes for the generated Simulated Programme to react to an input from the sensors is two seconds. The programme dictates that when the AGV is five centimetres from the wall or closer it must turn away from the wall. If the AGV is running at ten centimetres a second, the AGV will run into the wall before it has time to react, but if it is running at a mere centimetre a second, it will have enough time to react and turn away from the wall.

Small electronic motors run in the vicinity of 7000 revolutions per minute [19, p.3]. Lowering the voltage will slow down the motor, but the motor will not operate at a voltage lower than half the voltage specified for the motor [12, p.85]. Between the wheel and the motor a gearbox can be inserted to slow down the revolutions. In the gearbox a small gear is connected to the motor, which in turn turns a larger gear. If the small gear has twelve teeth and the big gear has forty-eight, then the small gear has to turn four times before the larger gear has turned only once. The velocity of the bigger gear is, as a result, a quarter of the velocity of the smaller gear. By connecting the larger gear's axel to another smaller gear, and again this smaller gear is turning a larger gear, the revolutions are brought down again. A combination of gears like this one is called a *gear train* [49, p.728].

Another way of determining the ratio of the gears is with the radiuses of the two discs. The velocity of a particle at the outermost part of a disc can be determined by the distance of the outline of the disc, multiplied by the revolutions per minute of the disc:

$V = 2 \Pi R N$

Where V = Velocity of particle

R = radius of circle

N = revolutions per second of the circle

If two discs are touching one another, then the peripheral velocity of the two discs is identical [48, p.45-49]:

$2 \Pi R_1 N_1 = 2 \Pi R_2 N_2$

$\Rightarrow R1\, N_1 = R_2\, N_2$

$$\Rightarrow N1 / N_2 = R_2 / R_1 \tag{5.1}$$

Formula one gives the ratio of the velocity of the two gears. *Moment* is defined as distance times the force that is applied at right angles to the measurement of distance [49, pp.64 & 65]. Not counting the friction in the gears, the force of the touching teeth is the same [49, pp.709–729].

$M = F \times R$

$\Rightarrow F = M / R$

$\Rightarrow M_1 / R_1 = M_2 / R_2$

$$\Rightarrow M_1 / M_2 = R_1 / R_2 \tag{5.2}$$

As can be seen from the two formulas, (5.1) and (5.2), torque is inversely proportional to speed [13, p.8]. The higher the *moment* is, the higher the acceleration will be [49, p.709–729]. **Figure 5.3** is not actual calculations and was created in order to illustrate an idea. The red line in the left hand graph illustrates an AGV with low acceleration and a high maximum velocity. If acceleration were ignored, as is the case in the Virtual Programme, then the shape of the curve would be like the one in the right hand graph. The lower the maximum speed, the less time the AGV will take to get to the maximum speed, as shown by the green line.

The stronger the output from a gearbox, the faster the acceleration of the AGV and the less time it takes to get to the maximum speed as shown by the blue line. This fast acceleration, in collaboration with a low maximum velocity, will lead to a shortening in time for the Physical AGV to get to the maximum velocity. This makes it possible to ignore the acceleration of the AGV in the simulations and, thus, simplify and shorten the simulation time.

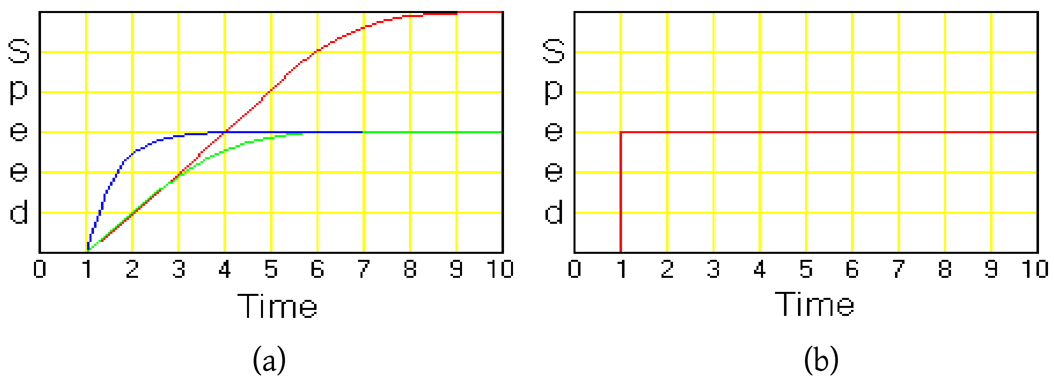


Figure 5.3 (a) Physical Acceleration of AGV (b) Ideal Acceleration of AGV

With strong motors and gearbox combinations the motors can be pulsated to slow down the AGV and still have enough strength to move it. Modified servomotors, normally used for radio control aeroplanes, are used for the movement of this AGV [13, p.9].

5.2 Components of the AGV

This AGV consists of the following components:

- Infrared-Sensors
- Physical-Controller
- Servomotors
- Controller-to-Motor Interface

5.2.1 Infrared Sensors and Whiskers

It is necessary to determine the distance to an obstacle to be able to avoid bumping into the obstacle. Distance sensors are, therefore, very important.

The distance sensors do not have to be calibrated. Drift must be small or else it will be difficult for the controller to adapt to the changing values.

Some distance sensors cannot determine distance if an object is too close to the sensor [51]. A scenario is given where the AGV is in the middle of two obstacles. If the obstacles are to close to each other then the sensors will not be able to detect either one of the two, as in **figure 5.4**. Thus, the minimum distance sensed by a sensor must be as small as possible to be able to move up close to an obstacle without touching it.

Gordon McComb (1987) said that the more sensors are used in a robot, the better it will be able to interact with its environment [12, p.12]. Because the AGV has eight sides, a sensor can be placed on each side of the AGV, where each sensor will face in its own direction.

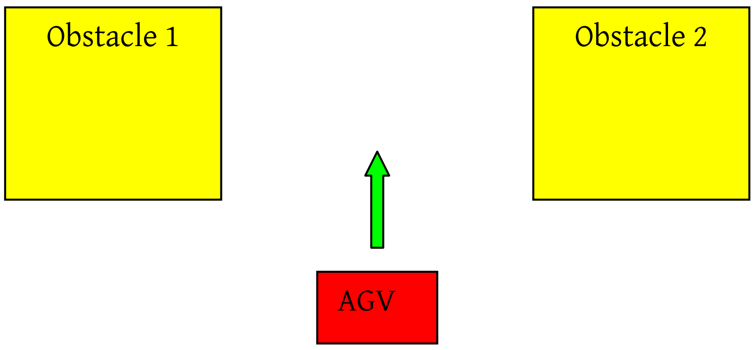


Figure 5.4 The Obstacles are too Close to the Sensors to be Sensed

A widely used method for determining distance in robotics is to use ultrasonic sensors, although, for smaller distances, infrared sensors can be used [50]. Infrared sensors are also small and, thus, make the whole AGV smaller [51].

An infrared sensor of the AGV is made up of an infrared transmitter and receivers shown in **figure 5.5**. Before sensing the transmitter is turned on, the intensity of the received light is used to determine the distance to a white obstacle. This analogue value is measured by an Analogue-to-Digital-Converter inside the PIC microcontroller.

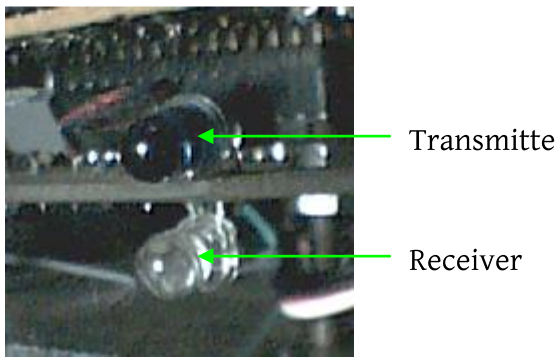


Figure 5.5 Photo of Infrared Transmitter and Receiver

This method of sensing with a PIC microcontroller is small, cheap and easy to implement. However, the disadvantages are:

- The signal is not linear. The closer the object gets to the sensor, the higher the value generated by the Analogue-to-Digital-Converter, and the more rapidly the readings generated by the Analogue-to-Digital-Converter change. Thus, to calibrate the reading of the Simulated AGV to correspond as closely as possible to the Physical AGV, extra calculations are needed in the simulation.
- The receivers do not distinguish between reflected light from the transmitter and other light in the same frequency band. Therefore, the light in the room will influence the resulting distance values generated by the Analogue-to-Digital-Converter. This causes a constant, random change in the output value generated. It is difficult for the controller to adapt to this constant change, as will be seen in the results of chapter 11.
- If the distance between the obstacle and the sensor is long, the intensity of the light being reflected from an obstacle is low and the influence of other light sources has a greater influence on the readings. This leads to the programme been unable to distinguish between the reflected light and other influences. As a result, the maximum distance that can be sensed is about 60cm under ideal conditions.
- The maximum output value is reached when the sensor is about 60 mm away from the nearest obstacle. This results in a large maze. Reducing the intensity of the transmitters light can reduce this minimum distance that can be sensed. Unfortunately it changes the maximum distance that can be sensed more dramatically.

- Specular reflection occurs when a light hits a mirror and all the light is channelled in one direction, whereas diffused reflection lets light reflect in all directions, such as when light hits a white wall [57, pp.98-99]. During calibration, it was found that, when it came to infrared light, the white obstacle adopted both types of reflections, thus resulting in higher readings when a sensor was at right angles to the obstacles surface.
- Because infrared is not visible to the human eye, the colour of the obstacle, as seen by a human, has less influence on the results than one might think but still has an influence on the amount of light being reflected off of an obstacle. White obstacles are preferred. [51]

By changing the resistor that is in series with the infrared transmitter, the maximum and minimum distance that can be sensed can be changed. Because the front sensors have to sense a larger distance than the other sensors, the front sensors have a 22 ohm resistor, instead of a 100-ohm resistor, in series with the infrared transmitter. These values were determined through experimentation.

Basically, Whiskers are made of a ring with a wire going through it that does not touch the ring, as shown in **figure 5.6**. When the wire is moved in any way, it touches the ring and current can therefore pass through the wire to the ring.

Instead of sending through more information, the registers from all the Distance Sensors, being eight bits in length, gave a maximum value of 255, thus emulating bumps from all the Distance Sensors simultaneously when any one of the whiskers are touched.

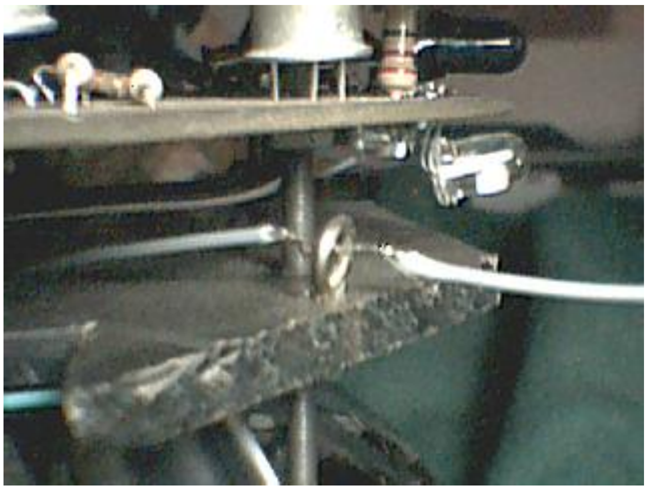


Figure 5.6 Photo of Whisker Touch Sensors

5.2.2 Physical Microcontroller

A small controller leads to a small AGV. Due to the size, low cost and simplicity, a normal buggy was used for the body and motors.

It was decided to use a PIC microcontroller as the controller for the AGV. It is small and does not have a hard disc and/or moving parts. The disadvantage is that functions resembling the *random* function, a function that is frequently used in algorithms of this type, is not part of the instruction set of the PIC. A procedure can be written to generate the random function, but it is not a perfect random generator and it takes up space in the programme [58]. Writing programmes for a computer with the aid of a higher level language is easier than writing a programme for a PIC. The speed of the PIC microprocessor, as well as the memory size, might not be sufficient for implementing Genetic Algorithms.

5.2.3 Controller Interface

This interface mostly consists of amplifiers to make it possible for the PIC microcontroller to drive the motors and to make the motors turn in both directions. To save on power, and because the infrared transmitters cannot handle the high current required for long periods of time, the transmitters are only turned on when it is needed [51]. Amplifiers are also needed to generate this high, controlled current. Because of the relatively small size of the AGV, care must be taken in the PC-board design.

Making it possible for the motors to turn in both directions, an H-Bridge configuration is used in collaboration with normal PNP and NPN transistors, as shown in **figure** 5.7 [13, p.43-46].

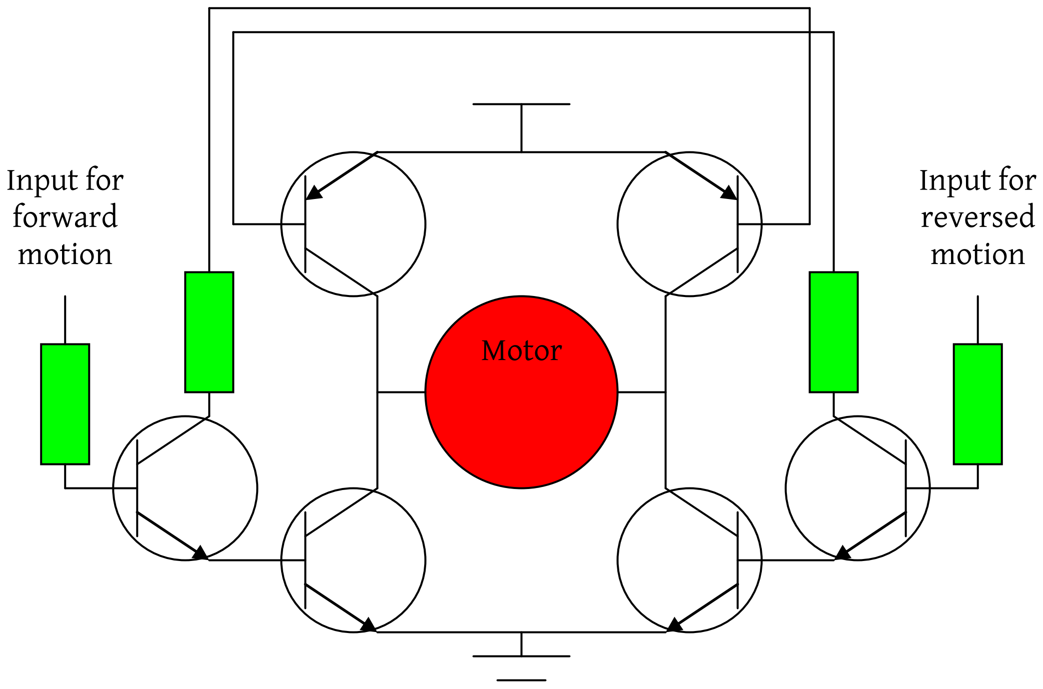


Figure 5.7 H-Bridge Lets the Motor Turn in Both Directions

5.3 Umbilical Cord

The use of the umbilical cord between the personal computer and the AGV has the following advantages:

- The processing power of a Pentium computer is much more than that of, for example, a PIC.
- An exact replica of the Simulated-Functions-of-the-Microcontroller can be used on both the virtual and Physical AGV. This can eliminate problems that might arise from not only the slight differences between the Simulated-Functions-of-the-Microcontroller and a physical PIC, but it can also eliminate problems with overheads that are not needed for the Simulated-Functions-of-the-Microcontroller.
- The power is fed through the cord, which makes the use of batteries on the AGV unnecessary. It is easier to make changes to the high-level-language than the assembler used to program the PIC. Even testing the hardware by changing the programme on the PC is easier than writing a new programme and programming the physical PIC.
- Loading the Simulated-Functions-of-the-Microcontroller with a newly generated Simulated Programme can be accomplished with the push of a button, instead of taking the PIC out of the circuit, placing it in a programmer and so on.

The biggest disadvantage of an umbilical cord is that it gets twisted when the AGV rotates. A radio-frequency link could eliminate the problem with twists but then an onboard battery is needed, as well as extra space for the transmitter and receiver.

Genetic Algorithms are time consuming. Unlike in this experiment, D. Floreano implemented Genetic Algorithms directly on a robot and not on a simulation, thus necessitating a power supply that could make a robot run for long periods of time [34]. This is an example of where batteries were insufficient. Instead, an umbilical cord was mostly used for power purposes. For future use of this AGV, it was decided to also use an umbilical cord.

5.4 The Physical AGV Used

The Physical AGV is built to evaluate the algorithms generated by the Simulated AGV. The design of the AGV is based on M. Barratt's *Wall Follower* [51] [51]. The differences between the *Wall Follower* and the AGV are:

- No onboard batteries are used.
- An umbilical cord is connecting the robot to a computer and power supply.
- All instructions for the AGV come from the computer, and the physical PIC on the AGV basically functions as a MUX, thus, the parallel information from the sensors are sent through serially, as well as the serial information from the PC, which is made parallel is sent to the interface of the motors.
- The Distance Sensors show in four directions all placed ninety degrees apart.
- In addition to the Distance Sensors, whiskers were added at the corners of the AGV.
- Because of their strength and slow speed, modified servomotors are used as motors to move and steer the AGV.

5.5 Protocol between AGV and Computer

The distance between a sensor on the AGV, and the nearest obstacle to the sensor are determined by the onboard PIC microcontroller. This numeric digital information is directly sent to the Personal Computer through the umbilical cord as shown in **figure 5.8.**

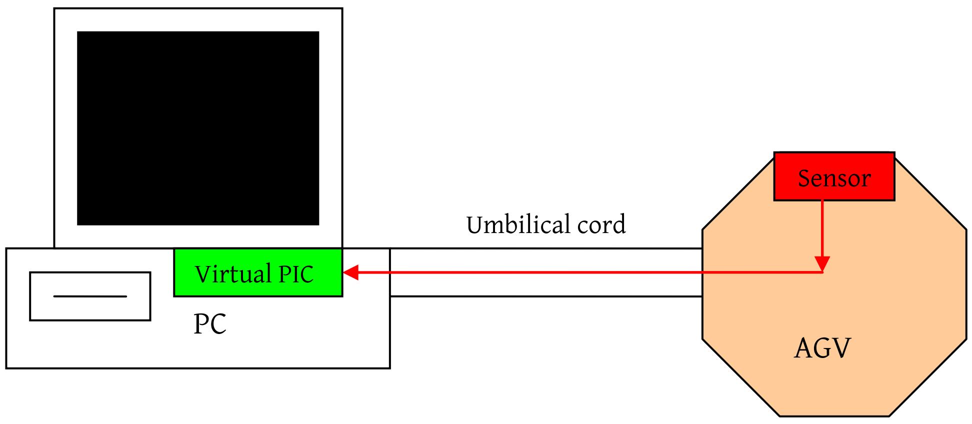


Figure 5.8 Sensor Information to Personal Computer

The information from the sensors is then fed into a virtual PIC processor inside the Personal Computer. This virtual PIC processor processes this information as if it was a physical PIC onboard the AGV. The resulting information from the virtual PIC is then sent through the umbilical cord to the AGV and ends up at the motors as shown in **figure 5.9**.

There are three links between the robot and the computer:

- Link one is to transmit commands so that they end up at the motors.
- Link two receives readings captured by the sensors.
- And the third is a clock signal from the PC to the AGV to synchronise the sending and reading of the individual bits. The rate at which the bits are sent though can easily be changed by adjusting the programme on the PC.

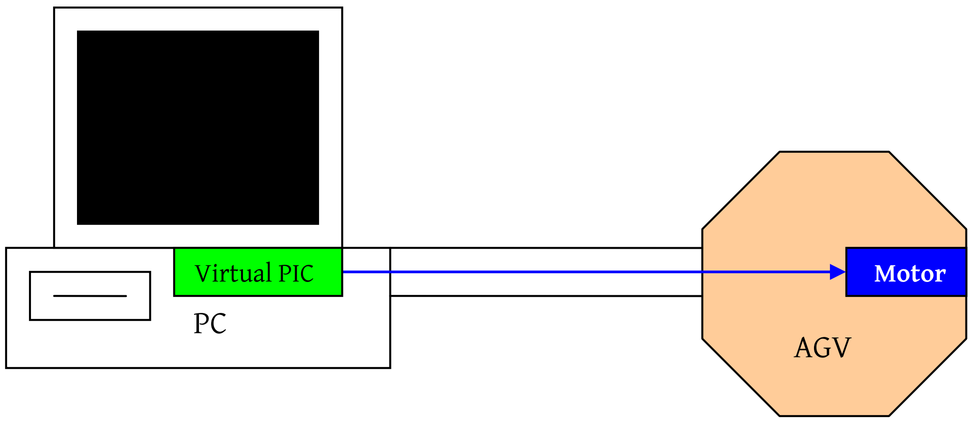


Figure 5.9 Information from Personal Computer to Motors.

Although the clock signal does not determine which bit is sent and received, it is responsible for determining when a bit is sent and received. Without this common clock signal, the PC and PIC microcontroller's internal clock timing has to be as close as possible to each other, and it has to be brought into sync, using a certain method, as many times as possible. By using a common clock signal, the tempo of throughput of bits could easily be changed. Therefore, using the common clock signal simplifies the system.

There are two reasons for making the PC responsible for the clock signal. Firstly, because the PC has to do many more calculations, it is mostly slower at sending through bits than the PIC is. Secondly, the rate at which the bits are sent through can be altered on the PC if needed and, thus, makes it more user-friendly. Both the rising and the falling edge of the clock signal are used as an indication that a new bit is sent and received.

The frame the PIC is sending out consists of forty bits. When a bit is received by the PC, one must question which bit it is and from which sensor is it coming. The first eight bits of the frame coming from the PIC are always the same and are known as the Header. The PC waits until it recognises this specific pattern. During this time no information is sent from the PC to the PIC.

The PIC sends out the bit following the Header as the most significant bit of the sensor known as *Sensor One*. The PC receives this bit and, because it has received the frame word, knows that this must be the most significant bit of sensor one. The PC places this bit's information in the most significant position of a register, known as "Sensor 1". At the same time the PC sends out a bit to the PIC. The PIC places this bit in a receiving register.

Following the most significant bit, the PIC sends out the second most significant bit. The PC knows the bit to follow must be the second most significant bit of *Sensor One* and places the information in the second most significant position of register "Sensor 1". This process is repeated until all eight bits of *Sensor One* have been placed in the register "Sensor 1" inside the PC. During this time, three more bits are sent in the opposite direction to the PIC. For the last four clock-pulses, the PIC ignores the incoming data.

The bit to follow is the most significant bit of *Sensor Two*. The same procedure that was followed for *Sensor One* was then followed for *Sensor Two*. After *Sensor Two*, *Sensor Three* and *Sensor Four* follow. In the opposite direction, the PIC is ignoring all incoming data from the PC.

After the completion of *Sensor Four* and, thus, the whole frame, the PC starts waiting for the Header again. The wait will not be long because the PIC will directly follow with the exact same Header as before. The whole process is repeated.

This Header (the first 8 bits) must always be "01011010". The first header that was tested was "01010101", but the computer confused it with the clock signal when there was a fault on the AGV. The next 8 bits is "channel one" and contain the information for the first sensor. The four channels, one for each sensor, and the header, all consisting of 8 bits, make up the forty bits.

A single variable is used in both the computer and the PIC to determine which bit is being transmitted and received. For example, if the variable in the computer is 24, then the value in the variable of the PIC has to be 24 and the bit, sent in both directions, must be the 24th bit. If these two variables are in sync because of the Header of just one direction, then just one Header is needed to put both directions in sync. The only information going to the PIC on the AGV from the personal computer is four bits. These four bits are used to indicate how the motors must react. If the PC and PIC are in sync, all that is needed is to send these bits at the right time.

The frames are as follows:

01011010AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD   from PIC to PC

XXXXXXXXXXEEEEXXXXXXXXXXXXXXXXXXXXXXXX   from PC to PIC

A: Distance indication from *Sensor One*.

B: Distance indication from *Sensor Two*.

C: Distance indication form *Sensor Three*.

D: Distance indication from *Sensor Four*.

E: Information coming from the PC to ultimately control the motors of the AGV.

X: The PIC ignores these bits.

5.6 Summary

The Physical AGV is built to prove that the programme generated by Genetic Algorithms can be used on a physical AGV. It is also built for future use on other experiments. For this reason, an umbilical cord gives power to the AGV. Having previously been used for power, this cord also connects the AGV to the laptop in order to communicate with the AGV. All the "thinking" is done by the laptop, while the AGV mostly acts as a slave.

Because of its small size, simplicity and low cost, infrared sensors are used. The disadvantage is that the output is not linear to the distance it is detecting.

Modified servomotors, normally used in remote-controlled model aircraft, are used as motors that turn the wheels. They are relatively small, strong and have a slow maximum speed. These qualities provide short time delays from standstill up to maximum speed. The slow speed gives the controller, in association with the umbilical cord, more time to react.

The controller is not the laptop, but a virtual controller programmed into the laptop. This is an exact replica of the virtual controller used in the Simulated AGV. This controller, known as the Simulated-Functions-of-the-Microcontroller is explained in greater detail in chapter 11.

6 Overall Software Design

Three programmes were written:

- Virtual Programme for generation of a Simulated Programme through simulation of the AGV's movement.
- Connection Programme, which utilises the Simulated Programme in order to control the Physical AGV. This programme, which is written for a PC, communicates with the Physical PIC via an umbilical cord and a set protocol.
- The programme for the PIC Microcontroller onboard the AGV is responsible for controlling the motors, getting information from the sensors and communicating with the PC.

In some of the applications, DOS was the better operating system and, in other applications, Windows was better, necessitating the use of both as operating systems for different applications. The programme written in C++ for Windows was more the simulator and the programme generator and the programme written in C++ for DOS was used to control the Physical AGV.

To save on computational time, as few procedures or functions were included as possible. This will lessen the overheads. This, however, is not without disadvantages as it makes the programming more difficult.

The Simulated-Functions-of-the-Microcontroller is controlling the Simulated AGV, as well as the Physical AGV.

6.1 Comparing DOS to Windows for this Application

All programmes were executed on a Pentium II computer. As an experiment, two programmes were written. The one was written in Turbo C++, using DOS as the operating system, and the other was written in Turbo C++, using Windows as the operating system. The programme written for the DOS domain was reacting in real time, while the programme written using Windows as the operating system was tested and the maximum speed that was accomplished was more than two seconds to send through one frame in one direction. This was accomplished when two bits were sent through before the screen was updated once. The bits had to be spaced some time apart because they were too close together to be able to connect to the Physical AGV.

In another attempt two programmes were running simultaneously, where the one was supposed to send information, via some means, to the other. The one programme was written using Windows as the operating system and the other was written using DOS as the operating system. These simultaneously running programmes were slowing each other down so much that it was the worst results of all the experiments conducted.

The best results would be to either do everything in the DOS domain or to save the results determined using Windows as the operating system and then use those results in the programme running in the DOS domain. The latter is used in this dissertation.

6.2 Summary

Three programmes had to be written:

- Virtual Programme generates the Simulated Programme.
- Connection Programme controls the AGV from the laptop.
- The programme of the PIC microcontroller onboard the Physical AGV.

Because of its user-friendliness, the Virtual Programme was written in C++ Builder for Windows, while C++ in the DOS domain gave a higher throughput to the parallel port and, thus, was used for the Connection Programme.

7 Virtual Programme

In the following chapter, the layout of the Virtual Programme will be discussed. This chapter will show how the simulation was made, that includes the surroundings as well as the moving Simulated AGV. This shows the way in which Simulated Programmes were generated. The flowcharts of the Virtual Programme are included.

The complete layout of the Virtual Programme is shown in **figure 7.1**. The result generated by the Virtual Programme is a Simulated Programme that can be saved as a text file. This text file can be read by the Connection Programme to test it on the Physical AGV. The programme is written in C++, with Windows as an operating system, to utilise the graphics interface and user acquaintance with the milieu. The programme consists of the following segments:

- An evaluation of the predetermined Simulated Programme is made by simulating the AGV and its surroundings. The result is the fitness level of the Simulated Programme.
- The Programme Generator generates the programme for the AGV that has to be evaluated. The fitness level is used in the transformation of the current Simulated Programmes into new Simulated Programmes.

The Virtual Programme alternates between the Programme Generator and the Evaluator. An information loop is formed, not only because of the alternations, but also because of the flow of information, as shown in **figure 7.2**.

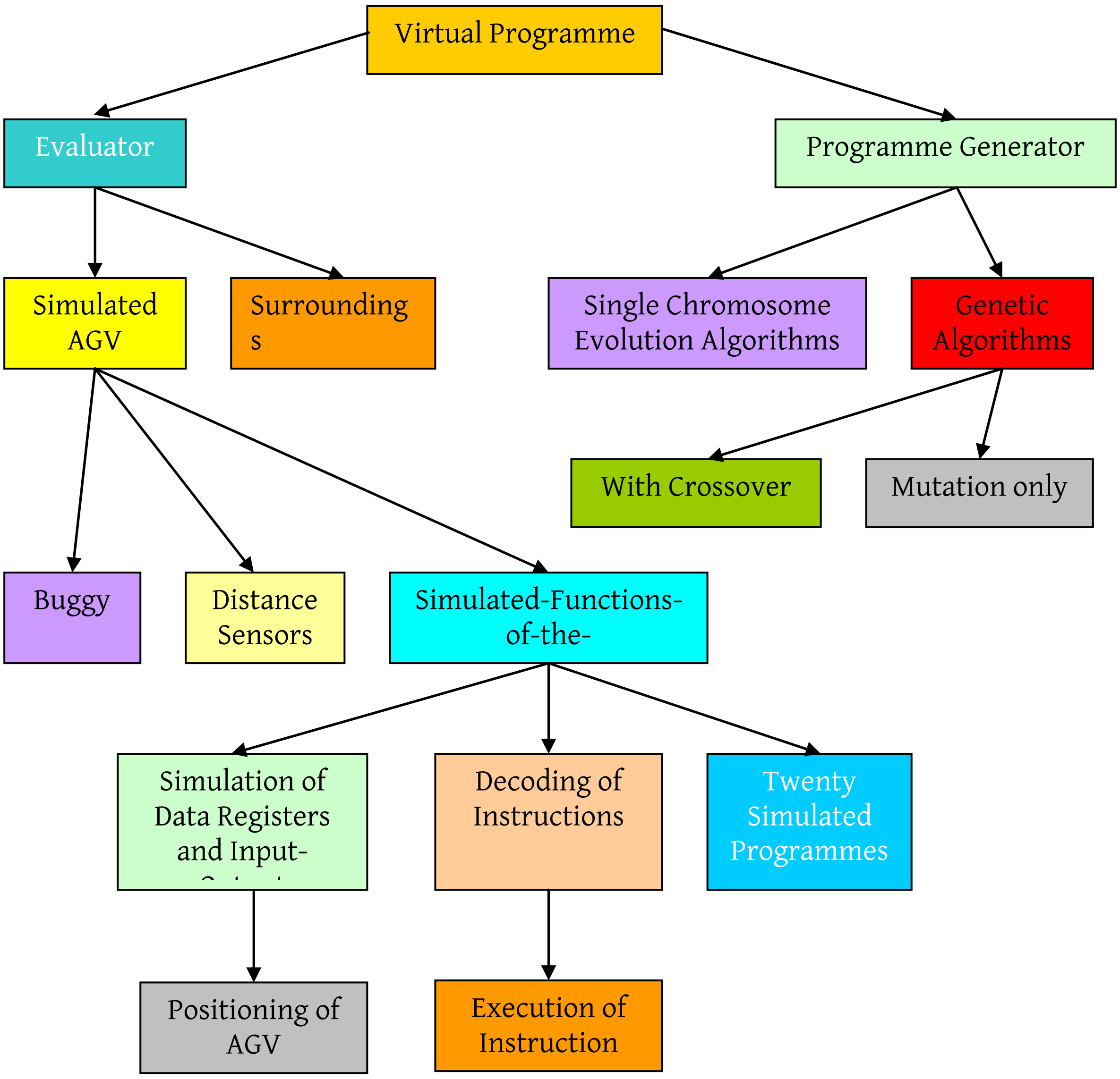


Figure 7.1 The Total Layout of the Virtual Programme

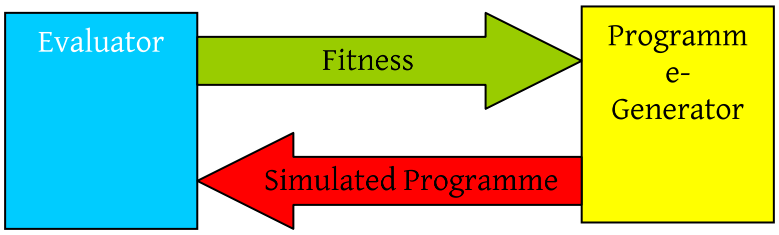


Figure 7.2 Interaction between Programme Generator and Evaluator

The fitness level in the Virtual Programme is set to the inverse of the shortest linear distance between the AGV's current position and the destination. In other words, the distance the "crow would fly".

7.1 Evaluation of the Simulated Programme

The evaluation of the Simulated Programme consists of the following segments:

- A Simulation of the AGV is made up out of the same components as the Physical AGV.
- The surroundings are different types of two-dimensional mazes.

The only connection between the surroundings and the Simulated AGV is the input though the Distance Sensors. The readings from the sensors are altered by changing the placement of the AGV with reference to the surroundings, as shown in **figure** 7.3.

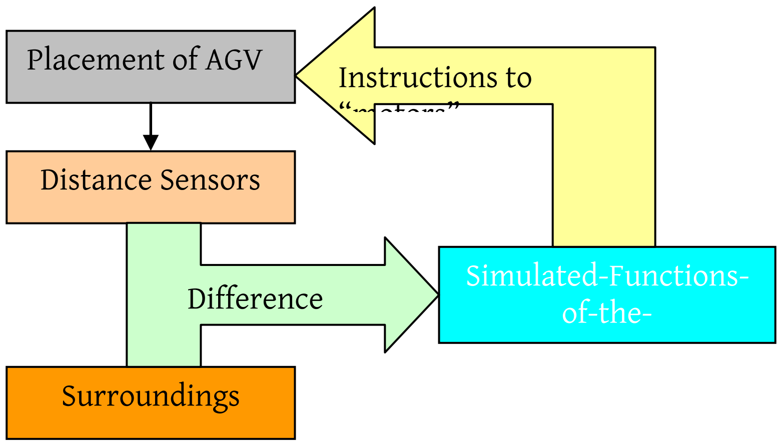


Figure 7.3 Interaction between Distance Sensors and the Surroundings

Because the Physical AGV moves in only two dimensions (forwards, backwards, left and right but not upwards or downwards), the Simulated AGV is placed in a two-dimensional virtual space. The evaluation is incorporated to determine the fitness of a chromosome or Simulated Programme. The AGV's movements are simulated in different situations with the maze and evaluated according to the Simulated Programmes performance. The evaluation is accomplished at the execution point of the Simulated Programme in the Virtual Programme. For the purpose of testing the Evaluator part of the Virtual Programme, a Simulated Programme was written for the AGV, whereby the effectiveness of the Evaluator, as well as the generated Simulated Programme, proved to be satisfactory.

7.1.1 AGV

The objective is to have the same characteristics for both the Simulated AGV as well as the Physical AGV. Before hardware was built, a software simulation was created. This was implemented to test theories and to make changes to the programme that would lead to changes in the Physical AGV. This process saved a lot of money and time. The Simulated AGV's procedure was written in the same consecutive order that the Physical AGV was built in.

7.1.1.1 Movement of AGV

The AGV reacts to a four-bit word coming from the procedure that simulates the functions of the microcontroller, as described in **table 7.1**.

The motors do not have to be simulated. The output from the Simulated-Functions-of-the-Microcontroller just has to make the AGV react in the same way it would in the real world.

Table 7.1 Binary Code Movement Buggy

Binary	Decimal	Movement	Classification
0000	0	No movement	No movement (no wheel is turning)
1010	10	Forwards	Wheels are moving in the same direction (Wheels are turning in the same direction)
0101	5	Backwards	
0110	6	Turing left	Revolving AGV (Wheels move in opposite directions)
1001	9	Turing right	
0001	1	Right and backwards	Moving forwards or backwards while turning (One Wheel is turning and the other is standing still)
0010	2	Left and forwards	
0100	4	Left and backwards	
1000	8	Right and forwards	
0011	3	Illegal operation	Illegal operation
0111	7	Illegal operation	
1011	11	Illegal operation	
1100	12	Illegal operation	
1101	13	Illegal operation	
1110	14	Illegal operation	
1111	15	Illegal operation	

Two bits can be used per motor. The illegal operation, shown in **table 7.1,** where both bits of a motor are one, can be changed to the high-speed forward motion by changing the current through the motors. However, this is not implemented because of the following reasons:

- This alteration in speed can also be accomplished by pulsing the input current to the motors. The Simulated Programme can adapt on its own to pulse the input to the motors, thus changing the current is not necessary.
- AGV exist that are equally comfortable going “forwards” or “backwards” [59]. The AGV will not react in the same way for forwards and backwards motion, when one changes the forward velocity by changing the current to the motors.
- It makes the circuits of the Physical AGV more complex. This also leads to a bigger Physical AGV and, thus, a bigger Simulated AGV with its own set of problems, as explained in section 5.1.1.

The procedure that simulates the movement of the virtual AGV is split into two sections:

- Moving Graphics - this makes any change in graphics possible
- Reaction to Input - this part decodes the information coming in from the procedure Simulated-Functions-of-the-Microcontroller (the part that functions as the controller) to make different movements possible for the AGV

Both parts interact with one another, and are placed in series with each other, in one procedure.

Moving Graphics was the first part that was completed for the Virtual Programme and represents all virtual objects. It is the foundation of the whole Virtual Programme.

In the process of generating a programme of this sort, many aspects had to be simulated. If the graphics are less complex, the whole process would be faster, thus only a moving line representing the axel of the AGV is drawn.

Making the graphics move is accomplished in the same way as accomplishing movement in the cinemas or on television. For example, a line is drawn, erased, and then redrawn a little higher, as in **figure 7.4**. If this process is performed at a fast pace, the result is a line that looks as though it is moving upwards. This explains, the need for a new drawing each time the screen was cleared, as shown in **figure 7.5.**

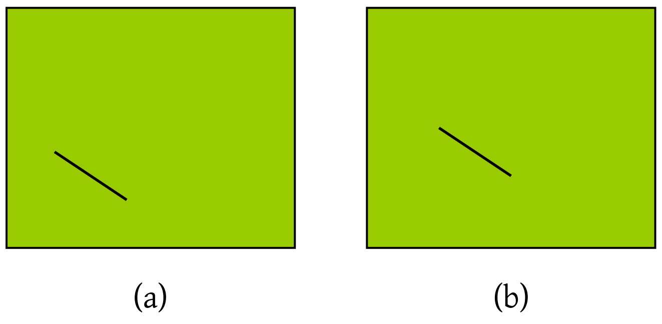


Figure 7.4 Simulating Movement (a) First Position (b) Second Position

The sequence has to be placed in such a way so as to let the calculations be executed while the graphics are on the screen, as shown in **figure 7.5**. If the calculations followed the clearing of the screen, the screen would be blank for a longer amount of time leading to more flickering of the screen.

Three variables, one for the X-axis, one for the Y-axis, and one describing the direction of the object, are necessary to describe the position of an object in a two-dimensional space.

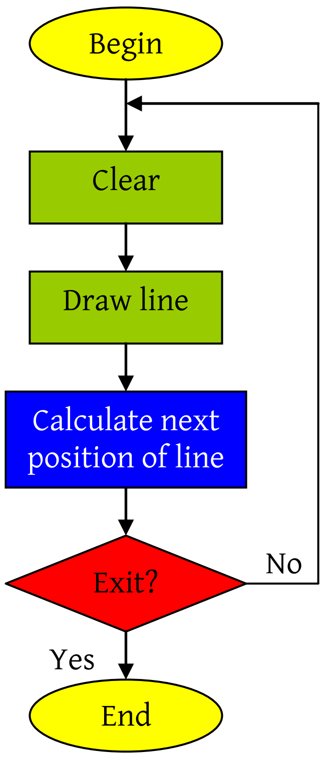


Figure 7.5 Flowchart for Moving Graphics

A line, representing the AGV, is drawn between two points. Each point has two coordinates thus creating the need for four variables that depict the line. The X-position, Y-position and the direction of the AGV, have to be represented by these four variables. Consequently, the last four variables have to be used by the drawing procedure as well as the calculating functions, and, thus, were declared global, which means that it must be accessible to all sub-procedures.

Figure 7.6 shows the axle of the AGV, its current reference position and the next reference position of the AGV. Angle (a) is the angle at which the AGV is displaced from virtual north, where virtual north is straight up. (AB) is half the distance between point (C) and (B). D is the distance of one step forward. The following formulas determine the positions of point C and B so a line, that represents the axle of the AGV, can be drawn:

X coordinate of B = Current X position + AB x cos (a)

Y coordinate of B = Current Y position + AB x sin (a)

X coordinate of C = Current X position – AB x cos (a)

Y coordinate of C = Current Y position – AB x sin (a)

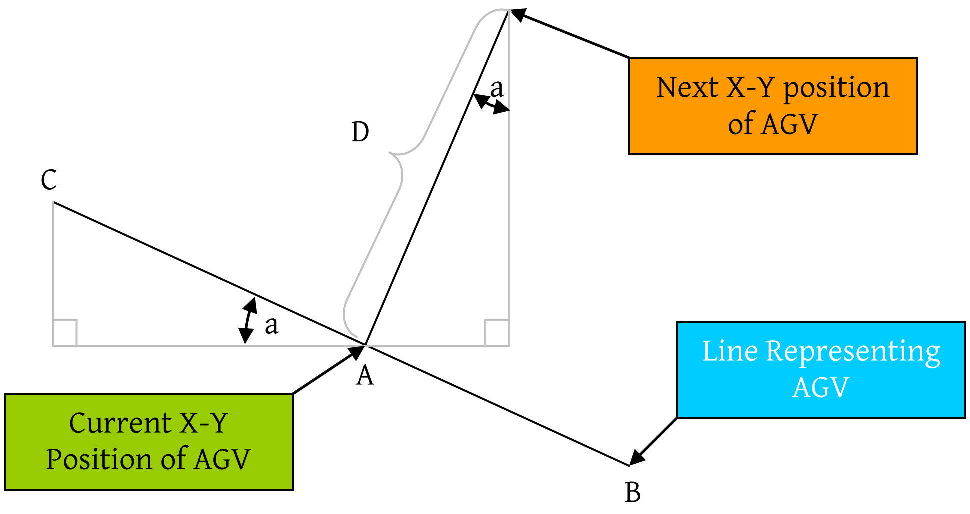


Figure 7.6 Calculations of Next Position of AGV

Forward movement is in right angle to the line that forms the axle of the AGV. The X and Y coordinates of the next position are needed to make the AGV move forward. Multiplying the distance of one step (D), the distance travelled while one instruction is executed with the cosine and sin, with regard to the direction of the AGV (A), makes the AGV go in the direction it is facing in, as shown in **figure 7.6**. The next position of the AGV, if it was to move forward by one step, is calculated as follows:

Next X position = Current X position + D x sin (a)

Next Y position = Current Y position – D x cos (a)

It is cosine and sin for the same angle, as was needed to draw the line; the positions of cosine and sin just have to be changed with respect to the X-axis and Y-axis. If the signs are not changed in the correct way, the AGV will turn in the opposite direction than what was intended. A line, temporarily drawn at right angle to the axle of the AGV, will show the direction the AGV must move in and a point on the line can be seen as the next coordinate of the AGV, if the AGV was to move forward.

Functions in the vein of sine and cosine are very complex and must therefore take a lot of time to compute. By computing it once, and using it many times, the total amount of computations can be lessened. All that is needed is extra variables that will house the temporary results of the cosine and sin calculations. These variables only have to be used in the sub-procedure that does these calculations, so it can be local.

Backwards motion is accomplished by inverting the signs of the calculations for forward motion.

Changing the reference angle of the AGV will turn it. The smaller the distance between the wheels of the AGV, the more the AGV will rotate for a specific angle turn of the wheels.

When turning while moving forward, the angle turn and the distance of one step is not the same as when the AGV is rotating or when it is just moving forwards.   It is not possible to synchronise the wheels of this AGV with the wheels of the real AGV if one does not know what the dimensions of the real AGV are. One of the biggest problems is that the angle at which the AGV turns is dependent on the distance of one step and the distance between the wheels, as already stated. The turn is supposed to look like the example in **figure 7.7(a) or (c).**

In all the sketches in **figure 7.7**, (B) is the distance of one step and (A) is the distance between the wheels. As can be seen in **figure 7.7,** triangle (d) is closer to arc (c) than triangle (b) is to arc (a). Thus, the smaller the step (B) is in relation to the wheelbase (A), the more the result resembles a triangle.

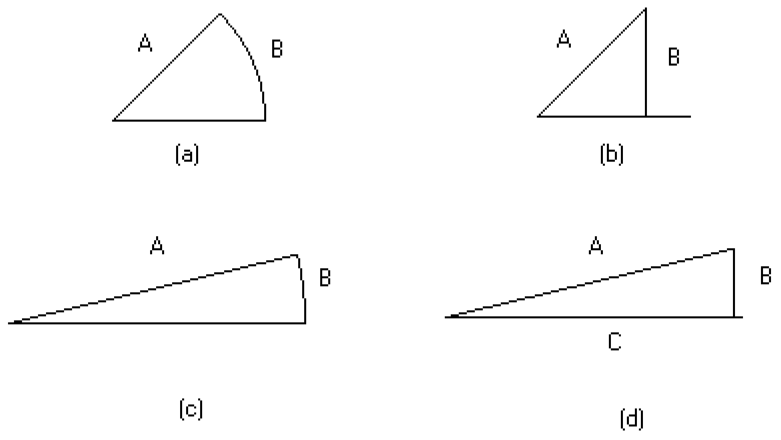


Figure 7.7 The AGV, While Turning, Could Use Either Triangles or Circles

The distance of the step is much smaller than the distance between the wheels, so one can use a triangle, as featured in **figure 7.7**(d). All that is needed is to divide the distance of the step by the distance between the wheels and then to get the arc tan of that result as follows:

Angle of rotation = Arc tan (B/A) (7.1)

The calculation could even be made easier, and more correct, with fewer computations. The radial angle is defined as the distance of the round edge divided by the radius of the circle, as evident in **figure 7.7**(c). Formula (7.1) is now changed to:

Angle of rotation = B/A

When the AGV is rotating, the length of the radius would be half that of the distance between the wheels. If the one wheel is standing still and the other is moving, then the AGV will move forward while turning. The radius for determining the angle will be the same as the total distance between the wheels. The two triangles in **figure 7.8** are identical except for their difference in sizes, thus, the following formula can be incorporated:

$a/A = b/B$

But $b = B/2$

$a/A = (B/2)/B = 1/2$

$a = A/2$

Where:

a = Distance of one step when turning.

A = Distance one wheel travels for execution of one instruction. The distance is the same as when the AGV moves forward during the execution of one step.

B = Distance between wheels

b = Distance from one wheel to the middle of the AGV.

The forward motion while turning would be approximately half that of forward motion while the AGV is not supposed to turn, as shown in **figure 7.8**.

Although the ideal is that the AGV moves forward while turning, it can be simulated by first letting the AGV move forward for a while and, thereafter, letting it turn without moving forward. The AGV, when moving forward and, thereafter, turning (**Figure 7.9** (b)), does not give exactly the same results as when the AGV is turning whilst moving forward (**figure 7.9** (a) and **table 7.1**). The smaller the steps are (**figure 7.9** (c)), the closer the result is to the ideal (**figure 7.9** (a)).

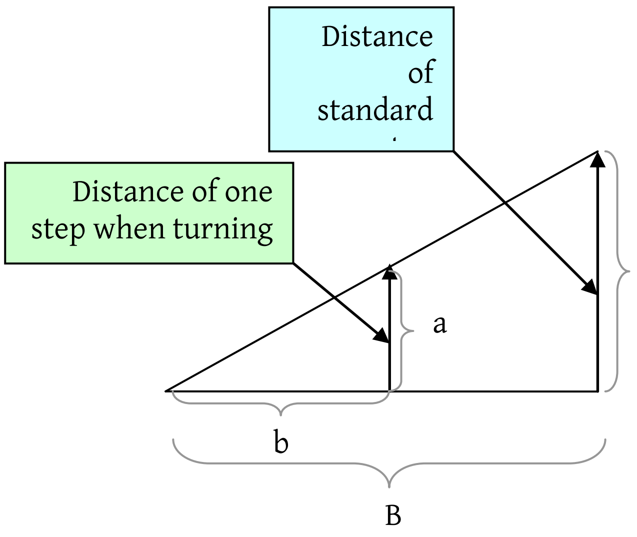


Figure 7.8 Calculating the Amount of Forwards Motion when Turning

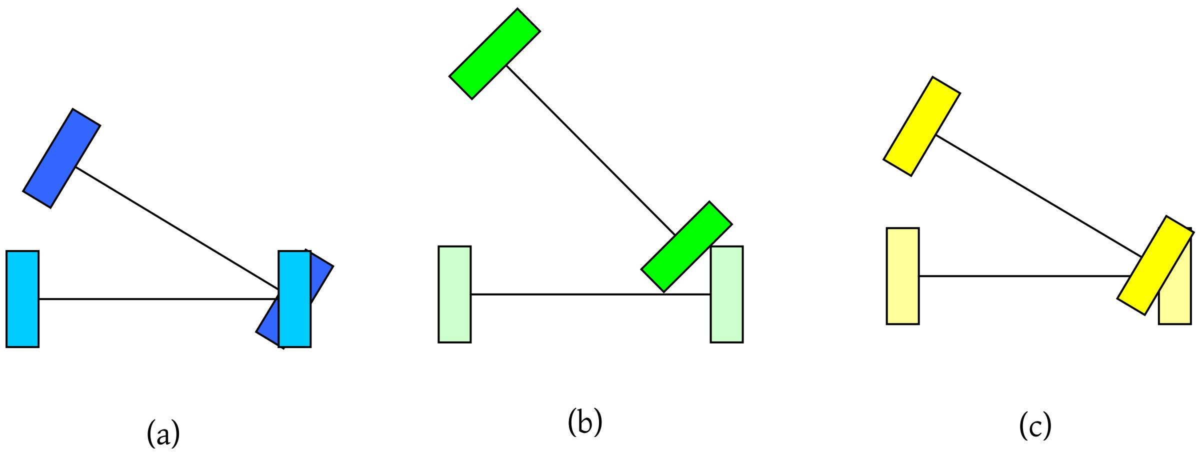


Figure 7.9 (a) Ideal Options when One Wheel Standing Still (b) Move Forward and then Turn (c) Smaller Steps Forward and then Turn

Calibrating the Physical AGV with the Simulated AGV is accomplished by turning the Physical AGV around on its own axle for two or more full rounds and then counting the amount of instructions executed during that time. It is then a simple division calculation that determines the angle the Simulated AGV has to turn for each instruction executed.

7.1.1.2 Distance Sensors

The calculation of distance is important for two reasons. The one is to let the Simulated-Functions-of-the-Microcontroller know more about the AGV's surroundings so the AGV will know how to react. Secondly, the Programme Generator uses the sensors to determine if the AGV has collided with something in order to let the next Simulated Programme be evaluated, or to let a new generation of Simulated Programmes be generated. The results from the Distance Sensors are placed directly into the following registers of the Simulated-Functions-of-the-Microcontroller:

- The sensor facing forwards' results are placed in register five.
- The sensor facing backwards' results are placed in register six.
- The sensor facing right's results are placed in register seven.
- The sensor facing left's results are placed in register eight.

The simulated Distance Sensors and the physical infrared sensors characteristics have to, as far as possible, be the same to be able to give the best results. Accomplishing this sometimes requires more that just calibration. In this dissertation, the linear readings from the virtual world have to be changed to a non-linear status to correspond with the real world readings.

Four aspects have to be considered:

- Placement of the Simulated Sensors

- Determining the Distance to an Obstacle
- Changing Linear into Non-Linear
- Uploading and Downloading Times

The placement of these sensors with respect to the surroundings shows not only in which direction the sensors are facing but also the starting point from where the distance is calculated.

From the reference point of the AGV, a sensor is displaced by an angle, as shown in **figure 7.10**. To be able to determine the angle of the sensor with reference to Virtual North, one must add the reference angle of the AGV to the displacement angle of the sensor with respect to the AGV. The displacement angle of the sensor, with respect to the AGV, is fixed because the sensor is mounted in a fixed position on the Physical AGV. Although the displacement angle of the sensor, with regards to Virtual North, is not the same as the Reference Angle of the AGV, it will change by the same amount, and in the same direction, as the reference angle of the AGV.

.

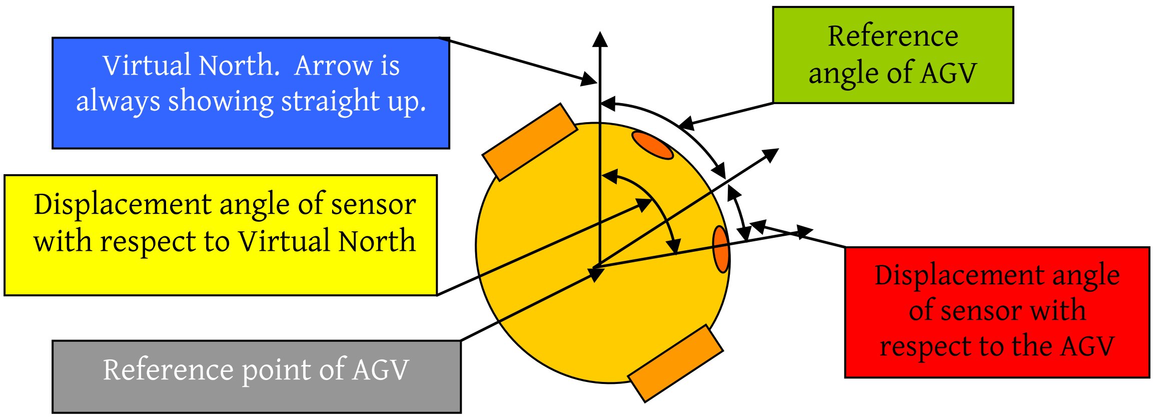

Virtual North. Arrow is always showing straight up.
Reference angle of AGV
Displacement angle of sensor with respect to Virtual North
Displacement angle of sensor with respect to the AGV
Reference point of AGV

Figure 7.10 Determining Position of Sensor

The total distance between the reference point of the AGV and the sensor is multiplied by sin and cosine of the angle of the sensor, with reference to Virtual North, to get the X and Y coordinates of the sensor with respect to the reference point of the AGV. The X and Y values of the AGV's reference point just have to be added to the X and Y values of the sensor to determine where the sensor is in the maze. Any point on the AGV can be determined in this way and, thus, one can draw the AGV in any form that one wants, if need be.

The angle the sensor is facing in might not be the same as the angle the sensor is displaced by, with reference to the AGV, as shown in **figure 7.11**. This constant difference can be added to, or subtracted from the displacement angle of the sensor, with reference to Virtual North. The result is the angle the sensor is facing in with reference to Virtual North. Computations can be lessened by making the displacement angle of the sensor, and the angle the sensor is facing in, the same.

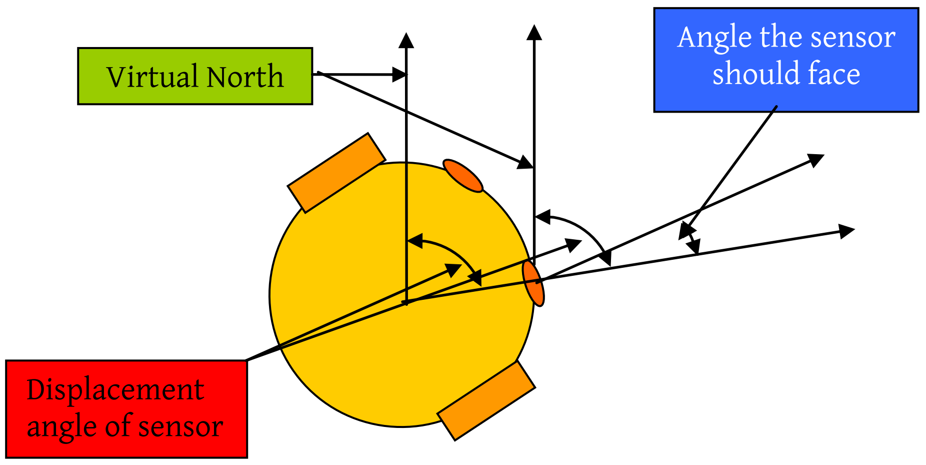
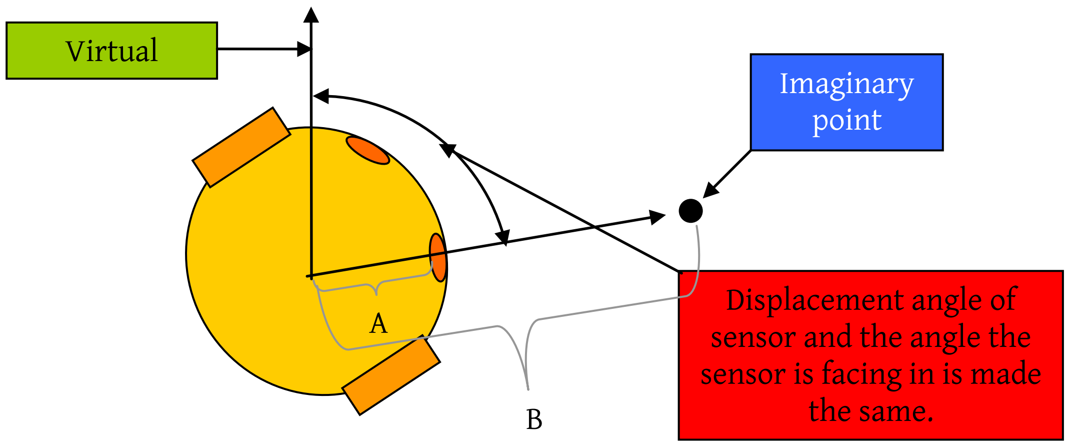


Figure 7.11 Determining the Direction the Sensor Should Face

When these two angles are the same, as is the case in this experiment, an imaginary point can be calculated directly from the AGV's reference point, as can be seen in **figure 7.12**. The distance from the reference point of the AGV to the sensor (A) just has to be subtracted from the input value obtained from the virtual sensor (B).

Calculations could be made even less complex by using the cosine and sine already determined for the drawing and forward motion of the AGV. Preceding a single step of the AGV, exactly the same calculation as the calculations performed to determine the next position of the AGV if it is moving forward, is performed to determine an imaginary point some distance directly in front of the AGV. If this imaginary point is not inside an obstacle, the distance is made larger and a second imaginary point is determined. This process is repeated until an obstacle is registered or until the distance becomes too far. This simulates a single sensor in the middle of the AGV's front, facing directly forwards.



Virtual
Imaginary point
Displacement angle of sensor and the angle the sensor is facing in is made the same.
A
B

Figure 7.12 Less Computations Option for Sensors

A backwards sensor can be simulated by simply inverting the signs of the calculations, as was implemented for the front sensor, or made to be similar in nature to backwards motion. The calculations for drawing the line that simulates the axle correspond to the calculations needed for the sensors facing to the right and left of the AGV. The imaginary sensory points are the same as the points needed to draw the line making up the axle if the line was to be made longer.

A loop in the programme is used, where a few tests are performed with every cycle. Firstly a point is determined. These tests will check to see if the point determined is inside one of the obstacles and each obstacle has its own test.

A loop can be programmed for each sensor. The result will be that the multiplication with sin and cosine will have to be performed for every sensor whenever distance is determined. Instead, one programme loop can be used. The calculations have to be executed once for a specific distance. The detection of an object will be like a circle moving outwards from the AGV.

However, the disadvantage of this method is that, if one looks at a single sensor, tests have to be carried out after the sensor has sensed an obstacle to make it possible for the other sensors to also determine distance. To overcome this, an "IF" function can be incorporated to do a test and see if it is necessary to execute all these tests again for a specific sensor. These tests also depend on the amount, as well as the complexity, of the obstacles. This makes the programmes execution time longer, and even worse than that is that it is in a programme loop, so the test will be executed again and again.

If only one loop is used, one must make sure that all the sensors' results have been met as well. For this, an extra test has to be performed. If one considers all the tests that have to be executed versus two single multiplications that have to be executed, then it seems that the best option would be to use a loop for every sensor.

The results showed that the AGV bumped into obstacles when only one forward-facing sensor was incorporated. A method to overcome this might be to use more sensors or to change the existing sensor's placement on the AGV. There are two options available:

- Parallel Simulated Sensors can be placed at the corners of the AGV which faces forwards. Only an obstacle sensed that is the closest between the two will be forwarded to the processor. An addition and multiplication calculation of the original calculated cosine and sine vectors makes up these vectors as shown in **figure 7.13** (a). This method is used in this dissertation.
- Diagonal Simulated Sensors facing forwards, as the ones on **figure 7.13** (b), need extra angle calculations, thus cosine and sine have to be calculated for each angle. Adding vectors to each other takes up less time than calculating cosine and sine for vectors facing in new directions.

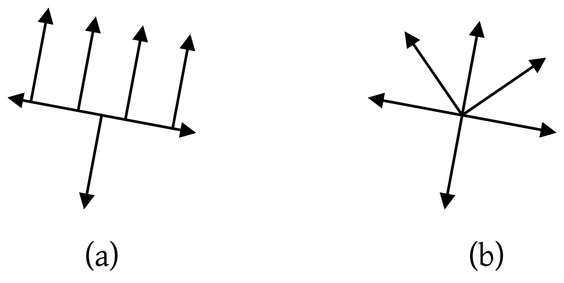


Figure 7.13 Options in Placement of Sensors on the AGV (a) Parallel Sensors (b) Diagonal Sensors

Out of the preceding parts, it can be seen that determining a distance with a linear result is easier than determining distance with a non-linear result. To make the simulated sensor output values non-linear, thus more like the physical sensors output, a formula had to be incorporated. The formula has to be something resembling the following:

$Y = A/X$

A: Some predetermined constant value.

X: distance from the obstacle.

Y: value generated by the sensor.

The output graph has to be moved with relationship to the X and Y-axes, so a value has to be added to the X, and to the Y, value in the formula. The formula, thus, has to

look as follows:

$Y = A/(X + B) + C$

Y: Value generated by the Analogue-to-Digital-Converter.

A: This constant determines how steep the climb of the formula is.

X: Distance to the obstacle in pixels.

B: A constant determining the displacement on the X axis.

C: A constant determining the displacement on the Y axis.

As an experimental set-up, four measurements are taken using the Physical AGV. The readings are taken while the obstacle is at an angle to the robot so that the specular reflection is not made part of the results [57, p.98-99]. For the forward facing sensors, the readings were as follows:

249	=>	13.5 cm
126	=>	19.0 cm
15	=>	66.5 cm
54	=>	30.5 cm

Because there are three unknown constant values that have to be determined, only three of the four values are used in substitution. The two outermost values and the most central value are used. Firstly, 249 is substituted in Y and 13.5 in X, as shown in the next example:

$$Y = A / (X+B) + C$$

$$249 = A / (13.5 + B) + C$$

$$249 - C = A / (13.5 + B)$$

$$(249 - C)(13.5 + B) = A$$

$$A = 3361.5 + 249B - 13.5C - BC \qquad (7.2)$$

In the same way, 15 and 54 are substituted in Y and 66.5 and 30.5 are substituted into X:

$$15 = A / (66.5 + B) + C$$

$$A = (15 - C)(66.5 + B)$$

$$A = 997.5 + 15B - 66.5C - BC \qquad (7.3)$$

$$54 = A / (30.5 + B) + C$$

$$A = 1647 + 54B - 30.5C - BC \qquad (7.4)$$

Because A is constant, formula (7.2) could be made equal to formula (7.3):

$$3361.5 + 249B - 13.5C - BC = 997.5 + 15B -66.5C -BC$$

$$2364 + 234B +53C = 0$$

$$53C = -2364 - 234B$$

$$C = - (234B + 2364) / 53 \qquad (7.5)$$

Formula (7.2) is now made equal to formula (7.4):

$$3361.6 +249B - 13.5C -BC = 1647 + 54B - 30.5C - BC$$

$$17C = - (1714.5 + 195B)$$

$$C = - (1714.5 + 195B) / 17 \qquad (7.6)$$

Because C is a constant value, formula (7.5) and formula (7.6) can be made equal to one another:

$$- (234B + 2364) / 53 = - (1714.5 + 195B) / 17$$

$$(234B + 2364) * 17 = (195B + 1714.5) * 53$$

$$3978B + 40188 = 10335B + 90868.5$$

$$-6357B = 50680.5$$

$$B = -7.97239$$

By substituting the value of B into formula (7.6), the value of C can be determined:

$$C = - (1714.5 + 195B) / 17$$

$$C = - (1714.5 + 195(-7.97239) / 17$$

$$C = - 9.404$$

Both the values of B and C are needed in formula (7.2) to determine the value of A:

$$A = (249 - C) (13.5 + B)$$

$$A = (249 + 9.404) (13.5 -7.97239)$$

$$A = 1428.3$$

All these values are now inserted into the initial formula in order to get the formula that changes the linear value into a non-linear value that is calibrated with the real AGV's sensors:

$$Y = 1428 / (X - 7.97) - 9.4$$

The fourth reading is used to prove how accurate the formula is.

$126 = 1428/(X - 7.97) - 9.4$

$135.4 = 1428/(X - 7.97)$

$X - 7.97 = 10.546$

$X = 18.516$ cm, where it should have been 19.0 cm. It is a 2.6% error.

The width of Maze 1 and 2 is 600 pixels. Making the maze small enough accommodate a car, it was decided that if the maze was physically constructed, it had to be two meters in length. Thus, a distance of 3 pixels represents one centimetre. Because X was made 3 times as big as it was before, the value of 1428 and the value of 7.97 both have to be multiplied by 3 to get the same results from the formula. External influences make the generated output value from the Analogue-to-Digital-Converter drift by a reading of approximately 3, so the addition of a random number of 3 simulates "noise" and is added to the total formula. The formula has to be altered accordingly:

$Y = 4284 / (X - 24) - 9 + \text{random}(3)$

The formula for the other sensors is determined in the same way and is as follows:

$Y = 1511/(X - 25) - 11 + \text{random}(3)$

The values in the two formulae are so similar that, if it was not for the difference between the value 4284 and 1511, the two formulae could have been identical. Reading errors could have caused the small changes in the varying values. The resistor, in series with the transmitter, thus influences the 4284 and 1511 values the most. Making sure the most accurate results are gathered, the formulae are not made to be more similar, but are used just as they are in the programme.

It is easy to see that, for a formula like Y = 1511/(X-25) – 11, the value of X must not be allowed to be a distance of 25 pixels, because it would cause Y to be infinitely big. If the value of X was a length of, for example, 26 pixels the resulting value generated by the distance sensor would be 1500. The result was that the value was more than 255, and it was assumed that the Simulated AGV had run into an obstacle.

A problem arose when the AGV suddenly came very close to an obstacle. It was so close that the value of X was lower than 25 pixels. For illustration purposes, the value of X is made 17 pixels. The resulting value was –200, which was significantly less than 255, and the controller of the AGV accepted the obstacle to be far away. To overcome this problem, the obstacle hit was registered when the X value was 25 pixels or less than 25 pixels away from an obstacle.

The following resisters in the data array are loaded with the indicated sensors information:

- Register five is loaded with information from the sensor facing forward.
- Register six is loaded with information from the sensor facing backwards.
- Register seven is loaded with information from the sensor facing to the right of the AGV.
- Register eight is loaded with information from the sensor facing to the left of the AGV.

The connection between the Physical AGV and the computer is slow. During the virtual evaluation, the sensors can determine the distance to an obstacle in the time it takes to execute only one instruction. To make the simulation more realistic, the sensors must react as if there was an umbilical cord, which would slow down the communication between the Physical AGV and the personal computer. In chapter five, it was seen that in order for all the sensors information to be updated only once, forty instructions had to be executed. Simulation of the slow connection is accomplished by updating the values of the sensors only once after the execution of forty instructions.

With this slow communication, the motors again are more problematic than the sensors, because even if the Simulation-of-the-Functions-of-the-Microcontroller intend for a motor to change its motion, the AGV must still continue in the same fashion. Simulating this, two variables are used. The one is holding the information generated by the Simulated-Functions-of-the-Microcontroller, and the other is the information to let the "motors" keep on doing what they have been doing. For example, if the AGV is moving forward. The instruction coming from the Simulated-Functions-of-the-Microcontroller is to start turning. The AGV will have to keep on moving forward, because that is the information that is still in the one variable. This will keep on happening until the forty instructions are complete and the instruction to turn is loaded into the other register. The AGV will then start turning.

The reading of the sensors and the loading of the new data into the register for the "motors" occur in the same cycle or at almost the same time. This however, does not mean that when the Simulated-Functions-of-the-Microcontroller changes the output to the motors that it will have to wait for forty cycles. It simply means that the output to the motors, as well as the reading of the sensors, can only take place at specific times. These specific times are forty instructions or forty cycles apart.

With the Physical AGV, the information is not changed instantaneously, but takes forty instructions to change. Thus, the change happens "systematically". The simulations are not totally correct, but adhere to the uploading times and downloading times more stringently and gave good results on the Physical AGV. It could even yield better results than the ones that would have occurred if the simulation were more correct. It will force the Simulated Programmes to adapt to foresee what will happen and react predicatively. Hopefully the Simulated-Functions-of-the-Microcontroller will react directly after the forty instructions have been completed, and not during.

7.1.1.3 Extra Destination Input

The instruction from the user comes in the form of a single bit. With this option activated, the AGV has to go to one of two destinations. If the value is one, the AGV has to go to the one destination and when the value in the bit is zero, the AGV has to go to the other.

In the simulation, a simulated input is randomly generated before each simulation and is placed in the first bit of register fifteen. The input also influences the fitness function, as the fitness levels are made directly proportional to the distance between the last position of the AGV, and one of two destinations in opposite corners of the maze, dictated by the input. The value of the simulated input determines which destination is used for that simulation.

When the user chooses the only one destination option, the user is given an extra option. This option, when activated, uses the first bit in register fifteen to indicate if the destination is to the right or the left of the Simulated AGV. The same bit is used to simplify the generation of the Simulated Programme.

7.1.1.4 Surroundings

The surroundings are not designed to represent any real life situation; they are simply intended to give a medium for testing the ability of the various learning methods. The surroundings play an important role in that they determine the level of complexity that must be overcome. The simulated world has a wall surrounding it and obstacles inside of it. The surroundings would not exist if it were not for the Distance Sensors that have to react to the surroundings. It is possible that the AGV can go through the obstacle course without using any sensors and without bumping into any obstacle. This is similar to a person walking in the dark. He knows how many steps to take and in which direction to take them. The problem is, to be able to "teach" the AGV, it has to bump into obstacles, and these bumps are registered by the AGV's sensors. The interaction between Distance Sensors and the surroundings was explained in section 7.1.1.2.

Although other forms can be incorporated, rectangles are used as obstacles because of their simplicity. The calculations take less time with rectangles and, thus, the simulation in total takes less time. Luckily, there are a lot of options available when using rectangles.

It was decided that a maximum of 10 obstacles would be used in the maze. As will be seen, the width of the passages and walls were factors. It proved itself to be an adequate amount as in some mazes only eight obstacles were needed.

Each obstacle needs four variables. These values are the most left X value, the most right X value, the upper Y value and the lower Y value. Matrixes with ten variables are used for each. For visual purposes only, a drawing of the obstacle course is made. The drawing of the surroundings is easy to do, as rectangles and lines are very easily drawn in C++. The biggest problem is when it must be drawn, or redrawn, as shown under section 7.1.1.1.

These ten obstacles are placed in different forms at different positions. It is up to the user to change the obstacle course into one of the given options. Some of these options are obstacles that change in position and form, while others are set mazes. In all the simulations, the line represents the AGV, and the cross is the destination.

The reason for incorporating different obstacle courses was to determine what problems the AGV could overcome. The mazes also evolved to get a better quality maze. In the end, it was seen that the AGV could not adapt to changing obstacle courses and that the best option was to use maze 2 for all experiments. The options in obstacle courses were:

- *Maze 1* is a fixed option
- *Maze 2* is a fixed option
- *Sliding Columns* changes in form
- *Moving Holes* changes in form
- The *Random* option changes randomly between all the earlier mentioned options

An extra option was given in order to choose whether the obstacle course must change for every simulation or only after every generation had been evaluated.

Drawing a wall in a two dimensional space is accomplished by making the length of a rectangle much more than the width. *Maze 1,* as shown in **figure 7.14,** consists out of nine of the ten rectangles. In section 7.1.1.2, it was seen that a distance of three pixels simulates a length of one centimetre. The passages in the maze are a hundred pixels wide, and the walls are ten pixels wide. *Maze 1* was the first maze constructed, and the walls width was enlarged to ensure that the sensors would be able to sense the walls. The mazes' passages are thus equivalent to a width of 333 mm and the walls are equivalent to a thickness of 33mm. The sketch shown is an actual copy of the maze, and, as can be seen, there are only two routes that the AGV can take to get to the destination.

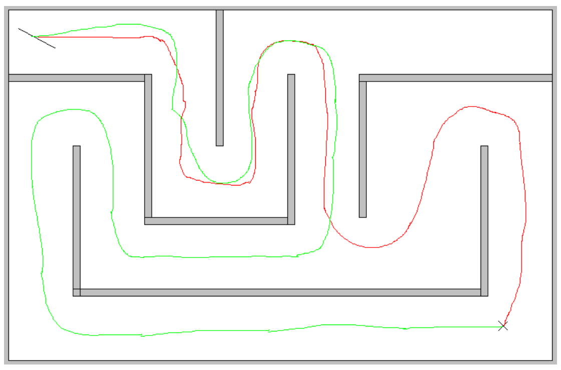


Figure 7.14 Maze One has Two Possible Routes to the Destination

Because of the small passages and thick walls of Maze 1, Maze 2 was developed. In the section "Distance Sensors", it can be seen that the passages must be wider than a certain specified width that was determined by the Distance Sensors. It was not possible to determine distance if the object was closer than thirteen centimetres. Thus, any obstacle thirteen centimetres and closer is seen as being collided with.

Physical walls of 33mm in thickness, if one can get hold of it, will be expensive and it will obviously be heavy. Even in the simulation, the wall took up space, where if it were thinner the passages could be made wider.

Instead of having two possible routes to the destination, as in *Maze 1*, there are three possible routes to the destination in *Maze 2* that were created. The AGV could also walk in wide circles in *Maze 2*, as shown in **figure 7.15**.

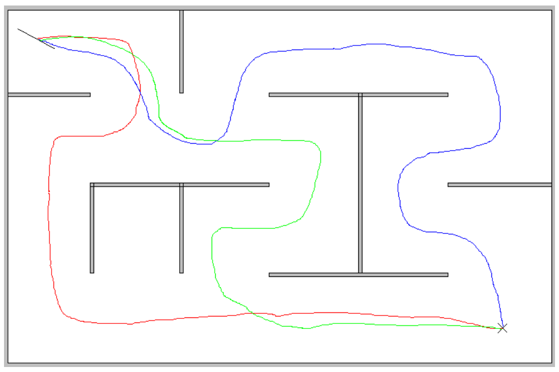


Figure 7.15 Maze Two has Wider Passages, Thinner Walls and Three Routes

The obstacle course "Sliding-Blocks", shown in **figure 7.16**, are made out of equally large, almost square, rectangles. The size and form of all the obstacles is the same. Their individual Y coordinates are determined randomly. Avoiding the obstacles' overlapping and being in one place, each obstacle has been given its own "Column" or, in other words, X coordinate that is fixed. This method also ensures that the Simulated AGV is not isolated from the destination. If all the obstacles were placed in such a way as to form a line, there would still be a gap big enough to let the AGV though. As can be seen, this obstacle course changes into a different form after each evaluation.

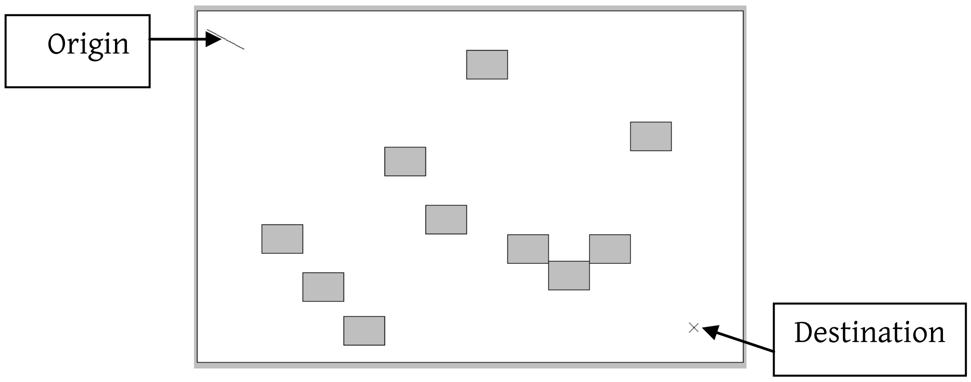


Figure 7.16 Sliding Blocks Obstacle Course

The "Moving Gaps", shown in **figure 7.17**, obstacle course is basically the opposite of the "Sliding Blocks" obstacle course. Instead of having obstacles that move "up and down", there are gaps in a wall that move "up and down". Thus, the placing of a hole is determined randomly.

Two obstacles make up one wall. The one obstacle is to the right-hand side and the other is to the left-hand side of the gap. To make the alleys wide enough, only four walls (eight obstacles) are used.

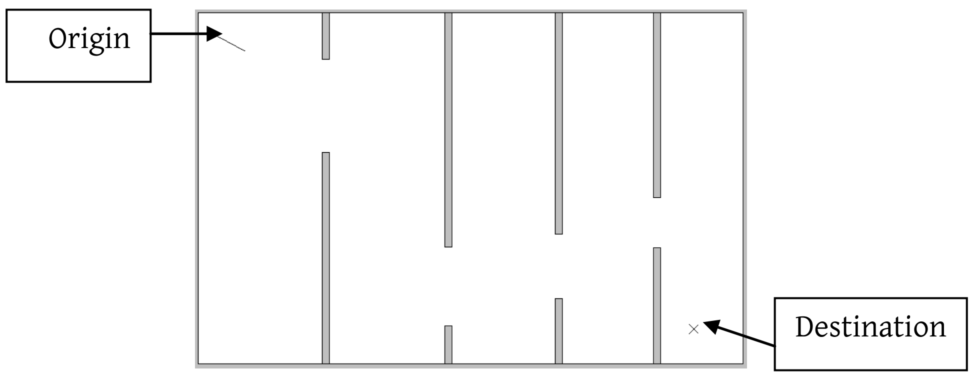


Figure 7.17 Moving Gaps Obstacle Course

It was decided to make the gaps bigger at the start and then let them become gradually smaller as the AGV got closer to the destination. The passages would also become narrower. This was introduced to allow the AGV to attempt overcoming bigger challenges the more it advanced within the experiment.

The final option is a random choice between all the previous obstacle courses. It is not an individual obstacle course, as with the previous ones. It is just an extra option.

7.2 Programme Generator

The Simulation-of-the-Functions-of-the-Microcontroller operates from a programme, as a normal processor would do. This programme has to be generated by some means. A human programmer could write the programme. Although a human programmed Simulated Programme was also incorporated here to prove that the simulator works, Simulated Programmes were mostly generated by algorithms.

Three methods of generating Simulated Programmes are introduced. These methods have overlapping qualities, and are an altered simulation of evolution in nature. The quality that all these methods use is mutation and they all need a fitness level. Different options are given to the user in order to make it possible to evaluate the processes against each other. These options are:

- Genetic Algorithms, with crossover, for generating Simulated Programmes
- Genetic Algorithms without crossover, where mutation is the only way whereby a new Simulated Programme is generated.
- Implementing Single-Chromosome-Evolution-Algorithms (SCEA).

After the completion of the evaluation of every single Simulated Programme, the Programme Generator's procedure is executed. Although Genetic Algorithms require the whole population to first be simulated before generating a new generation, the Virtual Programme was written in such a way as to accommodate the Single-Chromosome-Evolution-Algorithms.

Because Single-Chromosome-Evolution-Algorithms are new, they will mostly be evaluated and compared to the other options.

7.2.1 Determining the Current Fitness Level of the Simulated Programme

The current X position of the AGV is subtracted from the X position of the destination. The same subtraction is implemented for the Y coordinate. With these two resulting pieces of information, the linear distance between the AGV and the destination is determined. This linear distance is then subtracted from the initial distance between the AGV and the destination to get the fitness level of the AGV. In short, the closer the AGV gets to the destination, the higher the fitness level will be. This method is used in this programme because of its simplicity.

Another alternative to the problem is as follows: In section 2.7.5.3, it was shown that the AGV had to go to a position that was farther away from the destination than where it was presently at, in order to get to the destination. Therefore, if distance is used as an indication of fitness, then the AGV is more than likely not going to get to the destination, if the obstacle course used is similar to the example in section 2.7.5.3.

Imagine, for example, that the robot or AGV does not know where the destination is. The more the robot moves around, the more likely it will be to get to its destination. So the longer the distance traveled, the higher the fitness level must be. If it does succeed in getting to the destination, it must have done so in the least amount of steps. Consequently, if the destination is found, the distance travelled must be as small as possible, as this would save time and energy. It is now difficult to determine what the fitness level of a chromosome must be. The question is what the formula must be: higher fitness level for more steps taken or a lower fitness level for more steps taken?

There must be some change when the AGV has reached its destination. If the robot has already reached the destination, then it can change a variable into a different form, for example, from false to true. This can then force the programme to use a different formula to determine the fitness level. The problem, however, is to compare the results of these two formulas. To be able to compare the formula before the destination is reached with the formula for when the destination is reached, the formulas must have the same properties. This might be accomplished by extending the one formula to form the next formula, but how can one extend a formula in this way? One way is to say that the one formula is just the amount of steps taken. The next formula is the amount of steps taken minus two times the steps taken. The second formula is, therefore, just minus the steps taken. So, the one formula's results are going to be positive and the other's is going to be negative. However, one can see that this method will not work.

A method can be used where, in the case of the second formula, the amount of steps taken is subtracted from a fixed amount if the destination was reached. The problem, however, is what the value must be. The bigger the maze the higher the value must be because the AGV would have to take more "steps" in order to get to its destination. The second problem is that a robot that has not yet reached the destination, but has moved in circles for a hundred times or so, might have a higher fitness level than the one that has actually reached the destination in only a few steps.

The value that was referred to earlier can, therefore, become rather large. This means that the AGV that has found the destination has a much higher fitness level than the rest. A sudden jump in fitness will occur. This can be useful if the method of choosing a parent is where part of the population all had the same chance of been chosen. If the roulette-wheel-parent-selection method is used, then the ones that have reached the goal will have a much higher fitness level than the rest. This fitness level might be so high that the rest will, more than likely, not be chosen as a parent. This can make the problem of inbreeding prevalent, because only a small part of the population is used.

The maximum amount of steps taken by any AGV can be seen as the lowest fitness level a chromosome, that has reached the destination, can have. The amount of steps taken by the AGV that has reached its goal cannot be subtracted from this value because that could make this chromosome less fit than chromosomes that have not reached the destination.

The chromosome that has reached the destination must have a minimum fitness level of at least one more that the chromosome that has taken the most steps overall. The chromosome with the highest fitness level must be the one with the lowest amount of steps taken to reach the destination. The difference in the amount of steps taken by the one with the most steps and the one with the least amount of steps, when both have reached the destination, has to be added to the most steps taken by any AGV.

The formula is described in formula (7.7).

A =	Resulting fitness level of a chromosome that has reached the destination
B =	The most steps taken by any chromosome
C =	The most steps taken by a chromosome that has reached the destination
D =	The least amount of steps taken by a chromosome that has reached the destination
E =	The amount of steps taken by the individual that has reached the destination

$$A = B + (C - D) + 1 - E \tag{7.7}$$

Although it is difficult to give a specific fitness level value to a chromosome, it is easy to compare two chromosomes. The chromosome that has reached the destination as a higher fitness level than the chromosome that has not reached the destination. The amount of steps taken has already been explained.

With this information, one can determine which chromosome is the fittest, and which one is the second fittest. It is similar to a race, giving a person a position instead of a time. For instance, if there are ten chromosomes, then the fitness level of the chromosomes must be from one through to ten. Of course a special procedure has to be written to determine the fitness level (places, if it were a race). This is something that would not be needed if a mathematical formula was used.

7.2.2 Genetic Algorithms for Generating Simulated Programmes

Unlike the standard Roulette-Wheel-Parent-Selection, the parents could survive to the subsequent generation [32, p.8-11] [56, p.10-13]. In the same way as choosing a "More-Fit" chromosome as a parent, by changing the Roulette-Wheel-Parent-Selection in a small way, a "Less-Fit" chromosome (or Simulated Programme in this case) is chosen and then replaced with a newly generated chromosome, as will be seen in the following paragraphs.

Because the procedure for implementing Genetic Algorithms with crossover and the procedure for implementing Genetic Algorithms without crossover are so similar, they are imbedded into the same procedure in the Virtual Programme. The procedure, when implementing Genetic Algorithms, is broken up into two consecutive parts:

- Simulate the Whole Population. Because the Programme Generator's procedure is accessed after each and every evaluation, this part of the Programme Generator has to make sure the whole population of twenty Simulated Programmes is simulated to determine their fitness levels before a single new offspring is generated.
- Acquire Ten Siblings. The population is twenty, thus the "better" half of the old generation remains while the other half is replaced by siblings of the previous population.

7.2.2.1 Evaluate the Whole Population

The sub procedure shown in **figure 7.18** is the framework for the procedure that generates a Simulated Programme through both Genetic Algorithm methods used in this research. Imbedded within this procedure is the "Acquire Ten Siblings" procedure, which generates the new Simulated Programmes.

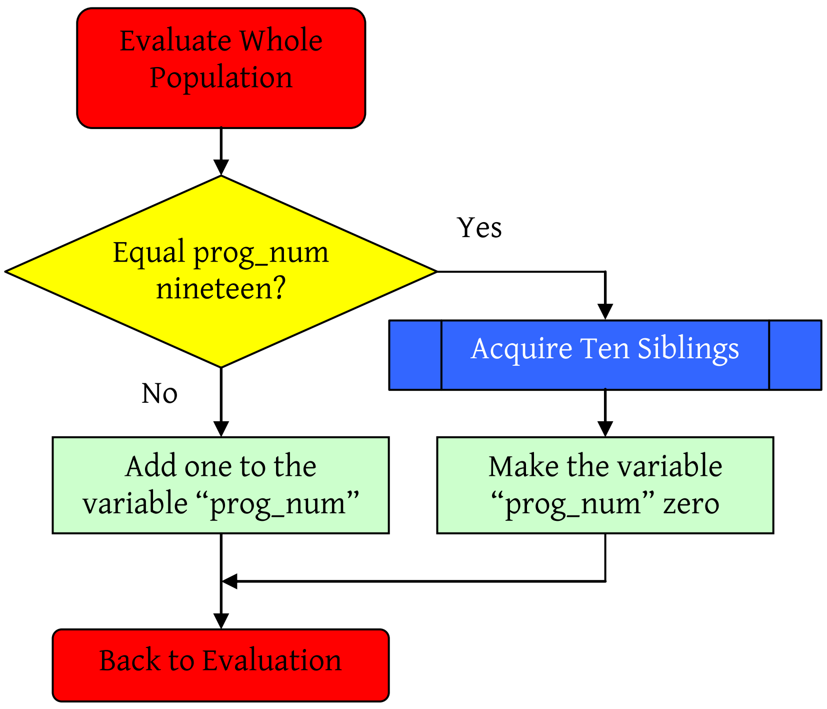

Evaluate Whole Population
Equal prog_num nineteen?
Yes
Acquire Ten Siblings
No
Add one to the variable "prog_num"
Make the variable "prog_num" zero
Back to Evaluation

Figure 7.18 Flowchart of Procedure "Evaluate Whole Population"

As with the rest of the Virtual Programme, the variable "prog-num" in this procedure, which is an abbreviation for programme-number, indicates which Simulated Programme is evaluated. This part makes sure all the Simulated Programmes are evaluated before generating a new generation.

7.2.2.2 Generation of Offspring

After each evaluation, the Simulated-Functions-of-the-Microcontroller is reset. This has to be incorporated; otherwise, the simulations are not identical to the previous simulations. This includes the array with the name DATA, which simulates the Data Registers of the microcontroller. Therefore, these individual variables of the DATA array can be temporarily accessed and used for other purposes, before it is reset. To save on the total amount of variables that are used by this programme, they are used by the functions in the following section.

The whole process of getting offspring is repeated ten times in order to get ten siblings. It is a basic "for" loop used in C++. It was decided that the count of the amount of siblings that have been generated is kept in the second variable in the DATA array, thus in variable "DATA[2]". The process is shown in the **figure 7.19**.

The highest fitness level obtained out of all the individuals in the previous evaluation of the whole population is needed in the sub-procedures "Choosing a 'Less Fit' Simulated Programme" and "Choosing a 'More Fit' Simulated Programme".

It precedes the whole of the generation of new siblings' procedures. The actual part of the programme that determines the highest fitness level is shown in **Programme Segment 7.1**. This procedure is illustrated as a flowchart in **figure 7.20**.

```
For ( prog_num = 0; prog_num < 20; prog_num++ )
 {
  if ( fitness [ prog_num ] > DATA[1] ) DATA[1] = fitness [ prog_num ] ;
 }
```

Programme Segment 7.1 Determine Highest Fitness Level.

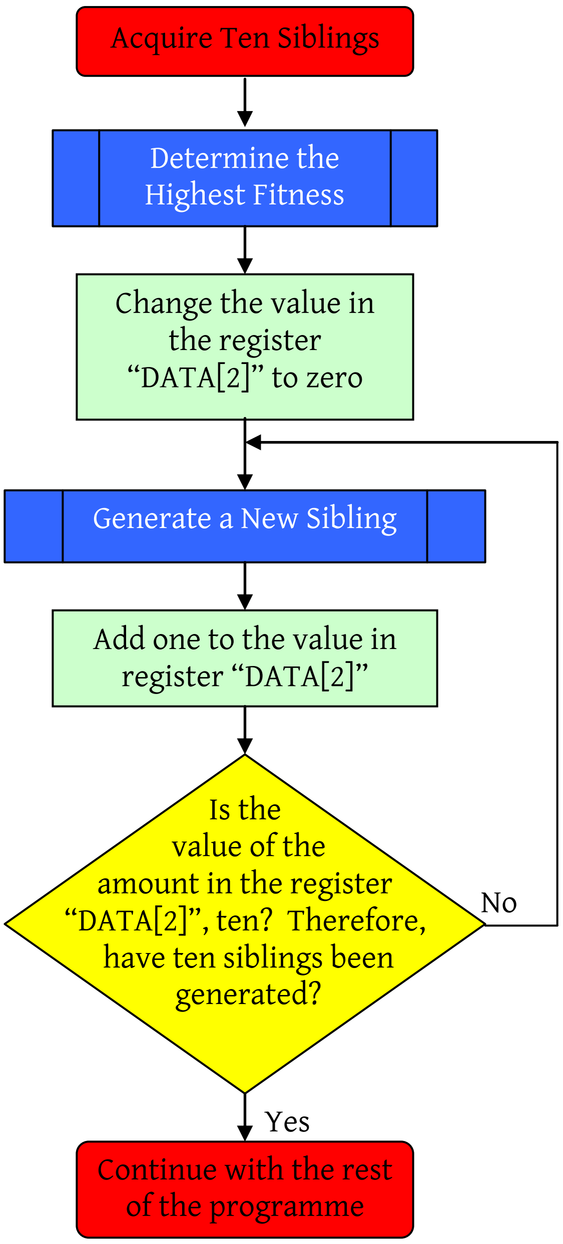
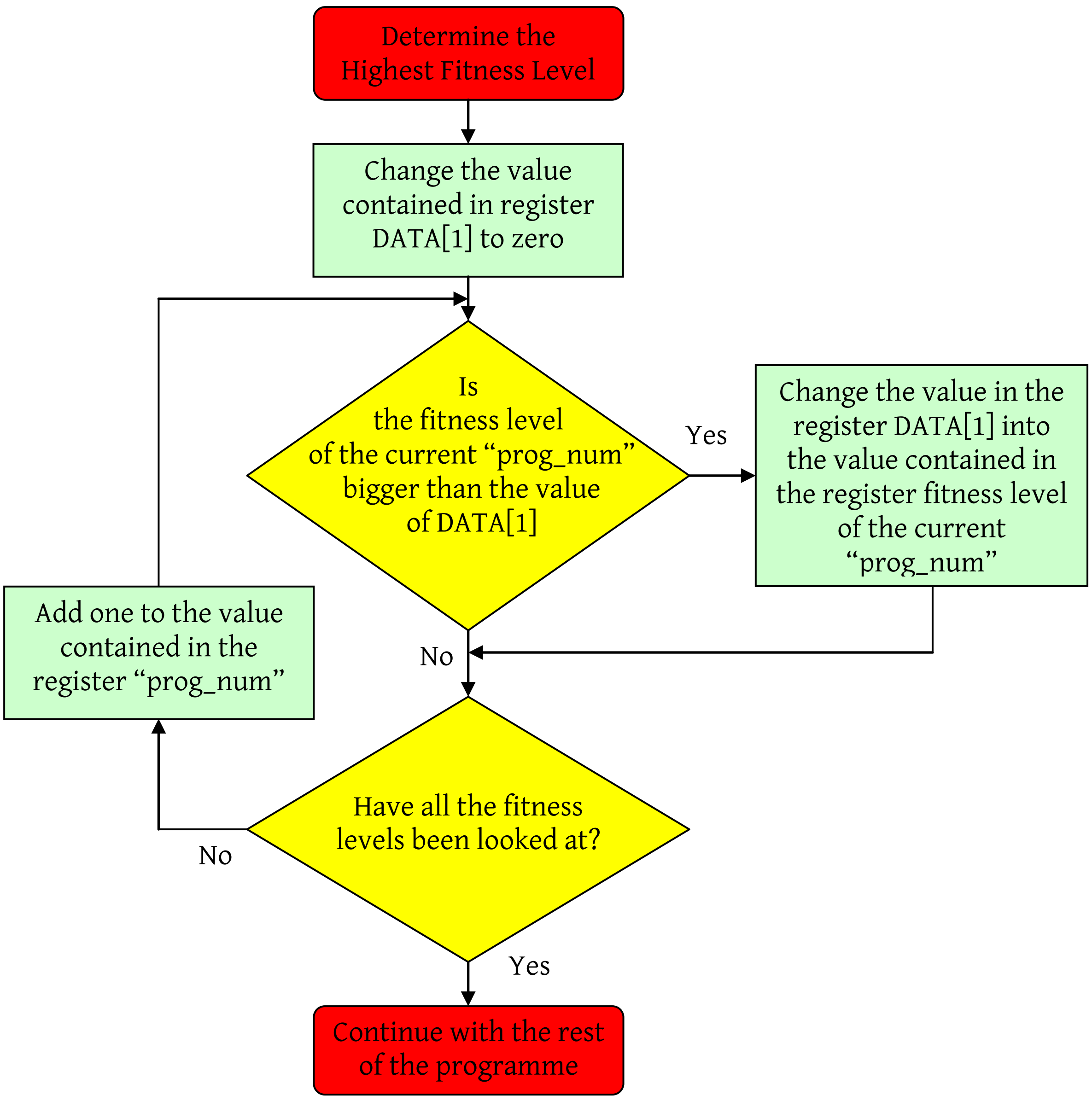

Acquire Ten Siblings
Determine the Highest Fitness
Change the value in the register "DATA[2]" to zero
Generate a New Sibling
Add one to the value in register "DATA[2]"
Is the value of the amount in the register "DATA[2]", ten? Therefore, have ten siblings been generated?
No
Yes
Continue with the rest of the programme

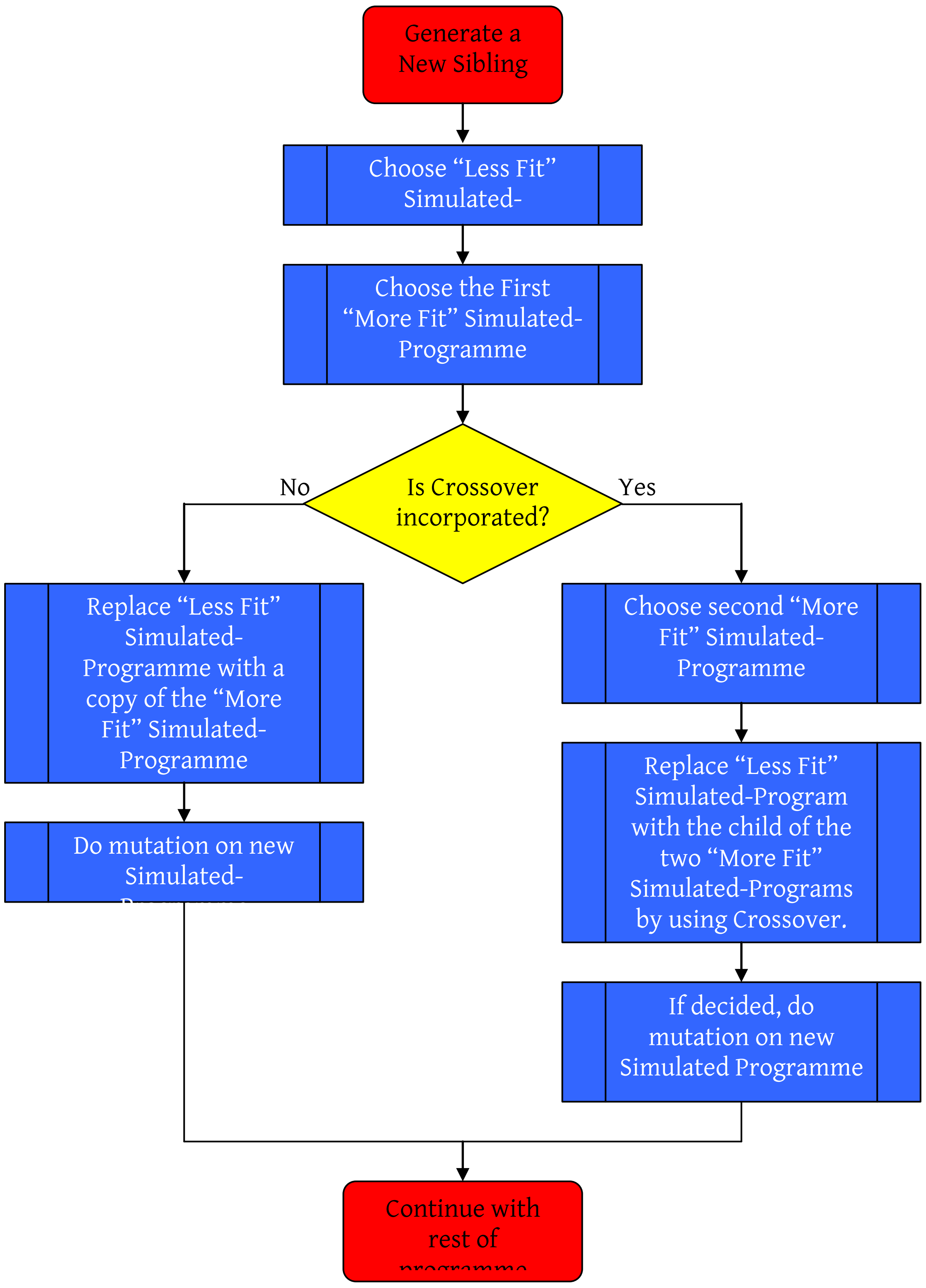
Figure 7.19 Flowchart of Procedure "Acquiring Ten Siblings"

Two variables are used in the programme to determine the highest fitness level. The one variable is the "prog_num" variable that is temporarily used because it is going to be changed into something specific afterwards. In the end, the highest fitness level will be contained in the variable with the name and number "DATA[1]".



Determine the Highest Fitness Level
Change the value contained in register DATA[1] to zero
Is the fitness level of the current "prog_num" bigger than the value of DATA[1]
Yes
Change the value in the register DATA[1] into the value contained in the register fitness level of the current "prog_num"
No
Add one to the value contained in the register "prog_num"
Have all the fitness levels been looked at?
No
Yes
Continue with the rest of the programme

Figure 7.20 Procedure to Determine the Highest Fitness Value



Generate a New Sibling
Choose "Less Fit" Simulated-
Choose the First "More Fit" Simulated-Programme
Is Crossover incorporated?
No
Yes
Replace "Less Fit" Simulated-Programme with a copy of the "More Fit" Simulated-Programme
Do mutation on new Simulated-
Choose second "More Fit" Simulated-Programme
Replace "Less Fit" Simulated-Program with the child of the two "More Fit" Simulated-Programs by using Crossover.
If decided, do mutation on new Simulated Programme
Continue with rest of

Figure 7.21 Flowchart Showing How a New Sibling is Generated

As was explained in section 2.7.5.2 Genetic Algorithms can be implemented with, or without, crossover [30, pp.513-516]. Here both options are implemented and are placed in one procedure. The flowchart of the procedure is shown in **figure 7.21**. First, the "weaker" Simulated Programme, that has to be replaced, and one "stronger" Simulated Programme that is going to replace the "weaker one", is determined. For Genetic Algorithms, that includes crossover, a second randomly chosen Simulated Programme is chosen to replace the "weaker" Simulated programme.

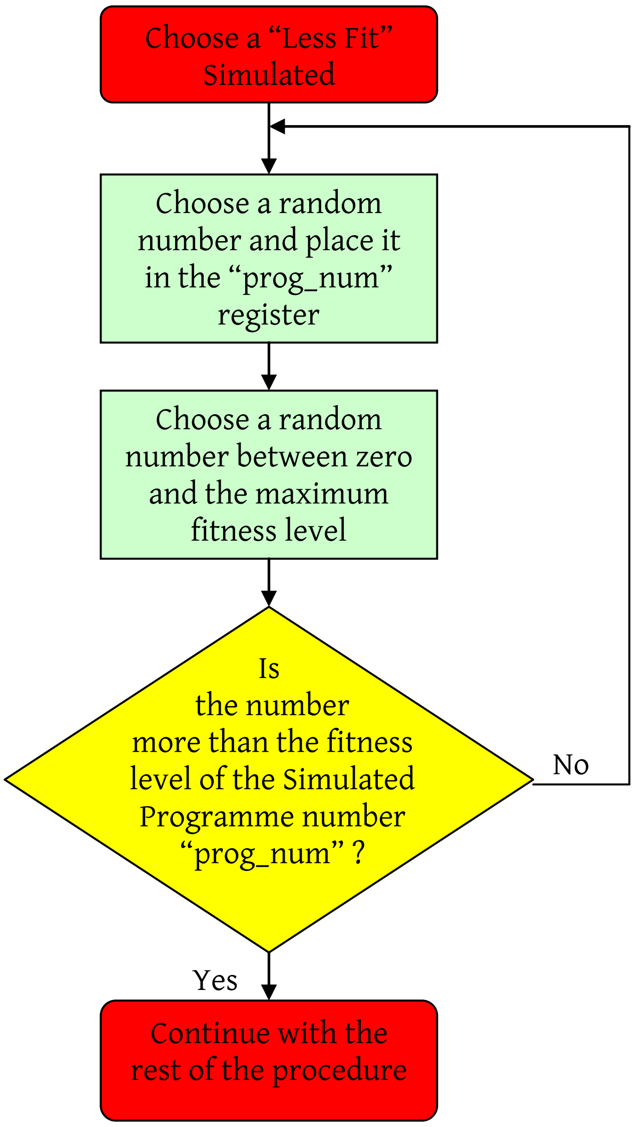

Choose a "Less Fit" Simulated
Choose a random number and place it in the "prog_num" register
Choose a random number between zero and the maximum fitness level
Is the number more than the fitness level of the Simulated Programme number "prog_num" ?
No
Yes
Continue with the rest of the procedure

Figure 7.22 Choosing "Less Fit" Simulated Programme

The Simulated Programme that is going to be replaced by a sibling is known as the "Less Fit" Simulated Programme. The worse a Simulated Programme's fitness level is, the better the chances for that Simulated Programme to be chosen as a "Less Fit" Simulated Programme. This part of the procedure is shown as a flowchart in **figure 7.22.** After completion of this procedure, a number identifying a Simulated Programme that has to be replaced is chosen, and that value is placed in the register "prog_num".

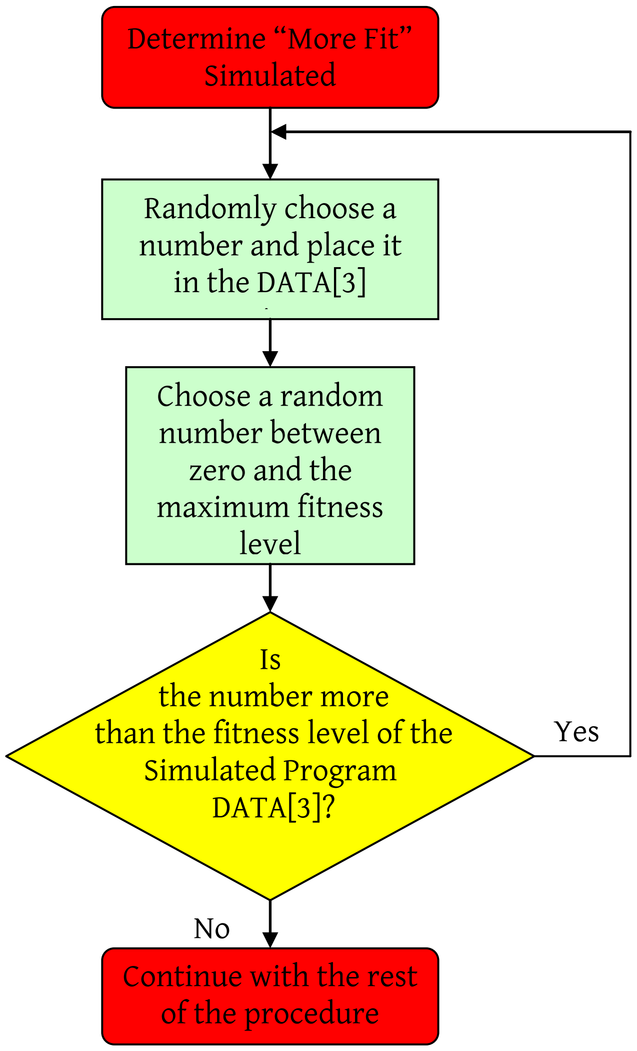

Determine "More Fit" Simulated
Randomly choose a number and place it in the DATA[3]
Choose a random number between zero and the maximum fitness level
Is the number more than the fitness level of the Simulated Program DATA[3]?
Yes
No
Continue with the rest of the procedure

Figure 7.23 Choosing "More Fit" Simulated Programme

Although most of the "Less Fit" Simulated Programmes have lower fitness levels than the average, any Simulated Programme can be chosen. This process is much like the Roulette-Wheel-Parent-Selection method. The only difference is that in this case the "weaker" programme is chosen instead of the "stronger" one. As already stated, the value of the highest fitness level is contained in the variable "DATA[1]".

There is a slight difference between choosing a "More Fit" Simulated Programme and choosing a "Less-Fit" Simulated Programme", as can be seen when comparing **figure 7.22** and **figure 7.23**. The only differences are that the result is kept in the register "DATA[3]" and the "Yes-No" options of the test are turned around.

7.2.3 Implementing Single-Chromosome-Evolution-Algorithms

This is a separate procedure to that of the two Genetic Algorithm methods. As illustrated in **figure 7.24,** the programme is much smaller and less complex than Genetic Algorithms.

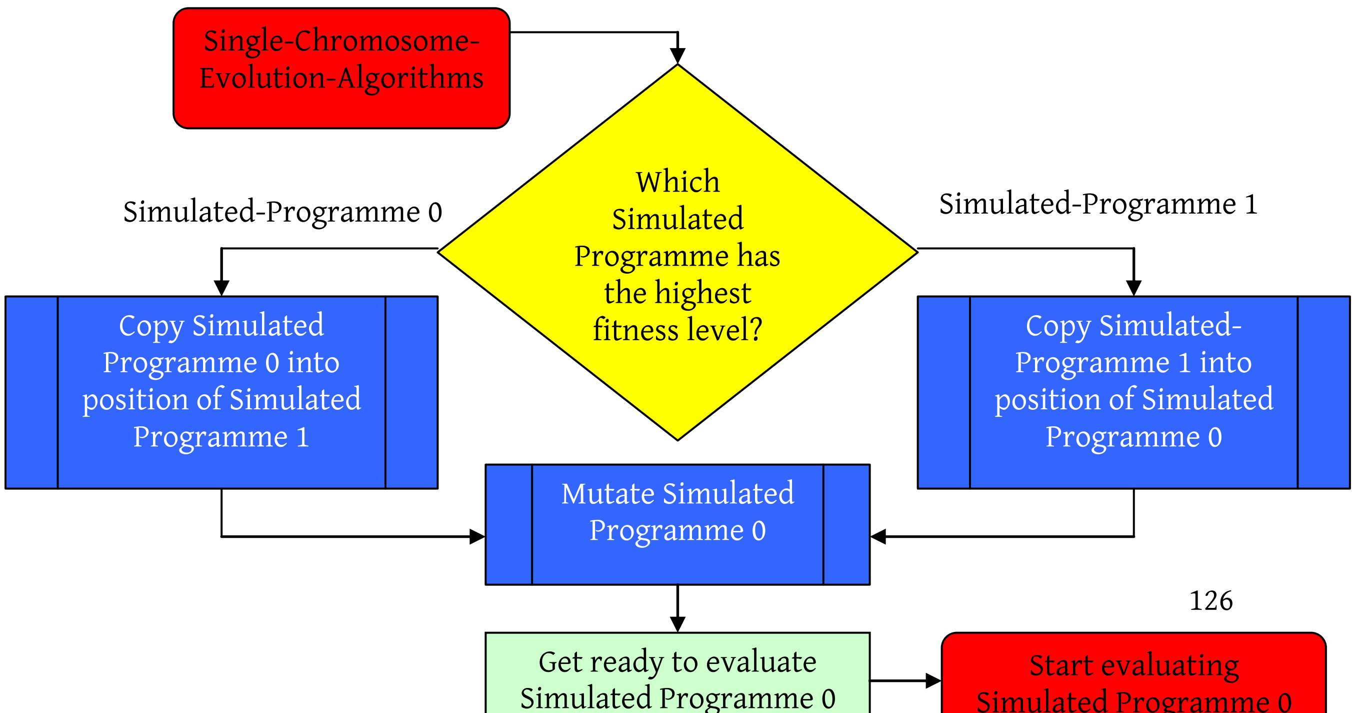

Single-Chromosome-Evolution-Algorithms
Which Simulated Programme has the highest fitness level?
Simulated-Programme 0
Simulated-Programme 1
Copy Simulated Programme 0 into position of Simulated Programme 1
Copy Simulated-Programme 1 into position of Simulated Programme 0
Mutate Simulated Programme 0
Get ready to evaluate Simulated Programme 0
Start evaluating Simulated Programme 0

Figure 7.24 Flowchart of Single-Chromosome-Evolution-Algorithms

When implementing Single-Chromosome-Evolution-Algorithms, all that is needed is a programme similar to the one shown in **figure 7.24** and the Evaluator.

7.2.4 Copying One Simulated Programme into Another

The method of copying one Simulated Programme into another is used by Single-Chromosome-Evolution-Algorithms as well as Genetic-Algorithms that only make use of mutation. The old version of the Simulated Programme is removed and replaced by a new one. At the completion of this function, there will be two identical Simulated Programmes.

As will be explained later, the crossover in Genetic Algorithms is similar to this. As an illustration, the flow diagram of the copy method of the Mutation-Only-Genetic-Algorithms in **figure 7.25** will be used. The way in which it is used in the programme is shown in **programme segment 7.2**.

```
for(DATA[4]=0;DATA[4]<50;DATA[4]++)
{
 programme[prog_num][DATA[4]] = programme[DATA[3]][DATA[4]];
}
```

Programme Segment 7.2 Copying Simulated Programme.

In the Mutation-Only-Genetic-Algorithm, a "Less Fit" Simulated Programme has to be replaced by a "More Fit" Simulated Programme. An instruction, as well as its operands, occupy a single register in memory and it possesses a single address [18, p.19&147].

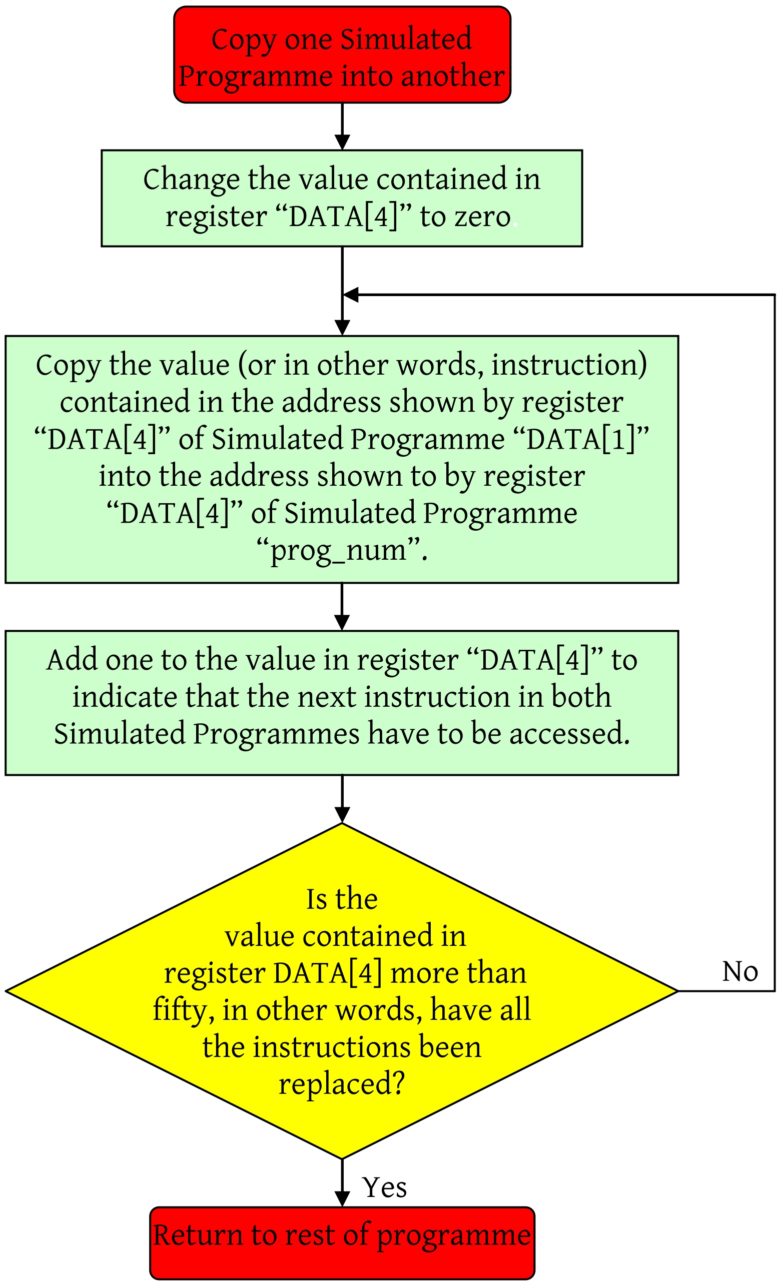

Copy one Simulated Programme into another
Change the value contained in register "DATA[4]" to zero
Copy the value (or in other words, instruction) contained in the address shown by register "DATA[4]" of Simulated Programme "DATA[1]" into the address shown to by register "DATA[4]" of Simulated Programme "prog_num".
Add one to the value in register "DATA[4]" to indicate that the next instruction in both Simulated Programmes have to be accessed.
Is the value contained in register DATA[4] more than fifty, in other words, have all the instructions been replaced?
No
Yes
Return to rest of programme

Figure 7.25 Procedure to Copy a Simulated Programme

The information in all of the registers of the "More Fit" Simulated Programme has to be copied, one by one, to the corresponding address' registers of the "Less Fit" Simulated Programme. A software loop, which goes through all the addresses in the Simulated Programmes, is needed. A variable is needed to keep hold of the instruction address that has to be copied at that instance. It was decided that this value would be kept in the "DATA[4]" register. Because there are fifty instructions in each Simulated Programme, this value has to go though all fifty addresses.

7.2.5 Crossover for Simulated Programmes

Crossover is the same as two "Copying One Simulated Programme into Another" procedures. Firstly, a crossover point is randomly determined, as shown in **figure 7.26. Programme segment 7.3** shows the part of the actual programme that is performing the crossover.

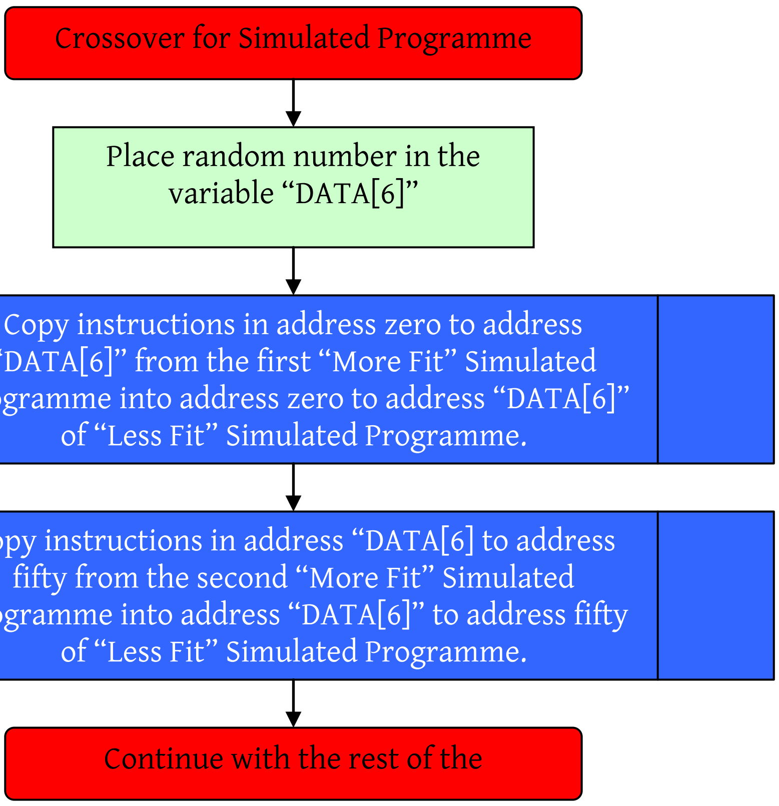

Crossover for Simulated Programme
Place random number in the variable "DATA[6]"
Copy instructions in address zero to address "DATA[6]" from the first "More Fit" Simulated Programme into address zero to address "DATA[6]" of "Less Fit" Simulated Programme.
Copy instructions in address "DATA[6] to address fifty from the second "More Fit" Simulated Programme into address "DATA[6]" to address fifty of "Less Fit" Simulated Programme.
Continue with the rest of the

Figure 7.26 Implementing Crossover on Simulated Programme

Because there are fifty instructions in a Simulated Programme, the crossover point is a number between zero and forty-nine. As shown on **figure 7.21** earlier in this chapter, before crossover is performed, two "More Fit" Simulated Programmes already had to be determined.

```
DATA [6] = random (49);
 For ( DATA [4] = 0 ; DATA [4] < DATA [6] ; DATA [4] ++)
 {
  programme [prog_num] [DATA[4]] = programme [DATA [3] ] [DATA [4] ] ;
 }
 for ( DATA [4] = DATA [6] ; DATA [4] < 50 ; DATA [4] ++)
 {
 programme [prog_num] [DATA [4] ] = programme [DATA [5] ] [DATA [4] ];
 }
```

Programme Segment 7.3 Crossover for Simulated Programme.

From the address of the first instruction, up to the crossover point, each instruction of the first "More Fit" Simulated Programme, with its operands, is copied, one by one, into the "Less Fit" Simulated Programme memory position with the same address. Then, from the crossover point, up to address forty-nine, the instructions, and their operands, of the second "More Fit" Simulated Programme are copied one by one into the memory position with the same address of the "Less

Fit" Simulated Programme. The result is a Simulated Programme where the first few instructions are exactly the same as the first few instructions of the first "More Fit" Simulated Programme, and the rest of the instructions are exactly the same as the second "More Fit" Simulated Programmes instructions.

7.3 Starting and Stopping the Evaluation

As was shown at the introduction to this chapter in **figure 7.2**, there is an alteration between the Programme Generator and the Evaluator. It is easy to see when the next Evaluation must start; it is directly after the creation of the new generation is complete. The problem is determining when the evaluation is finished. The following four reasons for stopping the evaluation will to be discussed:

- Detecting an error in the Simulated Programme
- The Simulated AGV has collided with something
- A timeout occurred
- The user has stopped the programme

The timeout termination was brought in because the Simulated-Functions-of-the-Microcontroller might not run errors, the AGV is not moving or is running around in circles, and as a result, does not collide with any walls. If it were not for the timeout termination, it would have been possible for the evaluation to continue without ever accomplishing anything.

7.3.1 Detecting an Error in the Simulated Programme

The following errors lead to termination of the current evaluation:

- The Simulated-Functions-of-the-Microcontroller wants to access an *instruction* at an address higher than fifty. As will be stated in section 11.1, the memory size of one Simulated Programme is only fifty instructions in length.
- The Simulated-Functions-of-the-Microcontroller wants to access *data* in a register with an address higher than fifty. Although there is only one set of data memory, there are also just fifty registers.
- The Stack is overrun. It was decided that, in order to compare the Simulated-Functions-of-the-Microcontroller with a physical PIC microcontroller, only ten stacks would be used.

7.3.2 AGV Collides with an Obstacle

The detection of a collision is made part of the Distance Sensor procedures. If the distance to the nearest obstacle is closer than a fixed amount, it is presumed that a collision has occurred. The two AGV in **figure 7.27** are about the same distance from the obstacle, although the AGV on the left has registered a collision while the one on the right has not. Thus, it is possible that an obstacle can be closer to the AGV than what is allowed and not be sensed as a collision.

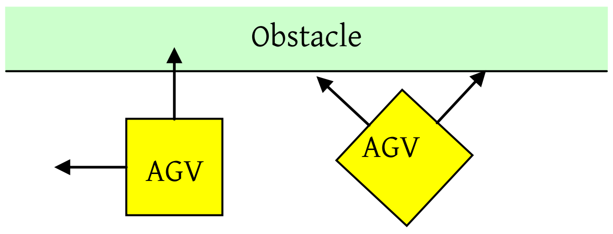


Figure 7.27 Rotation Influences Collision Detection

Two scenarios happen before a collision is registered. If the AGV systematically gets closer, so as to reduce the distance sensed by the Distance Sensors, it can be sensed as a collision. The bump will also be sensed as soon as the Simulated AGV turns, so as to let one of the sensors sense the obstacle more directly, as shown in **figure 7.27**.

7.3.3 A Timeout Occurrence

The timeout is accomplished by inserting a global variable into the programme. This variable counts the amount of instructions executed and, if the amount of instructions executed is more than a specified amount, a timeout occurs. Through examination, this amount was changed until the seemingly best results were met.

This maximum amount of instructions can influence the speed of the simulation and, if it is made too short, can influence the final fitness level of each Simulated Programme. This happens because, if this maximum amount is made too long, the AGV will take up a lot of time doing nothing, and if it is too short then the AGV will not be able to do enough steps to get to the destination.

Having too short a timeout will have a hidden advantage in that the AGV will look for the shortest way to the destination. This happens due to the fact that, if it were to choose the path that was not the shortest, then the timeout would occur when the AGV was further away than it would have been if it had taken a shorter route.

With all the options in mazes, it is difficult to determine what the maximum amount of instructions must be if they had to be just enough to get to the destination.

The following method can be incorporated to get short timeouts, if they are required, while giving timeouts that last long enough to get the AGV to the destination when it is able to reach it. The mazes can be divided into blocks. The AGV must not exceed the time allocated for it to enter a block that is closer to the destination, otherwise a timeout will occur. If the block entered is further away from the destination than the previous block, then the timeout must occur more quickly because the AGV has taken the wrong turn.

If these blocks were made infinitely small, then a continuous formula can be incorporated. The fitness level already indicates the difference in linear distance between the AGV and the destination. In this dissertation, a formula is used to determine the maximum amount of instructions that a Simulated Programme is allowed to execute for a given situation. If the AGV has not moved for some time, then a timeout will occur. The more steps the AGV takes in the direction of the destination, the more time must be allowed for the AGV to get to the destination.

It seems that the biggest problem when the AGV start moving is that it runs into walls instead of simply doing nothing. The AGV will, most of the time, not be able to walk to the destination in a straight line, as is one of the characteristics of the fitness calculation in this dissertation. Extra time for avoiding obstacles is added to the maximum amount of allowable instructions that may be executed. The formula for the maximum allowable instructions is:

Timeout = A + B * Fitness.

Fitness : Current fitness level of Simulated Programme

A : Maximum amount of instructions allowed to be executed when the AGV is not moving, or extra time to avoid obstacles

B : The distance the AGV travels while one instruction is executed plus extra time to avoid obstacles

7.3.4 User Stopping Programme

There are two ways for the user to stop the programme. Everything has to come to an end. By pressing the "ESC" button the Virtual Programme is terminated. One can also pause the programme in order to see a graph of the results. Because the evaluation is the most time-consuming part of the programme, it will seem as though the evaluation is always being interrupted. However, theoretically, it is possible to stop the programme during the Programme Generation stage. Unlike the other options, generating a new generation of Simulated Programmes will not necessarily be executed after the programme has been stopped in this way. The programme will either be terminated or will continue from where it was originally stopped.

7.4 Summary

A Simulated Programme is evaluated by executing it using a simulation of the AGV in a simulated environment. The result from the evaluation is a fitness level. Each Simulated Programme is evaluated individually and each is assigned its own fitness level. These fitness levels and their respective Simulated Programmes are used in the generation of new, hopefully better, Simulated Programmes.

The "sensors" and the "motors" of the Simulated AGV have to be calibrated with the physical infrared sensors, and modified servomotors on the Physical AGV. The surroundings are in the form of different mazes and only the sensors react to the obstacles in these mazes. The controller of the Simulated AGV is the Simulated-Functions-of-the-Microcontroller, and is explained in chapter 11.

The generation of a new Simulated Programme is accomplished by using Mutation Only Genetic Algorithms, Genetic Algorithms that use mutation and crossover, or through Single-Chromosome-Evolution-Algorithms. The user has to choose between one of these three options.

The evaluation of an individual is complete when the AGV has collided with a wall, a timeout has occurred or an error has been detected in the programme. The user can temporarily stop the programme, or the user can terminate the programme.

Crossover is accomplished by connecting the first part of a "More Fit" Simulated Programme to the last part of another "More Fit" Simulated Programme. Mutation of the Simulated Programmes is accomplished by using the methods explained in chapter 8. The resulting Simulated Programme replaces a "Less fit" one. When 10 of the 20 Simulated Programmes have been replaced, the evaluation of all the Simulated Programmes starts again.

8 Special Mutation

In this dissertation, mutation is a way of altering instructions in a Simulated Programme, in order to turn it into a new Simulated Programme. All three-programme generation methods use mutation. As will be illustrated in this segment, Special Mutation is different to normal mutation in that it speeds up the generation of Simulated Programmes with a high level of fitness.

An instruction is a combination of ones and zeros. This combination makes up an equivalent decimal number. In the PIC microcontroller, being either real or simulated, this single number contains the opcode as well as the operand [18, p.147]. It is possible that, if a change is made to a single bit, for example a zero has to be turned into a one, the result will be that the operand can be changed by a certain amount. On the other hand, if another bit is changed in the same combination of ones and zeros, the opcode can be altered, thus, changing the function of the instruction. This change to the instruction can dramatically alter the course of the whole Simulated Programme.

Some instruction opcodes, like the instruction “CALL”, are just three bits in length, thus, changing one bit could change this instruction into only one of three other instructions. In total, it is better to change the whole instruction so that it is possible to change the instruction into any other instruction.

8.1 Disadvantages of Normal Mutation when Generating Programmes

Randomly choosing an instruction and its operand makes it possible for the mutation to add any instruction to the Simulated Programme. This seems to be the only advantage to randomly choosing an instruction and its operand. Adding this as an additional option while giving it a small chance of being chosen, gives the Special Mutation procedure this advantage, as will be seen later in this chapter. The disadvantages are investigated in the following section.

8.1.1 Unequal Chance of Being Chosen

Not all instructions have an equal chance of being chosen. All instructions of the PIC are fourteen bits in length. The OPCODE for the PIC's "CALL" instruction is made up out of four bits; the rest is the operand, which points out the address of the next instruction to be executed [18, pp.147, 151]. There are $2^{14\text{bits} - 4\text{bits}} = 2^{10\text{bits}} = 1024$ possible operands which could point to 1024 different addresses. Thus, the "CALL" OPCODE, plus the operand, make up 1024 total possible combinations of ones and zeros, all executing a "CALL" instruction. The OPCODE for the "RETURN" instruction takes up the whole fourteen bits, thus, there is only one combination of ones and zeros that makes a "RETURN" instruction. In order to not overrun the stack, every "CALL" instruction must have a RETURN instruction in a programme. If a random number were chosen to determine the total instruction, then the "CALL" instruction would have 1024 times more chance of being chosen than the "RETURN" instruction.

8.1.2 Simulation Time to Get Correct Instruction

One of the functions most frequently used by the test programmes is the function to make the AGV move forward. As can be seen from **table 7.1,** the number ten had to be loaded into the output register to make the buggy move forward. Two instructions had to be executed one after the other. The first instruction loaded the W-register with the number ten. The second instruction loaded the information that was in the W-register, which was ten at that stage, into the ninth data register, that functioned as the output to the motors. These instructions could not be altered. If one was lucky enough to get these instructions in the right sequence then there was a one out of $2^{\text{14bits + 14bits}}$ = 268 435 460 chance of these instructions being chosen. That means that, over an average of 268 435 460 simulations, the Simulated AGV will start moving, thus taking a lot of processing time.

8.2 *Focused Mutation*

To better the odds, instructions commonly used in the test-programmes were favoured. The sequence in which the instructions were executed was also a consideration. Different options as to how many instructions must be updated at one time were made available to the user to determine which mutation size makes learning possible, and which option was the most ideal.

In the experiments, it was seen that the bigger the amount of instructions inserted at one time, the faster the generation of the Simulated Programme. The question “why not just update the whole Simulated Programme with a single prewritten Simulated Programme?” arose. As can be seen, this takes away the self-generation properties of the programme. The ideal is, thus, to have a generator that changes only one instruction per simulation. The option is given to the user to insert one, two or four instructions per mutation.

The Simulated Programme is divided into Blocks. Each block acts as a gene in a chromosome, and the blocks are made up of either four instructions, two instructions or only one instruction.

There are three levels of decisions the Virtual Programme goes through to get one mutation. The first decision is the amount of instructions that must be changed for one mutation and is given as an option to the user. Although the user can change the amount of instructions any time, it is recommended that it must stay constant throughout the whole evolution process.

The second decision is which block or group of instructions to use. If the first decision was four instructions per mutation, then those four instructions would work together to do some specific task. Each time the mutation procedure is called up, thus almost each time a new Simulated Programme is generated, a different decision is made as to which group of instructions to use. Although this choice is made randomly, not all the different types of blocks have the same chance of being chosen.

The last decision is to generate the operands’ values. Most of these values are generated directly by the random function. For example, the operand can be one of 255 values, and the random function will only choose one of these. Other operands are more specific, like, for instance, when the operand for the “GOTO” command has to be determined. As an example, it is assumed that the user has chosen four instructions that have to be changed per mutation. As always, the total amount of instructions in a Simulated Programme is fifty. Therefore, the total amount of blocks can only be 50/4 = 12 blocks. A number between zero and eleven is randomly chosen to decide which block to go to. Four is multiplied with the chosen block's number to get the address for that specific block's first instruction.

8.2.1 Inside the Blocks

```
Address   OPCODE Operand

  1000    MOVLW distance_before_turning
  1001    SUBWF      sensor1,0
  1002    BTFSS      STATUS , Carrie
  1003    GOTO    Alternative address
```

Programme Segment 8.1 Decision-making Procedure in Simulated Programme.

When the user has chosen to change four instructions per mutation, then the second choice is between one of the following options.

- This block is precisely identical to the procedure in **programme segment 8.1**; four instructions are needed to make a complete test of an input. The information on which the tests are performed mostly comes from the simulated sensors. Depending on the result, the programme can either execute the next block of four instructions or it can go to any block of four instructions in the programme. It is decided that this programme block must have the biggest chance of being chosen, because information from more than one sensor might be needed for only one reaction.

- Only two instructions are needed to load information into the data registers, which can act as the output to the "motors" of the AGV. As already explained, the "GOTO" command in the previous section can only point to the beginning of a block made out of four instructions. Two more instructions are needed to fill the gap. The other two instructions that made up the block were the "NOP" instructions. The "NOP" instructions do nothing but waist time [18, p.157].
- All four instructions are determined randomly. So, any of the four addresses in the block can contain any instruction, and the operand can be any amount allowable. This kind of block had the least chance of being chosen.

Each block is a working unit and it does not matter if, for example, one test leads to another, so it is less sensitive to the sequence.

The user then chooses to update two instructions per mutation, and the second decision is one of the following:

- The information in the registers, mostly the information coming from the sensors, is compared to a value. A flag bit is changed after these instructions are executed [18, p.161]. The flag will indicate the relationship between the two values that have been compared. These instructions are similar to the first two instructions in **programme segment 8.1**. The chance that this option will be chosen is very high.
- Splitting the Simulated Programme is performed by testing the information in the flag register. If the results in the flags are some specific information, the programme will continue with the next block. If not, then the programme will execute any block specified. These instructions are similar to the last two instructions in **programme segment 8.1**. Unlike the operand

for the four instructions where it chooses one of the four-instruction blocks, this option chooses any of the two-instruction blocks, thus, the Programme Generator could make mistakes more easily because the sequence in not necessarily correct. It has very high chance of being chosen as an option.

- This option is exactly the same as option two, where four instructions are updated, except that the two "NOP" instructions are not needed. It has a lower chance of being chosen than the first two options.
- Both instructions and their operands are chosen randomly. This option has the smallest chance of being chosen.

When the user has chosen the option that one instruction must be changed for one call to the mutation procedure, the mutation procedure must choose one of the following instructions:

- The "MOVLW" instruction loads information into the W-Register. This information is random. This is similar to the first instruction in **programme segment 8.1** and stands a very good chance of being chosen.
- The "SUBWF" instruction acts as a function for comparing two values. It mostly acts on the registers that contain the information from the sensors. After this instruction is executed, the flags are altered to indicate the relationship between the information in the registers chosen, for example the relationship between the inputs from the sensors, and the value in the W-register. It has a very high chance of being chosen and is similar to the second instruction in **programme segment 8.1**.
- The "BTFSS" instruction is used to test the flags. Normally it is used to test any bit in any data register, but in this case the operands are inserted is such

a way as to concentrates on the flag bits. So, it is a big advantage to give this a higher chance of being chosen than simply using random instructions. If the bit pointed to by the operand is one then the next instruction is skipped, if not then the next instruction is executed [18, p.151]. This instruction stands a very good chance of being chosen and is similar to instruction three in **programme segment 8.1**.

- The "GOTO" instruction makes the programme jump to any address in the programme. It is hoped that this instruction will follow the "BTFSS" instruction. This will split the programme so that the part of it that needs to be executed is executed. It can also force the programme into a loop so that the programme does not finish.
- Although this instruction is also a "MOVLW" instruction, as with the first option, it differs in that the information that can be loaded into the W-register is information mostly needed by the "motors" of the AGV. It makes up the first half of option three, when two instructions are updated at one time. It has less of a chance of being chosen than the first four options.
- The information in the W-register is loaded into the register that acts as the output. It forms the second half of option three, where two instructions are updated at once.
- Any instruction and its operand are chosen randomly. This instruction has the smallest chance of being chosen.

In all three options, four, two or one instructions change per mutation, the option where the instructions and the operands are changed totally randomly is the biggest source of lower fitness levels, but it could give the Simulated Programme what might be needed and cannot be generated in any other way. It also gives the programme certain uniqueness.

8.3 Summary

Changing one bit in a Simulated Programme changes an instruction into only a small amount of other instructions, depending on the length of the original instructions OPCODE, thus, it is better to change the whole instruction [18, p.147].

The next scenario is to randomly change the whole instruction. The bigger the OPCODE of an instruction is, the less chance there will be that an instruction with that OPCODE will be chosen.

To obtain the correct instruction or combination of instructions could be a lengthy process, as much time could be wasted on instructions that will not work. It could take an average of more than two hundred million simulations for the AGV to just start moving.

These two problems are overcome by using focused mutation, where instructions that were used by a human programmer were given a higher chance of being chosen. For experimental purposes, the user is given the option to change one, two or four instructions per mutation. These instructions normally follow each other, like, for instance, “BTFSS” followed by “GOTO” or “MOVLW” followed by “MOVWF”.

9 Connection Programme

This programme's primary function is to make it possible to implement the Simulated Programme on the Physical AGV. This programme only forms half of the total programmes that are needed to execute a Simulated Programme for the Physical AGV. The other half is the programme on the Physical PIC. This whole combination of hardware and software has the same function as the Evaluator in the Virtual Programme. The Connection Programme written using DOS as the operating system was broken up into the following segments:

- The Communication Section replaces, or does the same work as the simulated part in the Virtual Programme. It generates half the protocol that communicates with the Physical AGV
- The Simulated-Functions-of-the-Microcontroller for the Physical AGV part is identical to the Simulated-Functions-of-the-Microcontroller used in the Virtual Programme.

Because it is not easy to tell the distance to a destination in the physical world, Simulated Programmes are not generated by the Communication Programme. They are already generated by the Virtual Programme in any case, so why would one want to do it again?

9.1 PC to AGV Communication

The Communication Section is a gateway to controlling the Physical AGV. In the end, a value inside a variable has to be copied from the computer or laptop to the

Physical PIC in order to control the motors or from the Physical PIC to the PC to get the readings coming from the sensors copied to a variable in the PC. The whole connection between the AGV and the computer is explained in the section "Protocol between Computer and AGV".

Instead of coming from a simulated sensor, information is coming from the Physical Sensors. The information is going through the communication parts of both the computer and the Physical PIC. Thus, in other words, the values come from the communication component of the computer, and the information generated is not going to a Simulated AGV but, instead, is going to the communication part. From there, it is transported to end up influencing the Physical Motors.

9.2 Summary

The function of the Connection Programme is to implement the newly-generated Simulated Programme in the Physical AGV. It is comparable to the Virtual Programme except that here, one makes use not of a simulation, but the Physical AGV. Evaluation also does not take place, as the distance between the destination and the AGV cannot be determined by the AGV.

The programme is broken up into two parts:

- PC to AGV communication section that controls the tempo of throughput of information, as well as sends and receives information from and to the PIC onboard the Physical AGV.

- Simulated-Functions-of-the-Microcontroller, as already stated, is an exact copy of the controller for the Simulated AGV and is discussed in detail in chapter 11.

All information that flows through the PC to AGV communication section is either generated by, or is available for use by the Simulated-Functions-of-the-Microcontroller.

10 Programme for the Physical PIC

Information from the sensors is gathered and sent to the computer via the umbilical cord. Information from the computer is, in return, sent to the motors via the Amplifiers-H-bridge configuration, as shown in **figure 5.3** in chapter 5. The Physical AGV is under total command of the PC. The AGV is made to react as if it has a PC onboard, while in actual fact there is not enough room on the AGV for a PC.

The programme is broken up into three parts:

- Sensor Interface
- Communication Interface
- Output to Motors

The three parts are intertwined and timing is of utmost importance in order to get the highest transmission via the umbilical cord.

10.1 Sensor Interface

As was seen in section 5.2.1, the intensity of the received light is used for determining the distance to the obstacle. This electrical analogue value is turned into a digital value by the PIC's onboard Analogue-to-Digital Converter.

Information is not read from all the sensors at the same time. To save on power, and because the transmitters are not made to handle such high currents, the transmitters are only turned on when the sensor applicable to that transmitter is read from. In the programme, that is accomplished by simply setting the correct bit in the data registers. The added advantage of only turning on the specific transmitter is that only light coming from that specific transmitter is read.

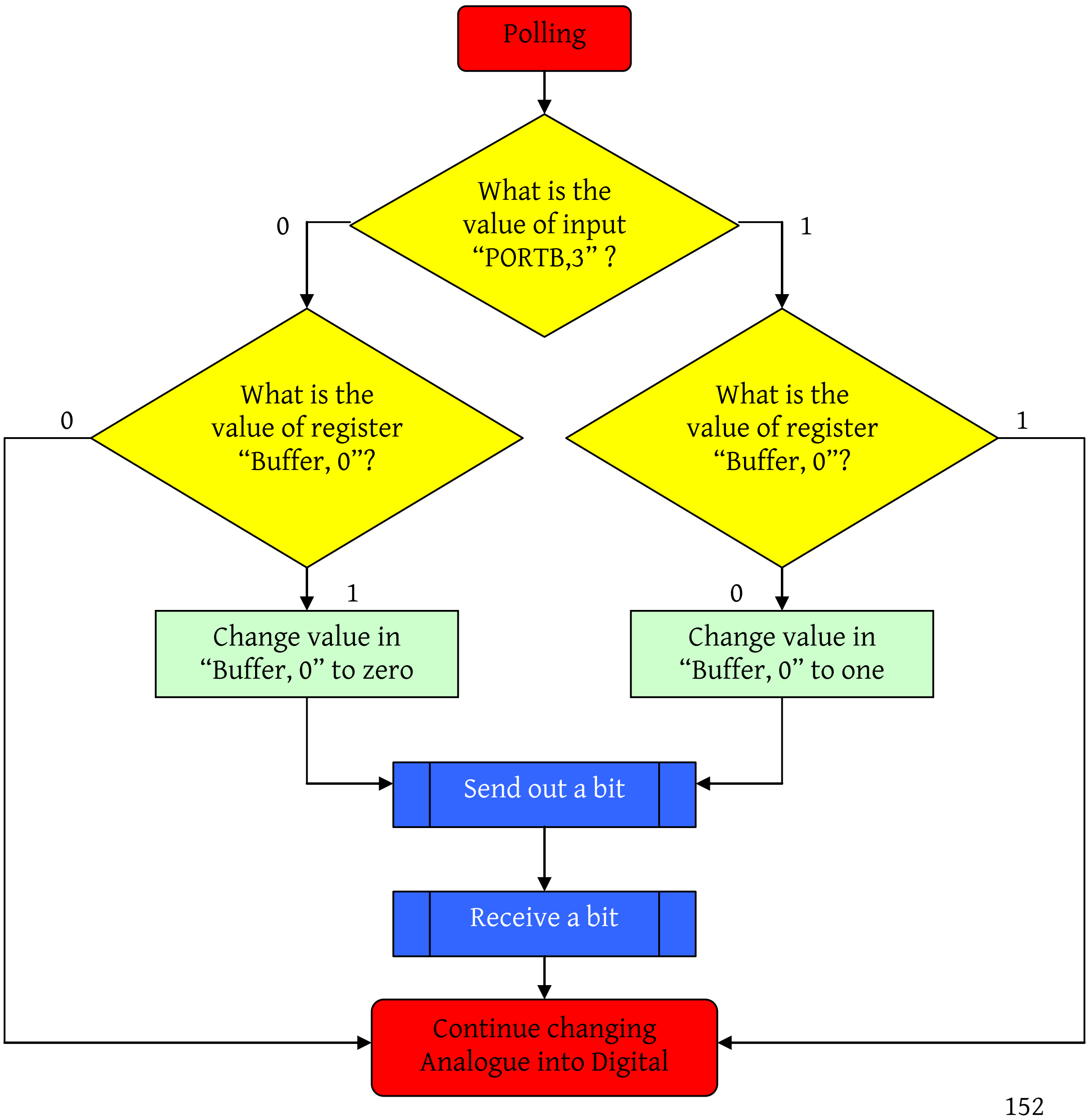
Each "Sensor" is dedicated to its own register in the data array. After completion of a standard procedure to change analogue voltage values into digital values, the result is copied into one of these registers. A fifth register, containing the Header, will be accessed in exactly the same way as the registers containing the information coming from the sensors.

10.2 Communication Interface

The PC communicates with the PC-to-AGV-Communication section in the Connection Programme. To understand it better, refer back to section 5.5 "Protocol between Computer and AGV".

Many times during the procedure that is accomplishing analogue-to-digital transformation, the polling procedure is called up. The metal link between the PC and the AGV, containing the clock pulses, is constantly polled by the PIC microcontroller to check if the voltage on it, respective to ground, has changed from positive to negative or visa versa. Determining if the clock has changed is accomplished by comparing it with a bit in a register, where the last bit was made to have the same value as the clocks, as shown in **figure 10.1**. Although polling might not seem to be the best method, it has proved itself to be sufficient for communication between the PC and PIC microcontroller.

The information transmitted from the PIC microcontroller to the PC always comes from the same register in the PIC microcontroller. Although the received information is placed in another register in the PIC microcontroller, the received information is also, initially, always placed in the same register. The two variables in these two registers rotate with every clock pulse that is received; so all the bits in a register are moved to the right. Looking at the variable that contains the transmitted information, the least significant bit is sent out for each change to the clock input. The received bit is inserted into the other register before the rotation takes place.



Polling
What is the value of input "PORTB,3" ?
0
1
What is the value of register "Buffer, 0"?
0
1
What is the value of register "Buffer, 0"?
1
0
Change value in "Buffer, 0" to zero
Change value in "Buffer, 0" to one
Send out a bit
Receive a bit
Continue changing Analogue into Digital

Figure 10.1 Polling Performed to Detect Clock Pulse

The current frame's number of bits that have been transmitted and received are kept in another register. After eight bits have been sent, the register that contains the information to be transmitted is loaded with information coming from another register that is either one of the sensors' information or the Header.

Received information from the Communication Interface is placed directly on a physical bus of the PIC microcontroller. From there it is amplified and sent to the motors to move and steer the Physical AGV.

10.3 Summary

The PIC microcontroller onboard the Physical AGV acts as a slave. Its only function is to capture data from the sensors, transmit it to the PC and then to take information coming from the PC and give it to the interface of the modified servomotors.

This programme changes the analogue voltage values from the infrared sensors into digital values by using the onboard Analogue to Digital Converter inside the PIC microcontroller onboard the Physical AGV.

Polling is used to determine when the clock signal from the PC has changed in order for the next bit to be transmitted and received, and was consequently found to be adequate.

Parallel to serial transition for information from the sensors, and serial to parallel transition for information going to the motors is made possible by rotating the registers and then sending or receiving only one bit of the register.

11 Simulated-Functions-of-the-Microcontroller

Simulated-Functions-of-the-Microcontroller can basically be defined as "controller of all". The instruction set is based on the PIC-Microcontrollers.

Information coming from the sensors is placed in a variable. This information is used and the resulting value is placed in a variable that is used by the AGV to move and steer.

The only difference between the Simulated-Functions-of-the-Microcontroller for the virtual Programme and the Simulated-Functions-of-the-Microcontroller of the Physical AGV is the source of the information that has to be used and the destination of the information generated.

The simulated controller does not have to be exactly the same as the real one, but if it was made as close as possible, the Simulated Programme could be tested on a physical PIC.

It is possible to simulate every transistor in a processor but was not implemented in this research because of the unnecessary level of complexity. Instead, the processor reads from programme memory, data memory and inputs. It then changes part of data memory or an output, as can be seen in **figure 11.1**. Only the functions have to be executed. For example, if the instruction "Move" is in the programme memory, it is not necessary to simulate all the registers, adders and gates it has to go through. All that is needed is to make a copy of some part of the memory and place it in another part. In the same way adding can be executed. It is not necessary to simulate an adder; all that is needed is to add two pieces of information and save it in the data memory. It is easy to see that this will apply to all instructions that have to be carried out.

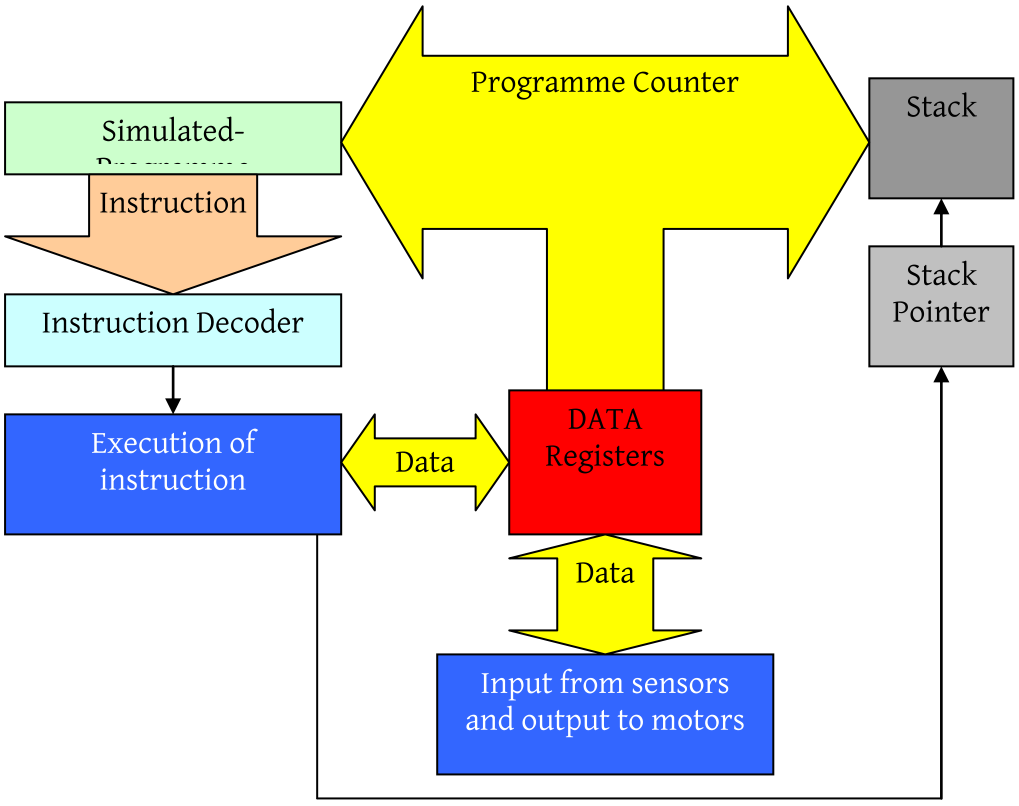


Figure 11.1 Architecture of the Simulated-Functions-of-the-Microcontroller

The time it takes to execute an instruction by the Simulated-Functions-of-the-Microcontroller depends on the time it takes to execute an instruction by the PC used to simulate the Simulated-Functions-of-the-Microcontroller.

11.1 Simulated Programme Memory

For the PIC microcontroller, the programme memory is not the same as the data memory [18, pp.19-20]. The advantage of having two sets of memory is that the data cannot overwrite the programme memory, as in the case of the Z80 [53, pp.20-21 & 46-47 & 42-43]. This array will be called "programme" for short.

Out of a Simulated Programme viewpoint, if Genetic Algorithms are used, it does not matter if all the AGV are simulated at once or individually, because all the programmes have to be in memory and the PC has a variable for each instruction. Each instruction has its own unique address and, in the simulation, the registers are replaced by variables.

Due to the fact that more than one programme is kept in the memory of the PC, the programme array will need two variables to indicate which instruction, out of all the instructions present in all the programmes, is accessed. **Table 11.1** shows a two-dimensional example of the variables' layout. The first variable will indicate the position or address of the instruction in the programme. There are only fifty instructions for each programme, thus this number cannot be higher than fifty. The address of the instruction that has to be executed for that specific programme is held in the Programme Counter, which is part of the Simulated-Functions-of-the-Microcontroller's registers, as will be explained in section 11.2.

Table 11.1 Instruction Layout

Instruction 1 of programme 1	Instruction 2 of programme 1	Instruction 3 of programme 1	Instruction 4 of programme 1	Instruction 5 of programme 1
Instruction 1 of programme 2	Instruction 2 of programme 2	Instruction 3 of programme 2	Instruction 4 of programme 2	Instruction 5 of programme 2
Instruction 1 of programme 3	Instruction 2 of programme 3	Instruction 3 of programme 3	Instruction 4 of programme 3	Instruction 5 of programme 3
Instruction 1 of programme 4	Instruction 2 of programme 4	Instruction 3 of programme 4	Instruction 4 of programme 4	Instruction 5 of programme 4
Instruction 1 of programme 5	Instruction 2 of programme 5	Instruction 3 of programme 5	Instruction 4 of programme 5	Instruction 5 of programme 5

The second variable will indicate the current programme that is evaluated. Each Simulated Programme will have its own number. The only place where this number is changed is in the Programme Generator procedure.

11.2 Simulated Registers

In the PIC microcontroller the data memory, output ports, input ports and the registers are basically the same thing [18, pp.19-38 & 43-55]. A matrix is drawn up where every possible entry functions as a register or data memory. In some PIC's, registers five to nine are used as inputs or outputs. In four of the five variables, the sensor's input values will be placed. The fifth register is used as the output and is then used to control the movement of the AGV. Because the word *register* is an instruction in C++, it is better to use the word *data* for this matrix. The information in the ninth register will be the same as the information described in **table 7.1**.

In a physical PIC microcontroller, even the Programme Counter, a register that determines which address the processor is reading from in the Simulated Programme, is in this register array or data memory part. This particular register is divided into two. The lower part is in register two and the higher part is in register 0Ah, or the tenth register. With each instruction the Programme Counter is incremented by one, unless there is a jump instruction of some sort, in which case the information in the Programme Counter will be changed to that location.

The same layout is applicable here but because there are only fifty instructions in one Simulated Programme, there is no need to use register 0Ah. As one can understand, the Programme Counter has to be reset before a new Simulated Programme can be evaluated.

The W-register and stack-pointer are not part of these registers and have to be simulated by adding extra variables. In the programme, the variables maintain their names. Also, the stack, which is a memory array much like programme or data memory, has to be simulated and this is done in the same way as, for example, data memory, only much smaller. Ten registers are necessary.

If a lot of AGV are all simulated at the same time, then there has to be a data memory, a stack, a stack pointer, and a W-register for each. If the AGV are simulated one after the other, as they are implemented in this dissertation, then there is only need for one data memory array, stack array, stack-pointer, and W-register which are then reused for every "new" AGV. Only programme memory is needed for each AGV.

11.3 Instruction Decoder

The Instruction Decoder determines which instruction should be executed by the Simulated-Functions-of-the-Microcontroller. In the memory, most of the PIC's encoded instructions are divided into two parts [18, p.147]. The first part is the instruction, shown as yellow in the sketch below, and the second is a quantity or operand, shown as green in the sketch. As an example, the instruction "MOVLW" is examined. It normally moves a literal value into the W-register [18, p.154]. Although the encoded instruction is always fourteen bits in length, the size of the operand, and the size of the instruction part, is not constant for each instruction.

OPCODE   Operand

1100 10101010101010

Getting part of the encoded instruction is accomplished by using the "AND" function in C++ in collaboration with a constant. With the "AND" function, whenever there is a one in both registers, the resulting bit would be a one. Other than that the output would be zero. It is thus possible to activate a bit by placing a one in the place where one wants the input bit in the other register to be the same as the output bit. If a zero is placed in the position, it is similar to deactivating the bit because the output of that bit will be zero no matter what the value in the other register is at that particular bit's position.

The example of the "MOVLW" instruction is used again. The first four bits of the total instruction in programme memory are used to indicate that the instruction is "MOVLW". Therefore, the incoming instruction has to be "AND" with the binary code 11110000000000 (15360 decimal).

```
1100 10101010101010  AND  11110000000000  =>  11000000000000
        12970          AND      15360        =>       12288
```

In the sketch, the green part has been thrown away. The first four bits' weights are 8192 for bit 14, 4096 for bit 13, 2048 for bit 12 and 1024 for bit 11. All these weights have to be added to get the decimal value of the binary value shown. The incoming instruction has to be "AND", with a decimal value of 15360.

With the first four bits removed from the incoming instruction, it has to be tested to see if it corresponds with the instruction 1100, which is the binary value of 12288 for the "MOVLW" instruction.

Many tests have to be performed on the extracted binary code, and each test is there to test for one particular instruction. That is why the "case" function in C++ is used rather than the "if" function.

Testing for instructions that are more than four bits in length is accomplished by performing another "and" and "case" function in C++. The bit-test functions and the jump instructions of the Simulated-Functions-of-the-Microcontroller change the information in the Programme Counter. These instructions must be handled with caution. Assume a jump has to occur to address ten, the Programme Counter is currently at address five, and all the instructions have not been tested for by the Virtual Programme. The next test for instructions, in this cycle, will then be on an instruction coming from address ten when it should be on an instruction coming from address five. The change in the Simulated-Functions-of-the-Microcontroller's Programme Counter's information must be performed when no further instruction tests are going to be performed. One would think that this change has to be incorporated at the end of the test, but there are other ways of doing this. It could be changed before all the tests, but it must not change during the decoding procedure.

A combination of ones and zeros can only signify one instruction. It is thus unnecessary to test for more than one instruction in a cycle. There are advantages to testing only once for an instruction and if an instruction is carried out, no more testing

is executed. The advantages are as follows:

- It saves time
- The Programme Counter's information can be changed at any time during the cycle if being dictated to by the Simulated Programme.

The “switch” command in C++, in association with the “break” command, will only test until the instruction is identified. The problem exists that, because the sizes of the OPCODE instruction part are not the same for all instructions, there is more than one “switch” command necessary [18, p.147]. The programme will go through some or all of the tests in the first “switch” section, depending on if it has identified a Simulated Programme instruction or not. It does not matter if a Simulated Programme instruction was identified or not, the second “switch” will go through some or all of its tests again. This is a problem if there was an identification of an instruction in the first “switch” section, because it is then not necessary to do the second “switch” section. This test to identify the current instruction, after the current instruction has already been identified, is an unnecessary waste of time.

The “switch” command can also be used in association with the “default” command. The “default” section will only be executed whenever there is no earlier identification in the “switch” section. It is almost similar to the “else” function that follows the “if” function. By placing the whole second “switch” section in the “default” section of the first “switch” section, the second “switch” section will only be executed if the first “switch” section did not execute any instruction. If necessary, a third and fourth “switch” instruction can be added in the same way.

In the Simulated-Functions-of-the-Microcontroller, the “BTFSS” instruction, tests a bit to see if it is a one or a zero. If it is one then the processor skips the next instruction. If not, it executes the next instruction. For this function, two operands are needed. The first is to show in which register the bit is and the next indicates the bit in the register. Therefore, the “AND“ function in C++ is used three times in this function.

The operand that determines the bit that has to be tested is in the middle of the total instruction. To save time, this part is not moved to the least significant part of the register so as to make its value between zero and seven. Its value is calculated for what it should be when in that position, an "IF" command in C++ is then used, in conjunction with this value, to execute a test on a specific bit in that register.

In the case where there is only a fixed amount of instructions that all have to be tested, it will not be necessary to test for the last one. If all had been tested except for the last one, there can only be one correct component, thereby allowing one to use the "default" instruction.

If an instruction is executed at the beginning of the testing part, the cycle will not take long to execute an instruction. The problem is when the instruction that has to be executed is at the end of the decoding procedure. The following method can be used to overcome this problem.

The most significant bit in the instruction can only be a one or a zero. The simulator can be programmed to first test this bit. If the bit is one, then half of the instructions have to be tested, thus, all instructions beginning with a one. The same method is followed when the first bit is zero. Thus, by adding one test to the procedure, the amount of tests is halved. The second most significant bit can also only be a one or a zero. For each of the two results, a second test can be executed. These tests split the total amount of instructions into four, as illustrated in **figure 11.2**. There are three tests. The first test tests the most significant bit. The second and third both test the second most significant bit. The second test is only executed when the first bit is a zero and the third test is only executed when the most significant bit is a one.

Although there are three test sections, only two are executed at any given stage. With the addition of only one more test, the number of tests that still have to be executed is again halved, so only a quarter of the tests, plus two more, have to be performed to determine what the instructions is.

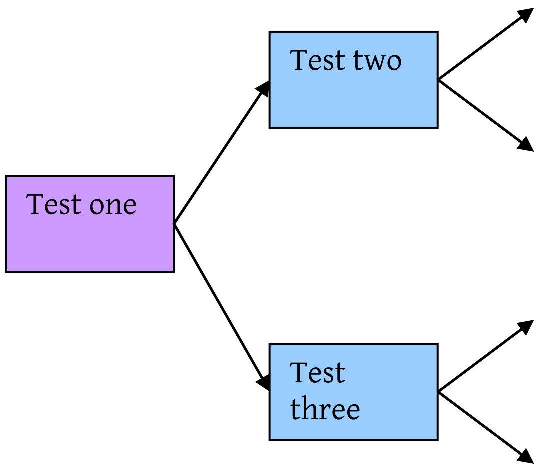


Figure 11.2 The Instruction Decoder forms a "Decoding Tree"

Every bit that indicates the instruction but not the operand can be tested for in this way. The Instruction Decoder, dictated by an instruction, is sort of directing the programme to the wanted sub-procedure.

11.4 Executing an Instruction

As explained in the intro section of this chapter, the "MOVE" instructions are easy to execute. The function "MOVF" influences the zero status bit. The result can be placed in one of two places depending on the eighth bit in the encoded instruction. The results have to be tested by the Virtual Programme to see if the result is zero.

As on the PIC microcontroller, the zero bit of the Simulated-Functions-of-the-Microcontroller is placed on bit two, the third bit in the Status Register. It is easy to change a bit in a register providing that it does not matter what the other bits in the register have to be. It gets to be more complicated if the other bits in the register must not be changed. By using the "AND" function in collaboration with the binary value of 11111011, the zero bit will be changed into zero. This must happen when the result from the executed Simulated Programme's instruction is not zero.

Changing the zero flag into a one without changing the rest of the binary code 00000100 can be accomplished by the "OR" function. As can be seen, the two codes are inverts of one another. This is the method that will be used to change any single bit in any register. More than one bit can also be changed in this way if the bits altered are changed to the same status.

The "ADDLW" function affects not only the zero flag in the status register but also the carry and DC flags. The DC bit is set if a carry has occurred between bit four and bit five and the carry bit is set when the result does not fit in eight bits.

This can be accomplished by using one of two methods.

- One must first concentrate on the least significant four bits of the operand or value that is going to be used. To get this, the "AND" function is used again. This has to be implemented to both the values that have to be added together. If the result is more than fifteen, then a carry occurs and the DC bit has to be set, sixteen has to be subtracted from the temporary result, and, in the future, one has to be added to the higher four bits. The same steps have to be performed on the most significant four bits, but before one can do that the four bits do not only have to be removed from the original, but also have to be moved to be the least significant four bits in the register. This is accomplished by dividing it by sixteen. Except for maybe a one that has to be added to the result if a DC carry occurred, the calculations are the same as for the least significant four bits. After all that, the result still has to be moved back to the most significant four bit position before being added to the least significant four bits.

- In the second method, to register that a carry has occurred, the two values are added together. If the value is more than 255, the most that an eight bit bus can carry, then 256 is subtracted, and the carry bit is set. The subtraction takes place to make the amount fit into the eight bits. This will give the result in a single procedure. The DC can be calculated in the same way as in method one and has to be executed before the result is calculated, because the result is placed in the "W_register", which is used in the beginning as an input register. The second method has fewer computations and is easier to construct. Sometimes sixteen has to be subtracted from the result, although the maximum number a four bit digit can hold is only fifteen. This process is used because, if one is added to fifteen, then the four least significant bits must contain zero and not one.

The "GOTO" command is an easy command to construct. The information given is just loaded into the Programme Counter. The only problem is that, with each cycle, the Programme Counter is incremented by one to point to the next address if a jump was not made. Therefore, to go to the required address, one must subtract one from the given address to get to the correct address.

An important function for the PIC, and other processors, is the command that does the subtraction. It is used to calculate the difference between two values or pieces of information. Flags bits are to be cleared or set according to this relationship between the two initial values. In the PIC's case, the Flag bits are part of the third register known as the STATUS register. By using the "BTFSS" and the "BTFSC" instructions to determine the status of the specific Flag bits, the Simulated Programme can alter its course.

The bits that are influenced by the subtraction instructions are the Carry bit and the Zero bit [18, p.160-161]. If the Carry bit is set, then the result is positive or zero, if the carry bit is cleared the result is negative. The Zero bit is only set if the result is zero.

If a result of a calculation is smaller than Zero, then the result must be the 2's-compliment. To get the 2's-compliment, the result must be subtracted from 256. If no flags are considered in a processor and just an eight-bit bus is used, then two-hundred-and-fifty-six is the same as zero.

The "CALL" command is one of the most complicated commands. In the case of the PIC it is the only instruction, except for "RETURN", which uses the Stack registers and the Stack Pointer.

Every time a "call" is made, the current address of the "CALL" instruction is placed in one of the registers in the Stack array. If a "RETURN" instruction is executed, then the information in the register in the Stack is placed back in the Programme Counter. The difference between the "GOTO" instruction, the "CALL" and the "RETURN" instruction is, with the "GOTO" instruction, the Simulated Programme always goes to the address specified, whereas when the "RETURN" instruction is executed, the Simulated Programme will go to the address following the address where the "CALL" instruction was executed.

A “CALL” can build up a procedure. This procedure can be used in a programme in the same way as one would an instruction. The difference is that the programmer develops this “instruction”.

The parts where the execution of the call is made, the part in front of the “RETURN” instruction, is called a procedure or sub procedure. A “call” can be made within a sub procedure. To do this, more than one variable in the Stack is needed. After the “CALL” instruction has placed the address on Stack, the Stack-Pointer is incremented by one to show to the next stack position. When the next “CALL” is executed, the address will be placed in the next register pointed to by the Stack Pointer before it is incremented again.

To get back the correct address, when a “RETURN” instruction is executed, the Stack-Pointer must be decremented by one. This will cancel out the increment that has been brought about by the “CALL” instruction.

All the instructions that increment and decrement a register have to be changed to zero when attempting to go over 255, and to 256 when attempting to go below zero.

11.5 Summary

The Simulated-Functions-of-the-Microcontroller has the same instruction set at the PIC microcontroller and, thus, a programme generated by the Virtual Programme will most probably work on a physical PIC microcontroller.

Instructions are stored in a Simulated Programme. It was decided to have a population of twenty, thus, there is a total of twenty Simulated Programmes within the Virtual Programme. Each Simulated Programme is made up of fifty instructions. All instructions are kept in an array that has a length of fifty and a width of twenty.

Because there is only a one dimensional *register* array, each Simulated Programme has to be executed individually. As with the PIC microcontroller, the second register contains the address of the instruction that has to be executed. This information, with the number of the individual in the population that has to be executed, is inserted into the *instruction* array to determine the current instruction that has to be executed.

The instruction decoder determines which combination of C++ instructions has to be executed to simulate the execution of the instruction coming from the Simulated Programme. In order to save time, the instruction decoder is made in a tree configuration. The execution of an instruction either tests, or makes changes to some part of the register array. The change in some of the registers eventually changes the movement of the motors.

12 Results

In this chapter the resulting fitness levels will be given and investigated. Observations and possible reasons for the outcome are made. Examples of Simulated Programmes that were generated by different means, illustrating obstacle avoidance abilities, are analysed and an example of a Simulated Programme that did reach the destination will be looked at. The hardware and the software that was developed, and their future abilities, will be illustrated. The conclusion indicates the effectiveness of Single-Chromosome-Evolution-Algorithms, compared to Genetic Algorithms, for generating a programme.

12.1 Resulting Fitness Levels

Because the use of the random function affects the results, it is not possible to predict how long it will take to accomplish a specific function or what the outcome will be. It is not even possible to determine if the required result will be met. For this reason, a lot of simulations were executed to get an average for some option.

All the values in the tables below have to be multiplied by a hundred to indicate how many simulations it has taken to get to the specified fitness level. It is much more difficult for the Simulated Programmes to adapt to randomly changing mazes than to adapt to static mazes. Therefore, it was decided not to use changing mazes in the results. Maze 2 revealed itself to be the better of the two static mazes and, thus, it was used to determine the results.

To avoid wasting time, while still getting a good result, it was decided to execute 100 000 simulations to generate one Simulated Programme. During the generation of the first Simulated Programme, the AGV stood still for a while at the following fitness levels: 42, 96 and 290. These fitness levels were then used as beacons for the following results. Additional beacons of 400, 500, 600, and 700 were added, but because the fitness level of 500 was only exceeded in 34.7% of the simulations, only the beacon 400 is shown here. In the tables, *Start* shows how many simulations it took before the AGV started moving, and it functions as the first beacon.

Method indicates what kind of method was used to generate the programme. SCEA, in the table, shows that Single-Chromosome-Evolution-Algorithms were used; *Mutate* is for Genetic Algorithms that use only mutation and *Cross* indicates that Genetic Algorithms, that implement mutation as well as crossover, were used.

If *Environment* is made *yes,* then the simulation of changes in the environment's light intensity has been activated. As already stated, the programme has to learn how to execute an instruction from a user. When *Split* is made *yes* in the table, it indicates that the AGV was trying to learn to react to an input from a user. A single bit indicates which one of the two destinations the AGV should go to. On analysing the results, the observation was that the results seem to split when the split option was activated. The programme was then changed to reveal what the individual results were for the AGV attempting to go to each destination. *Amount of Instructions* indicate how many instructions are updated for each mutation.

Although Identification Trees are not used in the experiments, a modified version is used here to analyse the results [30, p.423-442]. As with Identification Trees, the simulation options were randomly executed. Because Single-Chromosome-

Evolution-Algorithms are to be evaluated against Genetic Algorithms, the most simulations were executed to show the difference between these two options.

All relevant options had to be simulated at least once. As already stated, the AGV is suppose to be most flexible when only one instruction is updated per mutation, thus, most results were determined using one instruction update per mutation. Majority of the results generated are shown in **table 12.1**.

Table 12.1 Resulting Fitness Levels

Metho d	Environmen t	Split	Number of Instructions	Star t	42	96	290	400	Maximum Fitness Level	Comment
SCEA	Yes	No	1	3	4	4	4	78	468	
SCEA	Yes	No	1	12	12	13	13	52	510	
SCEA	Yes	No	1	18	29	35	93	NA	296	
SCEA	Yes	No	1	11	12	12	14	68	501	
SCEA	Yes	No	1	4	122	125	NA	NA	169	
SCEA	No	No	1	12	15	28	42	58	477	
SCEA	No	No	1	3	3	3	94	NA	361	
SCEA	No	No	1	4	4	5	19	NA	337	
SCEA	No	No	1	3	4	6	6	NA	367	
SCEA	No	No	1	4	12	12	20	236	499	
SCEA	No	No	1	4	5	11	16	160	453	
SCEA	Yes	No	2	3	4	4	8	45	510	
SCEA	Yes	No	2	2	3	3	14	14	742	

Table 12.1 Resulting Fitness Levels (Continued)

Method	Environment	Split	Number of Instructions	Start	42	96	290	400	Maximum Fitness Level	Comment
SCEA	No	No	2	1	2	2	6	23	699	
SCEA	Yes	No	4	2	2	2	2	4	749	
SCEA	No	No	4	2	2	28	37	93	545	
SCEA	Yes	Yes	1	1 15	15 15	19 209	279 NA	NA	182 49	Lower fitness level
SCEA	No	Yes	1	1 3	7 16	7 19	928 NA	NA	39 311	Register
Cross	Yes	No	1	150	150	152	348	NA	316	
Cross	Yes	No	1	228	234	241	291	323	466	
Cross	Yes	No	1	80	150	442	NA	NA	101	
Cross	Yes	No	1	207	208	218	242	NA	295	
Cross	Yes	No	1	391	391	391	405	440	313	Lower fitness level
Cross	Yes	No	1	324	475	478	482	NA	327	Non constant
Cross	Yes	No	1	134	144	144	226	NA	325	
Cross	Yes	No	1	133	195	219	220	224	458	
Cross	Yes	No	1	107	113	113	128	NA	309	Slow
Cross	No	No	1	153	175	175	NA	NA	208	
Cross	No	No	1	184	188	190	578	NA	331	

Table 12.1 Resulting Fitness Levels (Continued)

Method	Environment	Split	Number of Instructions	Start	42	96	290	400	Maximum Fitness Level	Comment
Cross	No	No	1	241	244	244	820	NA	300	Lower fitness level
Cross	No	No	1	104	107	107	292	NA	328	Lower fitness level
Cross	No	No	1	155	155	155	159	NA	332	Lower fitness level
Cross	No	No	1	97	98	157	354	NA	327	
Cross	No	No	1	46	208	211	305	NA	366	
Cross	No	No	1	326	327	333	370	NA	318	Slow
Cross	Yes	No	2	40	40	69	196	NA	386	
Cross	No	No	2	49	65	65	65	747	474	
Cross	Yes	No	4	32	32	32	58	91	506	
Cross	No	No	4	19	20	20	20	134	457	
Cross	Yes	Yes	1	46 78	82 102	93 294	118 NA	NA NA	100 247	Not registered
Cross	No	Yes	1	85 108	88 119	105 124	634 790	NA NA	197 316	Registered
Mutate	Yes	No	1	91	97	165	175	NA	298	
Mutate	Yes	No	1	93	97	118	142	168	480	
Mutate	Yes	No	1	176	291	339	741	NA	331	

Table 12.1 Resulting Fitness Levels (Continued)

Method	Environment	Split	Number of Instructions	Start	42	96	290	400	Maximum Fitness Level	Comment
Mutate	Yes	No	1	51	51	51	149	NA	373	
Mutate	Yes	No	1	76	105	281	293	NA	176	Backwards
Mutate	No	No	1	59	60	60	71	946	331	Lower fitness level
Mutate	No	No	1	109	569	619	690	965	456	
Mutate	No	No	1	78	78	79	NA	NA	361	
Mutate	Yes	No	2	30	30	30	76	193	508	
Mutate	No	No	2	54	54	54	84	222	510	
Mutate	No	No	2	24	24	36	139	908	465	Slow
Mutate	Yes	No	4	11	16	18	104	NA	350	
Mutate	No	No	4	14	21	24	32	169	654	Second best
Mutate	Yes	Yes	1	12	51 51	140 266	267 NA	NA NA	176 279	Registered instruction
Mutate	Yes	Yes	1	24 78	131 133	135 182	486 NA	NA NA	238 98	Not registered
Mutate	No	Yes	1	47 47	152 185	211 211	235 NA	NA NA	312 196	Register instruction
Mutate	No	Yes	1	42 130	222 226	388 NA	NA NA	NA NA	179 21	Registered instruction

In the comment field, the following comments have the following meaning:

- "Lower fitness level" means that the population had a higher fitness level than that of the end result.
- "Register" means that the Simulated Programme was able to register an instruction from a user.
- "Not registered" means the Simulated Programme that was developed was unable to register an instruction from a user.
- "Non-Constant" means that the fitness level of the population fluctuates.
- "Slow" means that the current AGV moved through the maze, at a much slower speed than what the other results, on average, accomplished.
- "Backwards" means that the AGV was moving in a backwards direction.
- "Second best" indicates that the Simulated Programme achieved a high level of fitness.

The AGV that moved backwards rendered a worse result than that of the average results. Slow AGV made the whole simulation much slower and did not necessarily produce a better fitness.

Table 12.2 shows the average results for each option. As an example, the difference between Single-Chromosome-Evolution-Algorithms and Mutation only Genetic Algorithms is calculated. The goal is to calculate the influence that each individual option has on the results. The average of all the options that use Single-Chromosome-Evolution-Algorithms is determined, as shown in **table 12.3**. In the same way, the average for the rest of the options is calculated, as shown in **table 12.3**.

Table 12.2 Average Specific Option

Method	Environment	Split	Number of Instructions	Start	42	96	290	Maximum Fitness Level
SCEA	Yes	No	1	10	36	38	>225	388
SCEA	No	No	1	5	7	11	32	416
SCEA	Yes	No	2	3	4	4	12	626
SCEA	No	No	2	1	2	2	6	699
SCEA	Yes	No	4	2	2	2	2	749
SCEA	No	No	4	2	2	28	37	545
SCEA	Yes	Yes	1	1 15	15 15	19 209	NA NA	182 49
SCEA	No	Yes	1	1 3	7 16	7 19	928 NA	39 311
Cross	Yes	No	1	195	229	269	>371	323
Cross	No	No	1	163	187	197	>485	314
Cross	Yes	No	2	40	40	69	196	386
Cross	No	No	2	49	65	65	65	474
Cross	Yes	No	4	32	32	32	58	506
Cross	No	No	4	19	20	20	20	457
Cross	Yes	Yes	1	46 78	82 102	93 294	118 NA	100 247
Cross	No	Yes	1	85	88	105	634	197

				108	119	124	790	316

Table 12.2 Average Specific Option (continued)

Method	Environment	Split	Number of Instructions	Start	42	96	290	Maximum Fitness Level
Mutate	Yes	No	1	97	128	190	300	332
Mutate	No	No	1	82	236	253	>587	383
Mutate	Yes	No	2	30	30	30	76	508
Mutate	No	No	2	39	39	45	112	488
Mutate	Yes	No	4	11	16	18	104	350
Mutate	No	No	4	14	21	24	32	654
Mutate	Yes	Yes	1	18 45	91 92	138 224	377 NA	137 259
Mutate	No	Yes	1	45 89	187 206	300 >606	>618 NA	246 109

Table 12.3 Influences Different Options

Option	Start	42	96	290	Maximum Fitness Level
SCEA	4	9	14	>52	571
Cross	83	96	109	>199	410
Mutate	46	78	93	>202	453
1 instruction	92	137	160	>333	359

2 instruction	27	30	36	78	530

Table 12.3 Influences Different Options (continued)

Option	Start	42	96	290	Maximum Fitness Level
4 instruction	13	15	21	42	544
Environment included	47	57	102	>149	413
Environment not included	42	64	72	>153	492

The similarity between two values is determined by dividing the smaller of the two values by the bigger. Multiplying the result by a hundred gives a percentage which indicates the similarity in fitness results between the two options, as in **table 12.4**. **Table 12.4** cannot show which one of the two is better, but it can show the similarity in results when an option is changed. The more dissimilar the results are when changing a single option, the bigger the influence of the specific change in the option was. To determine which option gave the better result, revert back to **table 12.3.**

Using crossover to generate a new offspring gives worse results than when it is not used for the generation of a programme, although it does not change the final fitness level a great deal. The change in fitness level caused by two instructions versus four instructions was also not too significant.

Looking at **table 12.3**, it can be seen that Single-Chromosome-Evolution-Algorithms gave better results than any other option, except where four instructions were used to get a fitness level higher than two-hundred-and-ninety.

The problem is that an input from the user was not registered in one of the Simulated Programmes generated by the Single-Chromosome-Evolution-Algorithm, while it was registered when both methods of Genetic-Algorithms were used.

Table 12.4

Method	Start	42	96	290	Maximum Fitness Level
SCEA – Cross	4.8	9.4	12.8	26.1	71.8
SCEA – Mutate	8.6	11.5	15.0	25.7	79.3
Cross - Mutate	55.4	81.3	85.3	98.5	90.5
1 – 2 instructions	29.3	21.9	22.5	23.4	67.7
2 – 4 instructions	48.2	50.0	58.3	53.8	97.4
1 – 4 instructions	14.1	10.9	13.1	12.6	66.0
Environmental changes	89.4	89.0	70.5	97.4	83.9

Percentage Similarity

With this in mind, a change was made to the Virtual Programme where a third fitness level was brought in. The first fitness level was used when the AGV was going to the first destination, the second was for the second destination, and the third was for the result of the Simulated Programme that was evaluated. With this method, the AGV was, at times able to register an instruction from a user.

Biewald said that the autonomous mobile vehicle is vulnerable to inaccuracies in sensors and actuators [15]. The inclusion of simulated, uneven environmental conditions did not change the results a great deal, although it did give slightly better results when environmental conditions were not included.

12.2 Influences on Results

During the simulation, it was found that some behaviour influenced the outcome of the results.

12.2.1 Simulation Time

It was seen, from the simulations, that the better the results were the slower the simulation was. This made sense, since the timeout component had been written in such a way as to give the simulation a specific time to reach a specific fitness level.

Although the fitness level of the current Simulated Programme is comparable to the time it takes to evaluate, it was not an exact calculation, as with the AGV which, at times, stood still for some time before continuing. A second influence on the time was that the AGV had to go around bends, whereas the fitness level is calculated directly towards the destination.

12.2.1.1 Comparing Chromosomes on the Same Obstacle Course

One of the problems experienced with the use of the simulator was that some of the obstacle courses changed into different forms during the same generation. As an illustration, Single-Chromosome-Evolution-Algorithms are used. The original, or in other words, the unchanged chromosome, was not evaluated on the same obstacle course as the changed chromosome. In some cases, the unchanged chromosome has been trapped in some way by the obstacle course at hand. The changed chromosome has not been trapped by its obstacle configuration. Although the original chromosome is overall "better", it will currently have the lower fitness level of the two. The changed chromosome is kept and, thus, the resulting chromosome is worse than the original. A way to overcome this problem is to change the obstacle course only once during a generation, and that is how it has been implemented in this programme.

12.2.2Extinction

In one of the experiments, the one obstacle course's wall was so close to the AGV that it was terminated before it started. At this stage, the obstacle course was only changed once a new generation was generated. Genetic Algorithms were used to generate the new generation.

The whole generations' individuals had a fitness level of zero, thus leading to changes being made to chromosomes or generated programmes that did not need to change. The result was that the overall fitness level did not get off the ground. By simply slightly changing the initial direction of the virtual AGV this problem was overcome.

12.3 Analysing the Generated Programme

All the programmes shown in this section are programmes that were generated, in one way or another, by the Virtual Programme. They showed the ability to avoid obstacles. The different colours in the programme segments have the following meanings:

- Blue represents instructions that were only executed once
- Green represents instructions that were executed more than once
- Black represents instructions that were not executed

As can be seen, only the green instructions were responsible for the outcome. The blue instructions ensured the green instructions were executed; however, the blue and black instructions have no effect on the programme.

The programme in **programme segment 12.1** built up a fitness level of 500 in less than 21800 simulations, making it one of the best results generated using these options. It was generated using the following options:

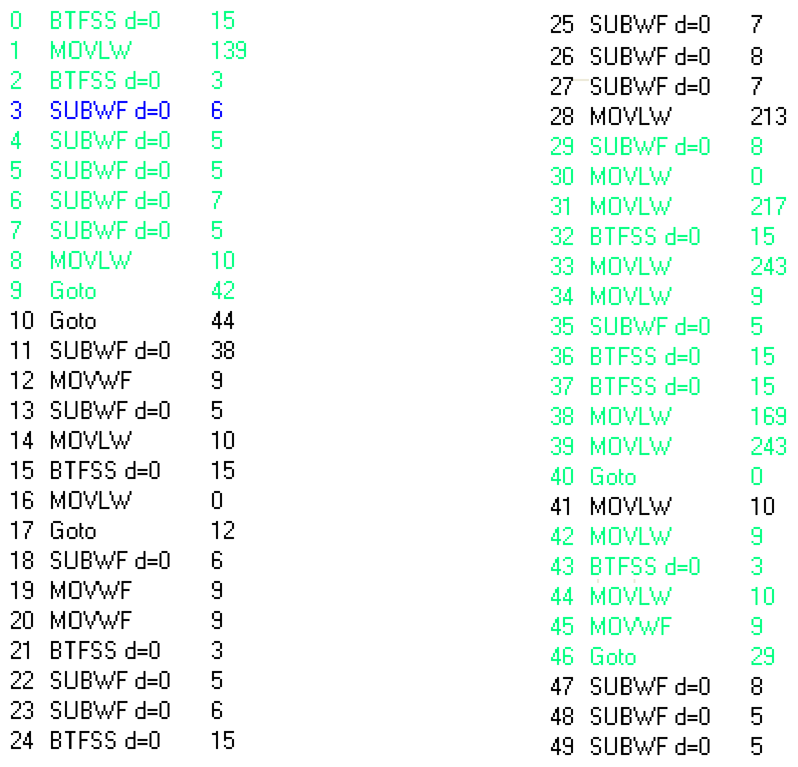
- Only one instruction is updated per mutation

- Genetic Algorithms, with crossover, are used
- Environmental changes are not included
- It is not made to detect an instruction from a user

The instructions at addresses 30, 31 and 33, load information into the W-register. It is then reloaded with a new value by the instruction at address 35 [18, p.156]. This loading of the W-register is never used. It also does not affect the status registers, and thus, although it is executed, does not affect anything in the programme.

Instructions at address' 4, 5, 6 and 7 simply change the result in the W-register, where it is, in any case, changed into a constant at address 42 afterwards. The difference is that the instruction at address 7 changes the Flags in the status register, while the instruction at address 42 does not change the Flags. This change in the Flags changes what the instruction at address 43 will do. At this stage, the W-register is loaded with 9 by the instruction at address 42. By skipping the instruction at address 44, the value of 9 is sent out to the motors by the instruction at address 45. The reaction of the AGV is to turn to the right.

Address	OPCODE	Operand	Address	OPCODE	Operand



```
0   BTFSS d=0    15
1   MOVLW        139
2   BTFSS d=0    3
3   SUBWF d=0    6
4   SUBWF d=0    5
5   SUBWF d=0    5
6   SUBWF d=0    7
7   SUBWF d=0    5
8   MOVLW        10
9   Goto         42
10  Goto         44
11  SUBWF d=0    38
12  MOVWF        9
13  SUBWF d=0    5
14  MOVLW        10
15  BTFSS d=0    15
16  MOVLW        0
17  Goto         12
18  SUBWF d=0    6
19  MOVWF        9
20  MOVWF        9
21  BTFSS d=0    3
22  SUBWF d=0    5
23  SUBWF d=0    6
24  BTFSS d=0    15
```

```
25  SUBWF d=0    7
26  SUBWF d=0    8
27  SUBWF d=0    7
28  MOVLW        213
29  SUBWF d=0    8
30  MOVLW        0
31  MOVLW        217
32  BTFSS d=0    15
33  MOVLW        243
34  MOVLW        9
35  SUBWF d=0    5
36  BTFSS d=0    15
37  BTFSS d=0    15
38  MOVLW        169
39  MOVLW        243
40  Goto         0
41  MOVLW        10
42  MOVLW        9
43  BTFSS d=0    3
44  MOVLW        10
45  MOVWF        9
46  Goto         29
47  SUBWF d=0    8
48  SUBWF d=0    5
49  SUBWF d=0    5
```

Programme Segment 12.1 Only One Instruction has been changed

If the instruction at address 44 is executed, then the W-register will be reloaded with the value of 10, before being written out to the motors. This will make the AGV move forwards.

In section 7.1.1.2, it was seen that register 5 contains the information of the sensor that is facing forwards, and that register 7 contains the information of the sensor facing to the right. In this programme, the combined values of three times the value of the information coming from the sensor facing forwards and one times the value of the information coming from the sensor facing to the right of the AGV are subtracted from the value 139. The instructions from address 29 up to address 40 do not seem to have much of a function.

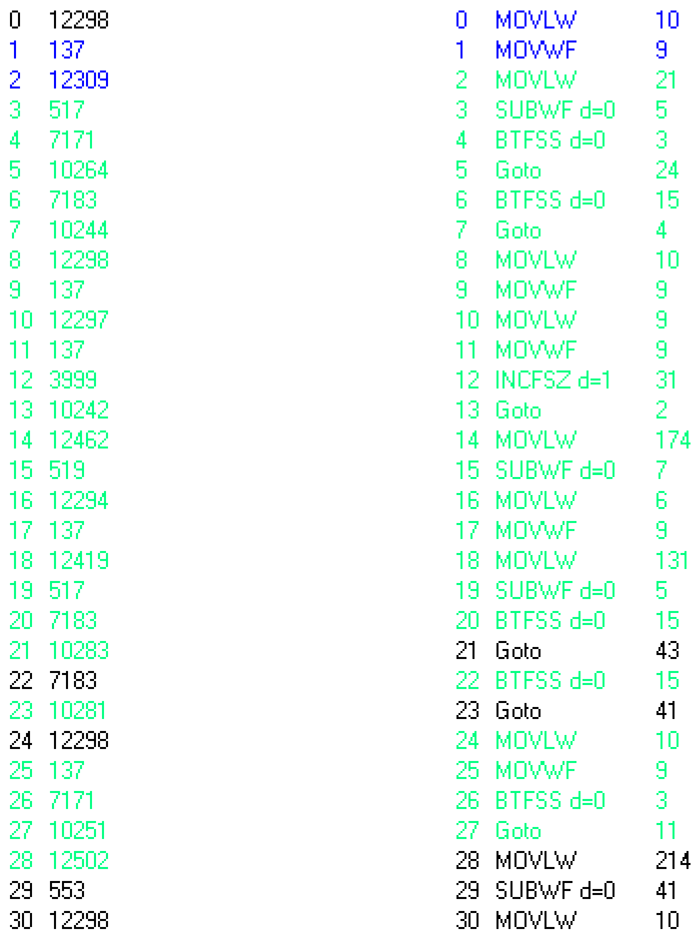
The Simulated Programme in **programme segment 12.2** reached a fitness level of 412 in less than 4700 simulations. It was generated using the following options:

- Two instructions were updated for one mutation
- Mutation Only Genetic Algorithms were used to generate this programme
- A simulation of the changes in the infrared light intensity of the surroundings was included
- It was not made to detect an instruction from an external user

Clearly the instructions at address' 20 and 22 have no function, as they always cause the programme to skip the next instruction. The instruction at address 26 always allows the next instruction to be executed, thus, it too is accomplishing nothing. The instruction at address 14 and 15 changed the values in the W-register as well as the flags in the status register. Without being used, the value in the W-register was again changed into 6 at address 16. The instruction at address 19 changed the flags before it was used, thus, the instructions at address 14 and 15 were not influencing the programme.

As explained in section 7.1.1.2, it takes 40 instructions to give an output to the motors. The instructions at address' 8 and 9 attempted to make the AGV move forward. Directly following these instructions, the instructions at address' 10 and 11, were instructions that attempted to make the AGV turn right. From one output attempt to the next, it took just two instruction cycles. In short, although the instructions at address' 8 and 9 would ensure that the AGV, in normal circumstances, would move forward, in this case it does little if anything.

Address	Instruction		Address	OPCODE	Operand



```
0   12298                          0   MOVLW       10
1   137                            1   MOVWF       9
2   12309                          2   MOVLW       21
3   517                            3   SUBWF d=0   5
4   7171                           4   BTFSS d=0   3
5   10264                          5   Goto        24
6   7183                           6   BTFSS d=0   15
7   10244                          7   Goto        4
8   12298                          8   MOVLW       10
9   137                            9   MOVWF       9
10  12297                          10  MOVLW       9
11  137                            11  MOVWF       9
12  3999                           12  INCFSZ d=1  31
13  10242                          13  Goto        2
14  12462                          14  MOVLW       174
15  519                            15  SUBWF d=0   7
16  12294                          16  MOVLW       6
17  137                            17  MOVWF       9
18  12419                          18  MOVLW       131
19  517                            19  SUBWF d=0   5
20  7183                           20  BTFSS d=0   15
21  10283                          21  Goto        43
22  7183                           22  BTFSS d=0   15
23  10281                          23  Goto        41
24  12298                          24  MOVLW       10
25  137                            25  MOVWF       9
26  7171                           26  BTFSS d=0   3
27  10251                          27  Goto        11
28  12502                          28  MOVLW       214
29  553                            29  SUBWF d=0   41
30  12298                          30  MOVLW       10
```

Programme Segment 12.2 Simulated Programme for Two Instruction Change

There are eight instruction cycles between address's 16 and 17, and address's 24 and 25. There were no "turnoffs" between the two sets of address. The instructions at address' 16 and 17 were also unable to have any influence as the output is again changed before 40 instructions have elapsed.

In section 7.1.1.3, it was seen that the first bit in register 15 acts as an input, either from a user or to indicate on which side of the AGV the destination is. Because the split option was not activated when the programme in **programme segment 12.2** was generated, this bit was used to indicate on which side of the AGV the destination was. The testing of this bit in this programme was performed at address' 6 and 7.

The instruction at address 12 will only cause the programme to skip the "GOTO" instruction at address 13 once every two-hundred-and-fifty-six time it is executed.

Figure 12.1 shows the equivalent flowchart for the Simulated Programme in **programme segment 12.2**. The loop formed by testing the directional bit lets the AGV continue with its current operation until the destination is at the desired side of the AGV.

The Simulated Programme in **programme segment 12.3** has reached a fitness level of 506 using the following options:

- Four instructions were updated for every mutation performed
- Genetic Algorithms, using mutation as well as crossover to generate the offspring, were used to generate this programme
- A simulation of the changes in the infrared light intensity of the surroundings was included
- It was not made to detect an instruction from an external user

Because some functional blocks only use two instructions, the other instructions of the four instructions were converted into "NOP" instructions that have no purpose other than to delay the process. The flowchart of the Simulated Programme shown in **programme segment 12.3** is shown in **figure 12.2**.

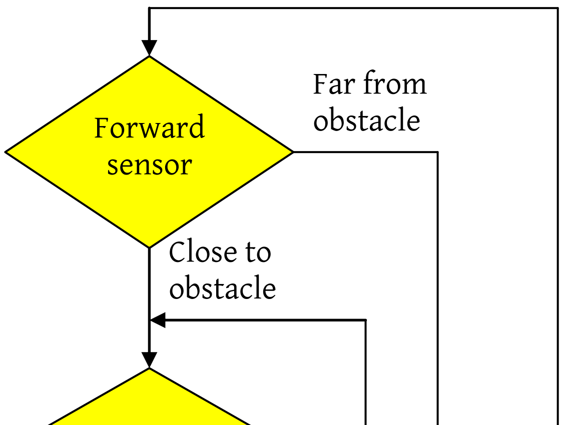
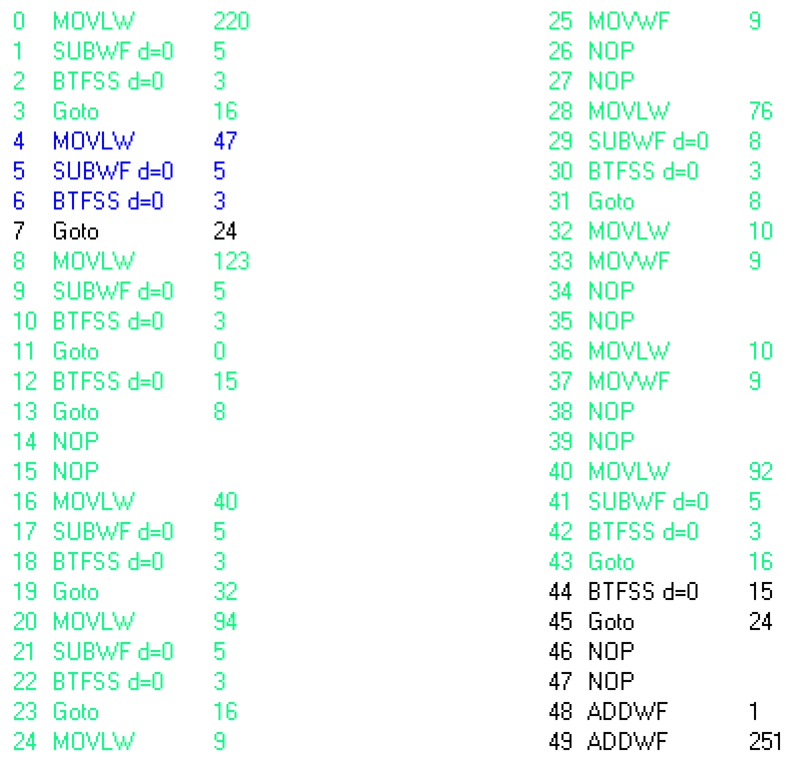


Figure 12.1 Simulated Programme Generated, Two Instructions are Updated

When two instructions and four instructions are updated per mutation, the Programme Generator does not reveal the ability to generate a programme where more than one subtraction is executed on one value, as was accomplished when one instruction was updated per mutation. In the first Simulated Programme, the

"BTFSS" instruction was directly followed by "MOVLW" instruction, both instructions making sensing and reacting to an obstacle possible. When more than one instruction is used per mutation, the "BTFSS" instruction is always followed by a “GOTO” command. This is the proof that the option of updating one instruction per mutation is more flexible than the other two options.



Address	OPCODE	Operand	Address	OPCODE	Operand
0	MOVLW	220	25	MOVWF	9
1	SUBWF d=0	5	26	NOP	
2	BTFSS d=0	3	27	NOP	
3	Goto	16	28	MOVLW	76
4	MOVLW	47	29	SUBWF d=0	8
5	SUBWF d=0	5	30	BTFSS d=0	3
6	BTFSS d=0	3	31	Goto	8
7	Goto	24	32	MOVLW	10
8	MOVLW	123	33	MOVWF	9
9	SUBWF d=0	5	34	NOP	
10	BTFSS d=0	3	35	NOP	
11	Goto	0	36	MOVLW	10
12	BTFSS d=0	15	37	MOVWF	9
13	Goto	8	38	NOP	
14	NOP		39	NOP	
15	NOP		40	MOVLW	92
16	MOVLW	40	41	SUBWF d=0	5
17	SUBWF d=0	5	42	BTFSS d=0	3
18	BTFSS d=0	3	43	Goto	16
19	Goto	32	44	BTFSS d=0	15
20	MOVLW	94	45	Goto	24
21	SUBWF d=0	5	46	NOP	
22	BTFSS d=0	3	47	NOP	
23	Goto	16	48	ADDWF	1
24	MOVLW	9	49	ADDWF	251

Programme Segment 12.3 Simulated Programme for Four Instructions Change

The sequence of instructions that is executed is also not what one might expect. It seems that jumps occur from any address in the Simulated Programme to any other address. In conclusion, the instructions used in the programme are intertwined between the instructions not used.

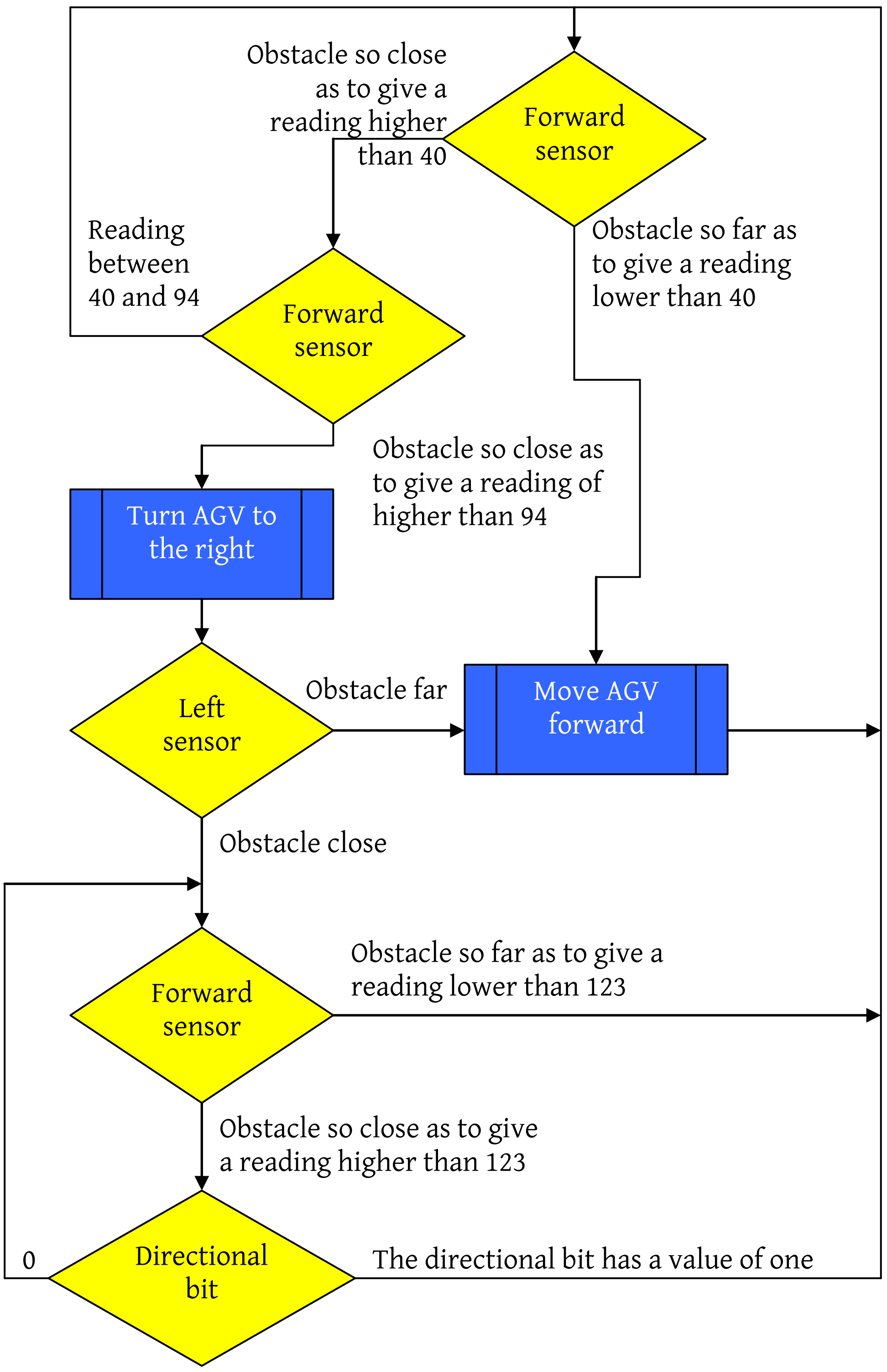

Forward sensor
Obstacle so close as to give a reading higher than 40
Obstacle so far as to give a reading lower than 40
Reading between 40 and 94
Forward sensor
Obstacle so close as to give a reading of higher than 94
Turn AGV to the right
Left sensor
Obstacle far
Move AGV forward
Obstacle close
Forward sensor
Obstacle so far as to give a reading lower than 123
Obstacle so close as to give a reading higher than 123
0
Directional bit
The directional bit has a value of one

Figure 12.2 Flowchart of Simulated Programme when Four Instructions are Changed

Some of the instructions that have been altered within the programme do not have any effect on the result. The Programme Generator is written is such a way that if the fitness level stays the same, or if the fitness level is higher, the change is kept.

This means that these instructions are changing all the time, although the fitness is not necessarily changing. It is possible that, at some point, an instruction that is used is changed in such a way as to make some of these unused instructions start affecting the simulation. Therefore, a procedure can be a sudden advantage or disadvantage to the whole simulation. Changing only one instruction can activate a "whole new programme" or "procedure in the programme". Because many instructions will have to be one hundred percent correct, the chance of this being an asset is unlikely, but possible, and the more simulations executed the more chances there will be that it is an advantage.

12.4 Results on Single Chromosome Evolution Algorithms

Getting a worse result when environmental conditions are included indicates that the more difficult the task is, the worse the result will be. Because Single-Chromosome-Evolution-Algorithms "learn" faster than Genetic-Algorithms, they can be seen as more "intelligent" for a specific condition. However its inability to react to an input from a user indicates that it is less "intelligent" in some applications.

Within a changing environment, for instance, if the obstacle course changed regularly, the Single-Chromosome-Evolution-Algorithms seem to wait for the ideal situation. An example of this is when the obstacle course is changed regularly. The results will only have a high fitness level if one specific obstacle course is used during the simulation. Because the fitness level is mostly made higher when that obstacle course is simulated, it only adapts to that specific course. This fault can be overcome by simulating the AGV in different options and getting an average of the fitness level before the chromosome is changed. That will lead to more simulations and one may question why one would use it as opposed to using normal Genetic Algorithms.

Strangely enough, the AGV was able to, at times, overcome local maximum problems when using Single-Chromosome-Evolution-Algorithms. The reason for this might be in the generation of a programme, as some parts of the programme can suddenly be made active, after being changed in many forms while staying inactive, as was seen in section 12.3.

12.5 Reaching the Destination

Figure 12.3 was generated with the aid of Single-Chromosome-Evolution-Algorithms. At first, the Simulated AGV ran into a specific wall. It seemed that the AGV would not ever be able to do any better, and then suddenly a huge jump in the fitness level occurred. The AGV almost reached the destination. At the end, it avoided one wall and ran into another. At this stage, the AGV was almost directly on top of the destination.

Things must have been easier in this simulation, because environmental changes were not included. In other simulations, were Genetic Algorithms with crossover were used, a fitness level of this magnitude never occurred, not even if environmental changes were not included.

Figure 12.4 shows the fitness level versus the amount of simulations for the result obtained in **figure 12.3**. The values below the graph have to be multiplied by a hundred to show the amount of simulations executed.

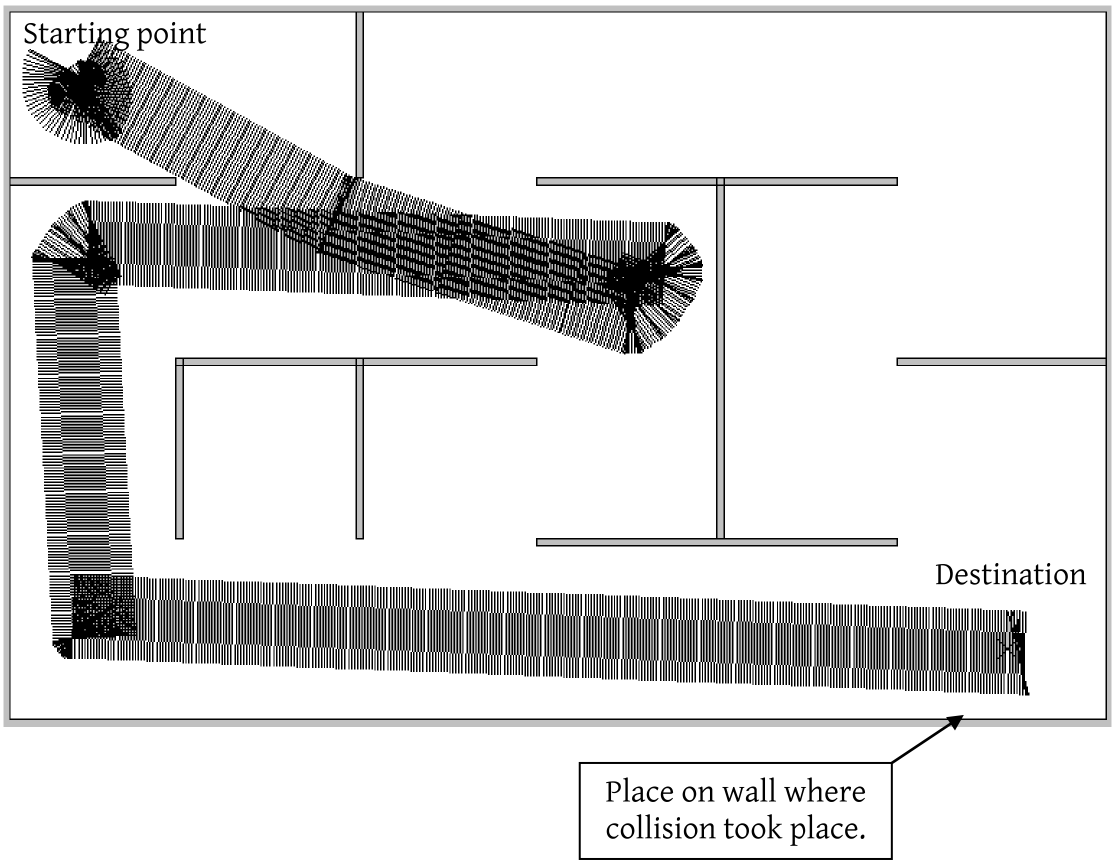


Figure 12.3 Route of AGV when Destination has been Reached

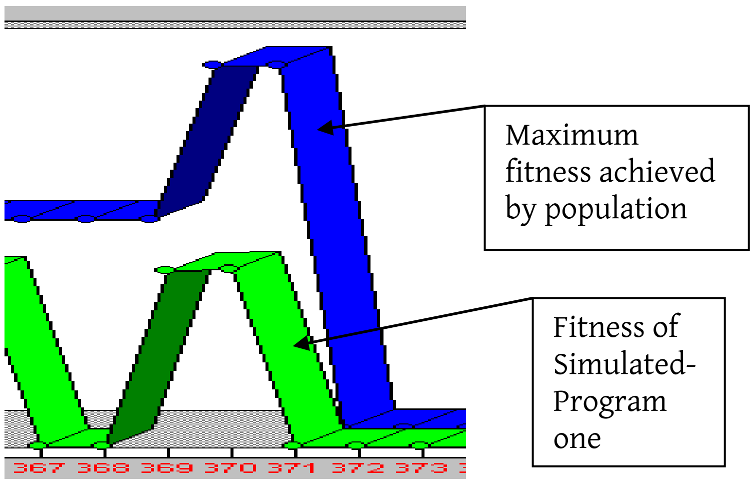


Figure 12.4 Fitness Levels for Best Result

As can be seen, the fitness level has suddenly improved by quite an amount. The value before the improvement must be the value of the fitness level when the AGV had only reached the local maximum. The higher value was reached when the AGV suddenly turned around after reaching the local maximum to find another route to the destination.

12.6 Developments

A software simulator was developed that was able to simulate the Physical AGV. A simulation of most of the AGV's individual parts was needed. The simulated sensors were not only calibrated to the physical sensors, but were also made to give out non-linear results. The instruction set of the simulated robot controller is the same as the instruction set of the PIC. Different virtual mazes were developed for the Simulated AGV. By writing a programme for the Simulated AGV, the whole simulation was proven to function correctly.

Using the simulation and Genetic Algorithms, or Single-Chromosome-Evolution-Algorithms, a programme was generated for the simulated controller. To save time, mutation was changed to favor instructions used when a human programmer wrote a programme for the controller. In future this programme could be changed as desired. For example, if a totally different controller was needed, it could be simulated here.

A Physical AGV was built to prove that the generated programmes could be used on a Physical AGV. The AGV was built as small as possible to make it portable and easy to use. Incorporating the umbilical cord for power purposes instead of batteries makes the AGV acceptable to a variety of future Genetic Algorithm experiments [34].

Although this AGV is too small to carry anything, a replica of the controller could be utilised on a much bigger AGV, making it suitable for use in industry.

A human written programme and a generated programme were individually tested on the Physical AGV. Although not tested within a maze, the AGV reacted more favorably to avoiding obstacles when executing the generated programme than when it was executing the human written programme.

12.7 Findings

If one looks at the learning process of the perceptron through back-propagation, the one question that comes up is how will the perceptron know what the outcome is and, thus, how the change is made if it was, for example, in a human brain? A transistor, or even a gate such as the NAND gate, does not change anything internal and has no memory. Only the information going through it changes. Take the example of the NAND gate. If it was placed in a circuit containing other gates (the same type or different) then it could be made to store a one or a zero, thus making the whole circuit able to remember or learn a small piece of information. It is possible that part of the organic brain functions much like a computer, where the logic gates are replaced by neurons, whose function is comparable to the function of logic gates in a computer.

Although it is possible that the perceptron does not change during the lifetime of the “organism”, it is possible to change through evolution. That is why the perceptrons working in a hybrid system, where the weights are changed through Genetic Algorithms, gave good results [33].

To make an “organism” learn during its lifetime, one can use search trees or schematic trees. A branch in a tree can be like a neuron. The difference between this type of neuron and the perceptron is the direction of the flow of information. The perceptron (and logic gates) have multiple inputs and only one output, while the search tree’s branch, or the whole search tree, has one starting point and multiple endpoints.

In the organic brain, the function of a search tree must be accomplished in either one of two methods:

- A specific type of neuron does the function of a branch in the search tree
- The function of the search tree is built up by neurons just as logic gates can build up the function in a logic circuit

The final goal of any organism is to survive. Any dog-lover will tell you that although the dog’s survivals depend on humans, the hunter’s instinct is still visible when playing, they will even, on occasion catch small prey. While the main goal is survival, hunting is a sub-goal that has become a goal. The sub-goal of the organism’s brain must have been developed by evolution using “survival of the fittest” as the goal. Evolution can also give “goals” or, in other words, instincts that do not directly improve the fitness level but, indirectly, can improve it. An example of this is the human’s desire to think and build.

The final robot brain must have some units, like the perceptrons, in which the placing, connection to others, and the weights of the unit are changed through evolution. A schematic tree must also be included. It is possible that the perceptrons will, in any case, build up a schematic tree. The goals for the schematic tree are determined by evolution. These goals could also be built up by perceptrons.

The schematic tree and the goals can be inserted into the robot in a more direct way where the placement, type, connections and size of the neurons are all determined by evolution. In the end, the robot's brain must be able to learn in one lifetime, as well as change through evolution.

Irrespective of the method used, most of the goals that were set for the learning robot were met. To build a more intelligent robot is to have the robot achieve more difficult tasks. The robot has already been able to walk through a maze, go to a certain location to get recharged, avoid obstacles and predators or to catch prey. The further question and this is the question on which this entire dissertation is based, is "what is the next level of complexity the robot has to contemplate?" The answer might be in the words, "desire to think" or "philosophy".

12.8 Conclusion

Although Single-Chromosome-Evolution-Algorithms gave results much more quickly, they were not able to make radical changes in movement as the result of change to the input, as would be instructed by an input from a user. Including an extra fitness variable for an extra input option made registering an input possible. The problem was that, for each desired reaction, a fitness value would be needed.

Changing one instruction per mutation makes the AGV take a longer amount of time to generate a result, but when analysing the Simulated Programmes it was found to be much more flexible in that almost any instructions sequence was implemented.

Although a simulation of environmental changes influenced the fitness level in a negative way, they did not seem to influence the outcome as much as most of the options did.

12.9 Recommendations

Because of the favourable results generated by the Single-Chromosome-Evolution-Algorithms, one tends to wonder where else it could be used. The reason for the favourable results may be due to the fact that a change to a single instruction can change the function of the Simulated Programme into a totally new function.

It is unlikely that back-propagation will work on neurons with feedback. Similar to self-generated programmes, a circuit with feedback might not use all the parts at any given time, where these currently unused parts can be activated and/or deactivated by changing a small part. In total, Single-Chromosome-Evolution-Algorithms might be an alternative method to make something like neurons with feedback work.

The whole of chapter 3 is basically conveying the following, which would theoretically improve the performance of Genetic Algorithms if implemented:

- By inserting a different chromosome, with similar fitness levels to that of the average fitness of a population, into that population, it will probably give a better fitness level average to the population in the end.
- Because of convergence, a different chromosome might not be easy to obtain.

In section 2.7.5.4, it was seen that, in the sexual preference experiment, the more similar the chromosomes were, the higher the fitness levels given to those chromosomes were. However, in the experiment on diversity, the more dissimilar the chromosomes were, the higher the fitness level that was given to them was.

Although these two experiments seem to be opposites, it is possible that both could be correct. It has been said that opposites attract. That might be true, but both parties must still be from the same species to give good offspring. It all depends on which section of the chromosome has to be similar and which has to be dissimilar. Obviously, determining which part has to be similar and which part has to be dissimilar cannot be determined beforehand. One way to determine this is through a separate use of Genetic Algorithms thus, an additional chromosome will have to be connected, in some way, to each of the original chromosomes.

One method is to generate a programme by using a schematic tree. This method can then refrain from making the same mistake more than once.

13 References

1. McKerrow, P.J. **Introduction to Robotics.** Singapore, Addison-Wesley Publishers Ltd. 1991.
2. Pawson, R. **The Robot Book.** London, Apollo Works, 1985.
3. Marsh P. **Robots.** London, Salamander Books Ltd. 1985.
4. Warwick K. Automatic Guidance. **Ultimate Real Robots,** London, Eaglemoss International Ltd. 2001 vol.7, p.5-8.
5. Andeen G.B. Introduction. **Robot Design Handbook,** New York, McGraw-Hill, 1988. p.1.1.
6. Groover, M.P., Weiss M, Nagel R.N. and Odrey, N.G. **Industrial Robotics. Technology, Programming, and Applications.** Singapore, McGraw-Hill Book Co, 1988.
7. Principe, J.C., Euliano, N.R. and Lefebvre, W.C. **Neural and Adaptive Systems: Fundamentals through Simulations.** New York, John Wiley & Sons, Inc. 2000.
8. Goldberg, D.E. The Existential Pleasures of Genetic Algorithms. **Genetic Algorithms in Engineering and Computer Science,** West Sussex, John Wiley & Sons, 1995. pp24-27.
9. Brooks, R.A. & Mataric, M.J. Real Robots, Real Learning Problems. **Robot Learning.** Boston, Kluwer Academic Publishers, 1993. pp193-210.
10. Craig, J.J. **Introduction to Robotics. Second Edition.** New York, Addison-Wesley Publishing Company, 1989.

11. Luger, G.F. & Stubblefield, W.A. **Artificial Intelligence. Structures and Strategies for Complex Problem Solving. Third Edition.** New Mexico, Addison Wesley Longman, Inc. 1997.
12. McComb. G. **Robot Builder's Bonanza. 99 Inexpensive Robotics Projects.** New York, McGraw-Hill, Inc. 1987.
13. Wise, E. **Applied Robotics.** Indianapolis, Sams Technical Publishing, 1999.
14. Ross, A. **Dynamic Factory Automation. Creating Flexible Systems for Competitive Manufacturing.** Lincoln. United Kingdom, McGraw-Hill, Inc. 1992.
15. Biewald, R. A Neural Network Controller for the Navigation and Obstacle Avoidance of a Mobile robot. **Neural Networks for Robotic Control. Theory and Applications.** Hertfordshire, Ellis Horwood Ltd. 1996. pp.162-165.
16. Filippidis, A., Jain, L.C. and De Silva, C.W. Intelligent Control Techniques. **Intelligent Adaptive Control. Industrial Application.** Boca Raton Florida, CRC Press, 1999. pp.2-21.
17. Nomura, T. Applications of Evolutionary Algorithms to Control and Design. **Intelligent Adaptive Control. Industrial Application.** Boca Raton Florida, CRC Press, 1999. pp.50-51.
18. **PIC16C7X Data Sheet.** MICROCHIP. Atlanta: Microchip Technology Inc. 1997.
19. Warwick K. Attaching the motors. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2001. vol. 2, p.3.
20. Warwick K. ROVS & AUVS. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2001. vol. 4, pp.8-9.

21. ANON. 1983. Hottentotgot. (*In* **Wereldspektrum 11**: p86).

22. Warwick K. Humanoids. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2001. vol. 3, pp 7-9.

23. Ardayfio D.D. **Fundamentals of Robotics.** New York, Marcel Dekker New York Inc. 1987.

24. Hayes, J.P. **Introduction to Digital Logic Design.** Massachusetts, Addison-Wesley Publishing Company, Inc. 1993.

25. Ridley J. **Introduction to Programmable Logic Controllers. The Mitsubishi FX.** New York, John Wesley & Sons Inc. 1997.

26. Logan **Intro to Nervous Nets.** http://www.lmsm.info, (2003)

27. MCNeill D. and Freiberger P. **Fuzzy Logic The revolutionary computer technology that is changing our world.** New York, Simon and Schuster Inc. 1994.

28. Warwick K. Neural Networks. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2001. vol. 2, p 9.

29. Warwick K. An Overview of Neural Networks in Control Applications. **Neural Networks for Robotic Control. Theory and Applications.** Hertfordshire, Ellis Horwood Ltd. 1996. pp. 4-17.

30. Winston, P.H. **Artificial Intelligence. Third Edition.** New York, Addison-Wesley Publishing Company, 1993.

31. Gooratilake S. and Khebbal S. Intelligent Hybrid Systems: Issues, Classification and Future Directions. **Intelligent Hybrid Systems.** West Sussex, John Wiley & Sons Ltd. 1995. pp.1-21.

32. Man, K.F., Tang, K.S., Kwong, S. and Halang, W.A. **Genetic Algorithms for Control and Signal Processing.** London, Springer-Verlag London, 1997.

33. Sushil J.L. and Gan Li. **Combining Robot Control Strategies using Genetic Algorithms with memory.** http://www.cse.unr.edu/, (1997)
34. Floreano D. **Evolutionary Mobile Robotics. Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, Recent Advances and Industrial Applications.** West Sussex, John Wesley and Sons Ltd. 1998. pp. 341 - 365.
35. Rana A.S. & Zalzala A.M.S. Collision-Free Motion Planning of Multiarm Robots using Evolutionary Algorithms. **Proceedings of the Institution of Mechanical Engineers - Part I - Journal of System & Control Engineering.** 25 September 1997. Vol. 211, Issue 5, pp. 373-384.
36. Meyer J, Roitblat H.L. & Wilson S.W. **From animals to animats 2. Proceedings of the Second International Conference on Simulation of Adaptive Behaviour.** Massachusetts, MIT Press, 1993, pp. 21-30.
37. Uffenbeck, J. **The 80x86 Family Design Programming, and Interfacing. third Edition.** New Jersey, Pearson Education, Inc. 2002.
38. Gray J.O. & D. G. Caldwell. **Advanced Robotics & Intelligent Machines.** London, The Institution of Electrical Engineers, 1996.
39. Strum, S.C. **Almost Human.** New York, Penguin Books Ltd. 1987.
40. Wood, B. **The Evolution of Early Man.** Stanmore, Cassell Australia Ltd. 1976.
41. Yves, C.J. **The Adventure of Life.** London, Angus & Robertson Ltd. 1973.
42. Schaller, G.B. **The Mountain Gorilla. Ecology and Behaviour.** Chicago, The University of Chicago Press, 1963.
43. Stephens, W.M. and Stephens, P. **Octopus Lives in the Ocean.** New York, Holiday House, Inc. 1968.

44. Von Ditfurth, H. **Children of the Universe The tale of our existence.** London, William Clowes & Sons, Ltd. 1970.

45. Van Rensburg, S.W.J. **Teelprobleme en kunsmatige inseminasie.** Goodwood Cape South Africa, Nasionale Boekdrukkery Bpk, 1980.

46. Vosloo, W.A. and Casey, N.H. Pig Production System. **Livestock Production Systems Principles and Practice.** Hong Kong, Colorcraft, 1993. pp. 207-209.

47. Sagan C. Cosmos **The story of cosmic evolution, science and civilizations.** London, Futura Publications, 1996.

48. Drago, R.J. **Fundamentals of Gear Design.** Stoneham United States of America, Butterworth Publishers, 1988.

49. Beer, P.B. & Johnston, E.R. Jr. **Mechanics for Engineers. Statics and Dynamics. Fourth Edition.** New York, McGraw-Hill, Inc. 1987.

50. Warwick K. Robot Sensors. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2001. vol. 6, p. 6-9.

51. Barratt M. Micromouse wall follower. **Electronics world.** Vol 110, Somerton England, Highbury Business Communications, June 2004, pp.36-42.

52. Barratt M. Micromouse wall follower part II. **Electronics world.** Vol 111, Somerton England, Highbury Business Communications, July 2004. pp.10-14.

53. Brey, B.B. **The Z80 Microprocessor Hardware, Software, Programming and Interface.** Singapore, Prentice-Hall International, Inc. 1988.

54. Whitley, D. Genetic Algorithms and Neural Networks. **Genetic Algorithms in Engineering and Computer Science.** West Sussex, John Wiley & Sons Ltd. 1995. pp. 205-207.

55. Connell, J.H. & Mahadevan, S. Introduction to Robot Learning. **Robot Learning.** Massachusetts, Kluwer Academic Publishers, 1993. pp. 5-6.

56. Goldberg, D.E. **Genetic Algorithms in Search, Optimization, and Machine Learning.** New York, Addison-Wesley, 1989.

57. Hecht, E. **Optics. Third Edition.** Massachusetts, Addison-Wesley Longman Inc. 1998.

58. Microchip Technology Inc. **Pseudo Random Number Generator.** http://www.Microchip.com, 1997.

59. Warwick, K. AGV's. The Unsung Heroes. **Ultimate Real Robots.** London, Eaglemoss International Ltd. 2003. Vol. 12, pp. 6-7.

60. Scott, P.B. **The Robotic Revolution. The Complete Guide**. Oxford, Blackwell, 1984.