uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Riley August

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Applying Genetic Programming to Scripted Mobile Robotics

TITRE DE LA THÈSE / TITLE OF THESIS

Amy Felty

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Tony White

Jochen Lang

Ahmed Karmouch

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Applying Genetic Programming to Scripted Mobile Robotics

Riley August

Thesis submitted to the
Faculty of Graduate and Post-doctoral Studies
In partial fulfilment of the requirements
for the M.CS degree in Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering (SITE)
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-65496-5
Our file Notre référence
ISBN: 978-0-494-65496-5

Abstract

In this thesis, we develop a new language for genetic programming, specifically designed for high-level controller scripting on mobile robots. We then test this language against previous conventions on the Robots Everywhere Antbot platform. We develop a genetic programming framework using Python and the new language, to create corridor-following programs in a simple simulation. Using this framework, we conduct a variety of experiments to find good parameters and techniques that apply to this new language, which can evolve the best controllers. Our results suggest that hierarchical GP using a measure of elitism is likely the best solution, and that the new language is very capable of evolving solutions with a high degree of robustness and generality.

Contents

List of Figures

Glossary of Acronyms

AI: Artificial Intelligence

API: Application-Programmer Interface

BASIC: Beginner's All-purpose Symbolic Instruction Code

DNA: Deoxyribonucleic Acid, one of the building blocks of cell structure in natural life

EEPROM: Electrically Erasable Programmable Read-Only Memory

GA: Genetic Algorithm

GP: Genetic Programming

GPS: Global Positioning System

GRL: Goal-oriented Requirement Language

KML: File extension used for Google Earth

LISP: LISt Processing language

NAVCOM: NAVigation and COMmunication. Can also refer to the NAVCOM AI, developed by M.K. Borri

NMEA: National Marine Electronics Association; in this text, used to refer to the NMEA 0183 standard

NOSIC: Navigation-Oriented Symbolic Instruction Code

PPM: Pulse-Position Modulation

RAM: Random-Access Memory

SD: Secure Digital

SEBeLS: State- and Event-Based Logic Scripting

STAIR: STanford AI Robot

UML: Unified Modelling Language

1. Introduction

1.1. Motivation

When dealing with mobile robots, the idea of having a single adaptive system that can handle many body configurations is attractive for many reasons. Having a standard that can be quickly referenced and used is obviously a boon, and for users, having only one system to learn makes training much easier. In an ideal world, this kind of system would make flying a supersonic jet plane identical to driving a car or boat, or even a submarine! Such a marvellous system is still a dream of the future, but before something so powerful can exist, one must first learn how such a system could function in the first place. Before even that, one must define what such a system would even be.

An autopilot or navigational AI that strives to be truly universal should at a minimum have the following characteristics:

- The processing and sensory hardware should be designed independently of the mechanical platform, means of locomotion, and steering mechanisms and should be able to operate any such system meeting a basic standard of input and output. This is standard software engineering practice, though difficult to implement in robotics.

- The autopilot software should be able to adapt to the mechanical platform. The internal adaptability of a system is inversely proportional to how much user or engineer input it takes to adapt it – the system should do most, if not all, of the work adapting itself to a new problem. Currently, such as on the NAVCOM AI[48] or ArduPilot[3] humans must do this manually.

- The artificial intelligence(AI) component of the autopilot software should be able to be reprogrammed at least partially in the field, ideally in real-time. This requirement is for field utility, as robots must often do many jobs.

Systems do exist that meet one or more of these requirements, but all of these systems still require considerable user input. The NAVCOM (NAVigation and COMmunication) AI [48], maintained by Robots Everywhere, is such a commercial product, with more capabilities of adapting to different platforms (ground vehicles, boats – both powered and sailed, and aircraft) than others such as the ArduPilot [3] and the Sparrow Autopilot [54], primarily designed for aircraft.

Field-programmability is an especially useful feature because not every operating condition in the field can be foreseen. Conditions such as the wind, surface traction, ocean waves, or simply operating requirements can change on a moment's notice, and having to retrieve a drone in order to make a programming change costs valuable time, especially when all that needs to be modified is a simple parameter. By allowing at least some of the AI's parameters to be modified in the field, a lot of time, as well as training in software modification, can be saved for the robot operators. A fully reprogrammable system exists in the Robots Everywhere Antbot [48], whose layered internal operating system allows the robot to be completely reconfigured for new tasks and even new hardware without

shutting down or aborting its current program to load a new one. Such field-reprogrammability has been used on research vehicles such as NASA's recent Mars rovers, Spirit and Opportunity [5], but until the Antbot, this capability was not available "off the shelf" on a commercial system, as far as we know.

The focus of this thesis is on making use of that field-programmability to improve the overall adaptability of navigation AI, by using evolutionary computation, specifically genetic programming, to evolve new programs for the robot, which can then be executed on demand, in real time.

1.2. Problem Statement

Robust, adaptable robotic systems programmed by humans take considerable amounts of time to optimize for field operation. Genetic programming has been applied to create robotic controllers able to act autonomously in the field. These controllers are written in traditional programming languages, usually LISP, C, or C++.

The purpose of this thesis is to consider the construction and application of a new language, optimized for genetic programming, while still remaining compact and run-time efficient so that it may be used for the scripting of real-time systems. The language will be tested in a dedicated framework based on traditional genetic programming, in a simulated environment on a simple test problem, in this case corridor following. The genetic algorithm's goal is to create a sufficiently general solution that a field deployment in an unknown environment allows the robot to function in a similar manner to its behaviour in the simulator, while it receives no input from the real-world tests.

1.3. Goals

The goal of this thesis is to explain the nature and explore the usefulness of a new scripting language designed for robot AI and autopilot purposes and optimized for genetic programming. The work required to reach this objective is divided into three tasks: the development of the language itself, the development of a genetic programming framework for that language, and finally, the testing of some genetic programming strategies on the final framework.

1.4. Contributions

Several useful contributions have come from our work on this thesis. First, and most prominent, is the language itself, developed specifically for a robotics platform and for genetic programming. It performs better than the existing NOSIC language on both the Antbot and NAVCOM platforms (see 2.6), and provides a simple, no-frills language for genetic programming that uses an inherent hierarchical structure.

The subtree-based crossover operator for the framework is also a significant contribution. It acts in a similar way to existing crossover operators in tree structures, but because of the ordered nature of children in the program tree, it is able to take multiple subtrees from a single parent node and cross all of them over in a linear fashion, just like a linear representation would be able to cross over sequential lines of code.

Testing and benchmark results for the language on the Robots Everywhere Antbot has been recorded, allowing a comparison of the performance of the new language against other languages built for this platform. These benchmarks show that the new language is a slight improvement in both speed and memory usage over the other scripting language for the platform. Additionally, testing results for several different genetic programming strategies using the framework and new language are provided, creating a baseline for further development of GP systems.

1.5. Organization of the Thesis

The thesis is structured as follows: First, a general overview of the requirements and design of the scripting language chosen for the experiments will be presented. Genetic programming will be discussed next, and how its concepts relate best to the development of an adaptable script. Next, the requirements and design of the genetic programming framework will be explained, followed by the experiments conducted in its optimization. Finally a summary of the experiments and their results will be presented, and future work on the topic outlined.

2. An Adaptable Scripting Language

2.1. Languages for Genetic Programming

Dedicated languages for genetic programming have been attempted before, and in many cases have provided an improvement in the specific domain of their function. Most such languages are specific to smaller domains, such as evolutionary neural networks or signal processing. There have also been some attempts at creating general-purpose genetic programming languages, often very simple ones such as Push [55]. In most cases, however, genetic programming operates on existing programming languages, such as LISP, Scheme, or C.

Many GP frameworks, such as OpenBEAGLE [23], Discipulus [47], and GPLAB [53], do not use an existing programming language as part of the representation, but instead use their own. These languages are not designed to run on other systems, but rather to be parsed by the GP framework and run, with the resultant algorithm then translated into a more efficient language such as C++. Most of these languages do not even have a built-in input parser, or even an input syntax – they instead rely on object- or struct-oriented programming to create the program tree. This is an effective method, but usually requires human input at the translation stage. These languages are intended to be general-purpose and able to solve GP problems in many different domains. They, and many others, have some use in evolving algorithms for use in robotic AI, but these algorithms must then be translated into an appropriate embedded language.

The most notably different language among these general-purpose GP languages is Push [55, 56]. Originally designed in 2000, Push has gone through three iterative versions and has become a very mature general-purpose GP language. Push is a very simple language that is defined in text rather than in a graph of objects; it uses a single stack for each type of literal, such as INTEGER and FLOAT, and has many operations that can be performed on single stacks, multiple stacks, or on the program's execution stack itself. Though it appears to have considerable simplicity, Push is a Turing-complete programming language that requires no external command extensions. Because of its simple representation, stack-based nature, and completeness, it makes an excellent language for use with genetic programming. Push programs may be represented linearly, as text strings (possibly delimited by parentheses, white space, or carriage returns), or as traditional GP program trees, which can easily output programs in the Reverse Polish Notation used by Push. It has, however, shown some limitations, for example an inability to evolve list sorting without explicit data structures and new operations being added [21].

More recently, a language called FIFTH [21] has been proposed for genetic programming as somewhat of a general-purpose solution. It is similar to the FORTH language in design and syntax[25] and contains a very optimized, minimalist vector handling capability that improved its work in signal processing. The stack is designed around "containers", each with a type (such as numbers, strings, or

pointers), and a shape, which refers to the number of dimensions and elements per dimension in the data structure. Operators are designed for multi-dimensional vector spaces, making the language ideal for the domain of signal processing. Though it is optimized for a specific domain, any domain that uses vector spaces (such as data mining, or map-based navigation) can make good use of this language.

In 2008, the PlasmidPL language was designed to apply the artificial polymer chemistry [60] method to genetic programming using the behaviour of plasmids in biology. Plasmids are small rings of DNA that occupy a cell of a host organism, acting and reproducing independently, both within and leaving the host cell. PlasmidPL programs are structured as multisets of rings, which can be rings of code, or circular arrays (using modulo indexing). Rings of code can be threads in execution, or again, passive rings of data to be called upon by other rings. These calls are handled by a virtual reactor, which chooses when and where these ring interactions occur, and when each thread is assigned processing resources. Another key element of PlasmidPL is the fact that the genetic operators themselves are rings in the same set of plasmids as the data and code of the evolving program. The language is still very young, but its possible range of applications seems quite wide – it is a new, general-purpose way of expressing programs and genetic operators in a natural, co-evolutionary manner..The use of rings to express functions and threads could, in theory, express states and event handling mechanisms in artificial intelligence programs, as well.

Another related subset of GP languages are systems designed to solve multiple problems, either within one subset of genetic programming or in general. Neural networks are one subset of problems that often use dedicated languages for the evolved programs. Cellular encoding [16] is a method of representing a neural network suitable for genetic programming. An extension of this method, appropriately called Extended Cellular Encoding (ECE) [18], acts as a procedural language for the development of cellular neural networks, and was specifically designed to use genetic programming to evolve neural networks. The applications of this language to robotics are quite clearly apparent, as neural networks are commonly used for robotic AI. An application of cellular encoding to robotics was developed by Gruau in 1996 [17]; this combined human programming with genetic algorithms to evolve an effective algorithm for eight-legged locomotion. Another example of using GP with neural networks is the Swarm-Bots project [36], further discussed in chapter 4.

Though some of these genetic programming languages have considerable potential in their application to robotics problems, and to navigation AI in specific, no domain-specific languages designed to solve the problem have emerged yet. The following sections motivate this approach, and detail the requirements, design, implementation, and testing of such a language, designed specifically for autonomous systems.

2.2. SEBeLS: A Robotics-Specific GP Language

The choice of designing a new language specifically for use by genetic programming, and specifically for autonomous vehicles, was made largely because application-specific GP languages such as FIFTH [21] and ECE [18] have previously shown considerable promise in their own fields. For evolving navigation-oriented artificial intelligence, these languages have no additional benefit over a conventional language such as LISP because the focus in AI is not on manipulating data but on creating a series of responses to stimuli. An explicitly defined state machine, designed in clear, hierarchical tree structure, would be a significant improvement for both human-readability and suitability for AI; it would also allow for clear conditional and loop structures to evolve through state transitions and events, rather than statements within the program structure. The language, named SEBeLS (State and Event Based Logic Scripting), and its requirements are detailed in the next sections.

2.3. Requirements for SEBeLS

In addition to being well suited for use with genetic programming, SEBeLS must still be able to act as a functional scripting language for the user of the platform it runs on. The following requirements document defines the new language to be designed, and its implementation on the Robots Everywhere Antbot [48]:

1 The language's parsing subsystem and all required subcomponents shall exist as a module, or series of methods, capable of inclusion within the existing NAVCOM or Antbot code.

1.1 The module shall not require more than 8 kilobytes additional EEPROM space.

1.2 The module shall not require more than 2 kilobytes additional RAM.

2 The language's parsing subsystem and all required subcomponents shall consume no more than a single core of the Propeller [43] microcontroller.

3 While a program is running, the operation of the console should not be delayed by more than 100 milliseconds, reduced to 20 milliseconds if the program is running on any airborne vehicle.

4 The language shall explicitly define system states, and the actions to be considered within each.

4.1 The language shall explicitly define a global state, and the actions to be considered regardless of system state.

5 The language shall define a state as an identifier, followed by a series of rules.

5.1 Each rule shall consist of a single trigger event and a response to the trigger.

5.1.1 A rule's trigger event may be a null event, and may be always-false or always-true in

that case.

6 The system shall parse each rule in a given state in sequence, then return to the start of that state and repeat.

6.1 The system shall not terminate a program unless an external command, or the program itself, calls for a halt.

6.2 The system shall make all state change commands immediately upon completion of the current rule.

7 The language shall, when possible, use existing commands from the underlying NAVCOM console as its keywords.

The rationale behind some of the requirements may be unclear. Requirements 1 through 3 allow the SEBeLS interface to meet the standards of the Antbot's hardware for memory, storage space, and response time. Requirement 3 was chosen arbitrarily by observing the response time of NAVCOM and NOSIC vehicles, and allowing for a small increase in response time if necessary.

2.4. Design of SEBeLS, the new Antbot Scripting Language

By the time the requirements were finished, ideas for a language were already beginning to form. The end result of this set of requirements is a grammar that stays faithful to the concept of using NAVCOM commands, and exists as an explicitly state-based and event-driven language:

```
RULESET → [STATE]+
STATE → STATEDEF[\n][RULE CRLF]+
CRLF → \n
STATEDEF → [S|s][T|t][A|a] NUMBER
NUMBER → [0-9]+
RULE → [COMMAND;]?COMMAND[,COMMAND]*
```

COMMAND refers to any console command, defined in the manual for the console type being used. They do not need to be explicitly defined in the grammar, as the language parser will access their command parser directly. For a list of commands for each specific platform (such as the Antbot and the NAVCOM) see the appropriate datasheet, on the Robots Everywhere website. This flexibility allows any implementation with a similarly-built command parser to the Antbot and NAVCOM to use the language without significant changes (notably, NMEA uses a similar syntax for actuator devices).

Within a RULE, the first command is the condition command (followed by the semicolon). The condition command can be any command, though on the Antbot only @E and @? (math expression parser without and with result output on the terminal respectively) will return integer values – all others commands simply return 1.0 if they execute successfully, and 0.0 otherwise. This behaviour is dependent on the external command set used. This first command is optional, and if not present, the

rule will always execute. Commas then separate the commands to be executed; if the conditional command returns a non-zero value, the other commands on that line are executed. At the end of each state, the system will go to the next state if it is set, otherwise the current state will be repeated.

With the grammar now defined, it must be implemented and tested. The implementation is done in Spin, the Python-like language for the Parallax Propeller microcontroller, and added onto the console for the Antbot. The architecture of the Antbot itself is very receptive to such a change, and at the time of writing, the only necessary change is a replacement of the central console module (built around NOSIC) to one built around SEBeLS. The current Antbot architecture is shown below:

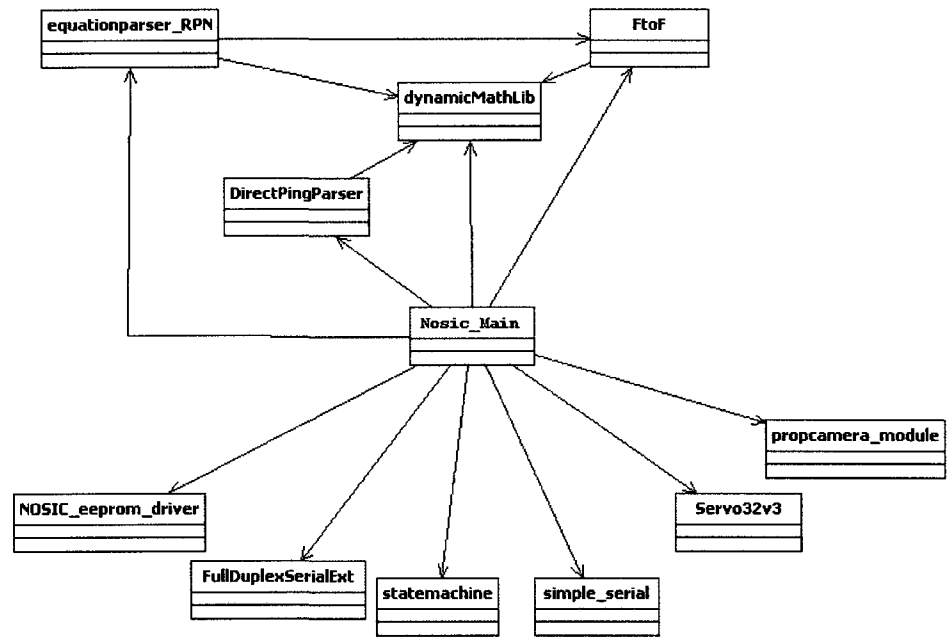

Figure 1: The architecure of a NOSIC Antbot.

As the above UML class diagram shows, the software is designed in a hub architecture, with different modules being called by a central class. The names of the modules are mostly self-explanatory – there are two serial communication modules, an EEPROM driver, a state handler, a camera module, a servo driver, a math library, and an expression parser (which uses the math library to return the result of a RPN expression). The DirectPingParser refers to the Parallax Ping sonar rangefinder (the current NOSIC implementation offers this and a NMEA sensor parser as options), and is used for sonar data. FtoF is "float to formatted number" and is used to convert string numbers into floating-point values,

and back.

The camera parser on the Antbot, in addition to performing basic image processing such as finding the brightest and darkest spot in view, uses ASCII characters to represent light intensity and display a visible image; this was originally done to allow any terminal program to work with an Antbot out of the box. An example screenshot is below:

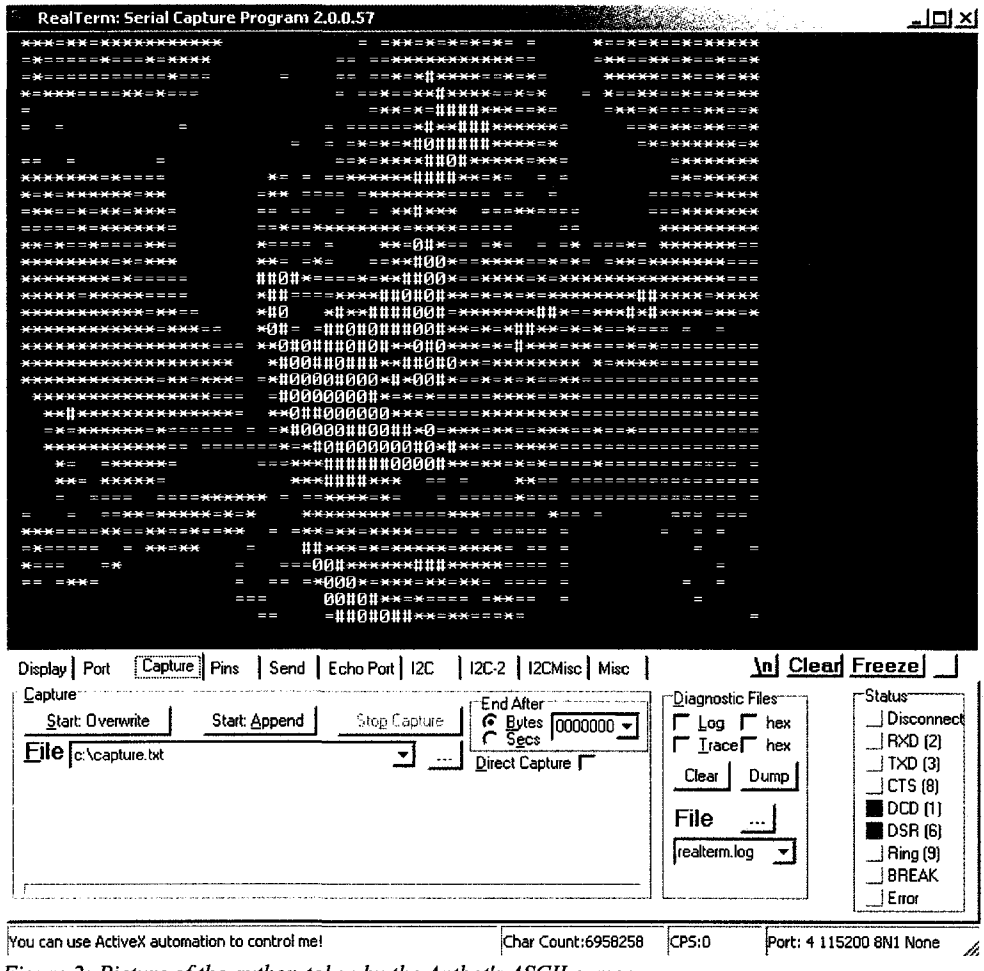

Figure 2: Picture of the author, taken by the Antbot's ASCII camera.

The process for changing this system to use SEBeLS is simple. It requires changing only a single class, as shown here:

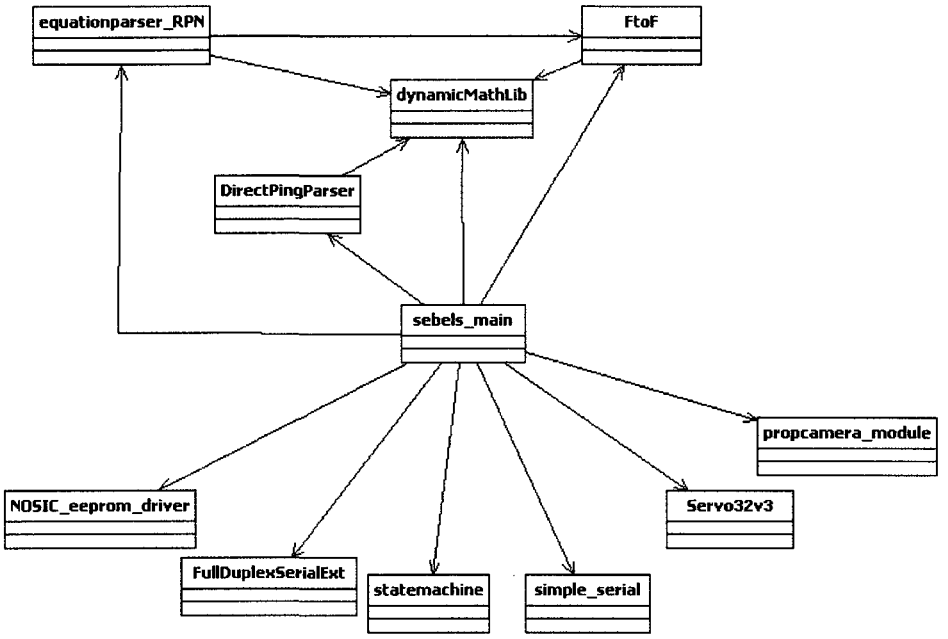

Figure 3: A UML Class Diagram for an SEBeLS-enabled Antbot

The implementation of this class diagram requires the replacement of only a few console methods within the main hub. Source code for the SEBeLS implementation of the Antbot can be found on the Robots Everywhere website. Some example SEBeLS programs can be found in the next section, where they are used for testing the language's ability to perform various tasks.

2.5. SEBeLS IN ACTION

Testing of the SEBeLS system was done through the execution of several programs and the comparison of the results with existing NOSIC code. In the following tests, an attempt to manually program a SEBeLS Antbot to perform various tasks that a NOSIC Antbot is known to be able to perform is undertaken; the tests cover the ability to manipulate the robot's motor and camera systems, and read data from the camera and sonar sensors on the Antbot.

NOSIC and NAVCOM commands all begin with an @, followed by a token, followed by the token's parameters; this is analogous to the $-identifier-parameters structure of a standard NMEA string. For example, the command @?2 1 + adds 1 to 2 and displays the result. The Antbot commands used are:

```
      @"    print the characters that follow it, treating " as 'end command and
print carriage return'
      @?    print the result of an expression and return the result
      @E    as @?, but without term output (just return the result to the parser)
      @P    print the current camera frame to the terminal
      @J    move the Antbot in a given direction, allowing a numeric keypad to
            function as a joystick: 1 - backward and counter-clockwise,
            2 - backward, 3 - backward          and clockwise,
            4 - counter-clockwise, 5 - stop, 6 - clockwise,
            7 - forward and counter-clockwise, 8 - forward, 9 - forward and
            clockwise.
      @F    set a scratchpad variable (A-Z) with the result of an expression;
            for example @FA 20 10 + sets A to 30.
      @T    print telemetry on sonar, battery levels and execution times
      @S    set writeable system variable (such as motor drive strengths)
      @Q    stop parser and end the main implied program loop
      @N    change state machine state to the following value (may use an
            expression). Example @N 4 sets the machine to state 4. White space is
            optional. This command is state machine specific: @N in NOSIC is used
            as a gosub instruction. In the NAVCOM this is also used for the state
            machine.
```

Further details on these commands can be found in the in the Antbot manual soon to be available on the Robots Everywhere website [43]. This includes the list of mathematical operators available on the Antbot platform. Additionally, comments in the code will be represented by italic text.

In addition, some properties of the Antbot must be explained. The Antbot's basic configuration possesses four sonar rangefinders, whose outputs are continually relayed to the read-only (lower-case) values g, h, i, and j in the Antbot firmware. The Antbot also has a camera, as described in 2.4. In addition to the image, it stores the x and y coordinates of the brightest point in the lower-case values n and o, respectively. Finally, the Antbot has an emergency stop setting, stored in the system value E, which defaults to 30 arbitrary rangefinder units – approximately 30cm for the Parallax Ping rangefinders in use. For all of these tests, it has been changed to 10, due to the small physical area available for testing. The rangefinders are mounted so that the left and right beam cross to avoid the frontal blind spot issue common to these designs, and activated by the DirectPingParser class with such timing that they do not interfere with each other.

In NOSIC, unlike SEBeLS, only one command can be used per line. The @ sign is placed before the line number, instead of before the command identifier, and there are no explicit state definitions. The NOSIC code corresponding to each test program is added for familiarity, as it is the standard for Antbot scripting. The I command in NOSIC is used to indicate an If statement, with an @ proceeding to indicate the result (the Then statement).

2.5.1. Test 1: Sonar Telemetry

This test is simply designed to ensure the sonar rangefinders return telemetry properly when accessed by SEBeLS.

Precondition: Sonar sensors attached to Antbot; if fewer than four, the empty sonar mounts will return a marker value (0 or 999).

Variables: The variables g, h, i, and j hold the sonar data taken in by the Antbot's rangefinders.

Code:

```
STA 0
@"LF:,@?g,@" RF:,@?h,@" LB:,@?i,@" RB:,@?j      Print text for each variable g, h, i, j
@Q
```

Corresponding NOSIC code:

```
@1 "LF:                          Print text "LF:"
@2 ? g                           Print variable g
@3 "  RF:
@4 ? h
@5 "  LB:
@6 ? i
@7 "  RB:
@8? j
@9 Q
```

Expected Result: This should print a string along the lines of LF:### RF:### LB:### RB:###, repeating as long as the stop flag is not set.

Where ### are the sonar range values for those sensors.

Incidentally, the @T command outputs a similar string; this code is intended to show how to access single sonar results individually and should probably be replaced by a @T instruction for actual field use.

Actual Result:

```
LF:.    316.698 RF:.     20.011 LB:.     39.850 RB:.     83.798
```

2.5.2. Test 2: Camera Image

This test is simply designed to ensure printing camera frames is done properly within a SEBeLS run.

Precondition: Camera attached to Antbot.

Code:

```
STA 0
@P          print camera frame
```

Corresponding NOSIC code:

```
@1 P
```

Expected Result: The ASCII camera frame should be printed repeatedly. Note that in both SEBeLS and NOSIC, the program is assumed to loop and must be stopped explicitly.

Actual Result: As expected.

2.5.3. Test 3: Motor Movement

This test is simply designed to ensure the program can control the motors.

Precondition: Antbot motors functional and connected to motherboard.
Code:

```
STA 0
@J8         drive forward
@N1         go to empty state 1
```

Corresponding NOSIC code:

```
@1 J8
@2 N 1
```

Expected result: The robot will move forward until it encounters an obstacle, then stop.

Result: As Expected.

2.5.4. Test 4: Sonar with Motors (Wall-Follower)

This is the first performance test of the SEBeLS equipped Antbot. It will test basic indoors navigation by having the Antbot use its sonar rangefinders to find the centre of a corridor, and drive forward along it By treating it as a short and wide corridor, the program will be able to navigate a convex room using this method. No obstacle avoidance is included in this code since adding the extra state needed for that is trivial.

Precondition: Antbot motors and front rangefinders functional and connected to motherboard.
Variable B is set to 20.
Variable C is set to 20.
Variable A is used as a work variable.

Code:

```
STA 0
@T                output telemetry
@J8               drive forward
@FA g h -         set A equal to g-h
@EA C >;@N1 if A > C, goto state 1
@EA C! <;@N2      if A < -C, goto state 2
@Eg B <;@N2 if g < B, goto state 2
@Eh B <;@N1 if h < B, goto state 1
STA 1
@T                output telemetry
@J7               go forward while turning counter-clockwise
@FA g h -         set A equal to g-h
@EA C <;@N0 if A < C, goto state 0
@EA C! <;@N2      if A < -C, goto state 2
@Eg B <;@N2 if g < B, goto state 2
STA 2
@T                output telemetry
@J9               go forward while turning clockwise
@FA g h -         set A equal to g-h
@EA C >;@N1 if A > C, goto state 1
@EA C! >;@N0      if A > -C, goto state 0
@Eh B <;@N1 if h < B, goto state 1
```

Corresponding NOSIC code:

```
@1 T
@2 J8
@3 FA g h -
@4 I A C > @G10
@5 I A C ! < @G20
@6 I g B < @G20
@7 I h B < @G10
@8 G 63
@10 T
@11 J7
@12 FA g h -
@13 I A C < @G1
@14 I A C ! < @G20
@15 I g b < @G20
@16 G 63
@20 T
@21 J9
@22 FA g h -
@23 EA C > @G10
@24 EA C ! > @G1
@25 Eh B < @G10
@26 G 63
@63
```

Expected Result: The robot should drive around the room, staying centred between the left and right walls. Should it find a blind spot or overcompensate, it would drive close enough to a wall for the firmware to trigger an emergency stop.

Result: As expected.

2.5.5. Test 5: Camera Targeting (Light-Follower)

This test will show SEBeLS performance with the Antbot's on-board camera. It will test basic targeting, and the following of a moving point. This simple algorithm can be combined with other navigation programs to do operations such as pursuit if used with other types of directional sensors, or short-range targeting and motion detection as it is.

Precondition: Antbot and motors and camera functional and connected to motherboard. A flashlight with brightness significantly contrasting the lighting of the room should be available.

Variables: Variable A controls how quickly the Antbot will move forward or back, depending on the distance the light is from the camera. Variable B controls the maximum speed of the Antbot.

Variable C controls how quickly the Antbot will turn based on the position of the light to the left or right of the camera. Variable D controls the reaction time of the Antbot's turning.
Variables X and Y are set by the program, and are the input registers for the Antbot's motor controller. This is similar to using @J commands, but more precise.

The value n is the x-coordinate of the brightest point the camera can locate; the value o is the y-coordinate of the same point, treating the camera's image as a Cartesian plane.

Code:

```
STA 0
@SX n 30 - A * B }      set left-right turn axis equal to brightest (n – 30 * A), clamped to B
@Eo 15 <;@SY o C - D *  if o < 15, drive forward or reverse based on value of o
@Eo 15 >;@SY 0          if o > 15, stop
```

Corresponding NOSIC code:

```
@1 SX n 30 - A * B }
@2 Io 15 < @SY o C - D *      if o < 15, drive forward or reverse based on o
@3 Io 15 > @SY 0              if o > 15, stop
```

Note that the command @SX and @SY set non-scratch variables, in this case the motor drive strengths. This is not to be confused with @FX or @FY, which would alter scratch variables. For example, @SY 1.0;@SX 0.0 sets the motors to full forward and @SY 1.0; @SX 1.0 sets the motors so that an arcing turn is made (as per @J7), while @SY 0.0;SX 1.0 causes a spin turn as per @J4.

Procedure: Aim a flashlight where you want the robot to go, within the field of view of the camera.

Expected Result: The robot should drive towards the light, turning to keep it roughly centred in the camera view. If the light is too close to the camera, it will reverse to maintain a reasonable distance.

Result: As expected.

2.6. Conclusions on SEBeLS

After testing SEBeLS for a variety of tasks, it proves itself to be compact and effective at replacing NOSIC for scripting on the Antbot. NOSIC's main advantage of usability, due to its similarity to BASIC, is offset by the smaller memory footprint of SEBeLS programs (6152 bytes vs NOSIC's 7988), as well as a small increase in execution speed (6347664 clock cycles vs 6372784 clock cycles for a simple wall-following program). Additionally, due to its explicit definitions of states and events, it is very well suited to automatic code generation, both through Computer-Aided Software Engineering (CASE) tools, and through fully automated techniques such as genetic programming, described in the next section.

3. Genetic Programming: Background and Concepts

3.1. Introduction to Genetic Algorithms

Genetic Programming (GP) is a relatively old concept in the field of computing [30]. Based on the concepts of genetic algorithms[22] and the iterative nature of software development, genetic programming is a technique for solving a problem that uses a computer program as a representation of a possible solution. Each program acts as an individual, and a population of such individuals is allowed to evolve through the use of various genetic operations in a simulation of natural evolution.

At each generation, the most fit individuals (according to a fitness function that outputs a numerical representation of how fit an individual is, that is, how valid a solution it represents) reproduce to create new individuals. During reproduction, genetic operations such as mutation and genetic crossover occur similarly to how they occur in nature. The offspring of this reproduction are then put into a new population, which forms the next generation; generally all of the previous population's members are then removed, keeping the population at a constant size for ease of computability. The basic genetic algorithm is as shown:

```
1. GeneticAlgorithm(int size, int m, int c, expression stop, selectionmethod
      selection, crossovermethod crossover, mutationmethod mutation,
      fitnessfunction fitness)
2.    // generate the initial population randomly
3.    population = []
4.    while q < size:
5.          population.add(new individual("random"))
6.          q++
7.    print max(population by fitness.getFitness()) // output best individual
8.    while not stop:
9.          newpopulation = []
10.         while q < size:// select individuals for reproduction
11.               [a,b] = selection.select(population, 2)
12.               if random.randomnumber(0,1) < c:
13.                     a = crossover.cross(a,b)
14.               if random.randomnumber(0,1) < m:
15.                     a = mutation.mutate(a)
16.               newpopulation.add(a)
17.               q++
18.         population = newpopulation
19.         print max(population by fitness.getFitness()) // output best individual
```

Algorithm 1: The basic genetic algorithm.

The ultimate goal of any genetic algorithm is continuous improvement of the population until an adequate solution to the problem is reached. Unlike conventional methods, a GA is not automatically going to reach the optimal solution to a problem; it is possible for a GA to run indefinitely and such a perfect solution never to be found. Instead, it will creep closer and closer to the optimal solution, until one "good enough" is found, based on a chosen stopping point. This stopping point may be based on an arbitrary average or maximum fitness number for a generation, a certain number of generations with no change in the best individual, or a fixed number of generations overall. Different experiments will halt on different conditions based on their specific requirements.

This said, there is a practical halting state known as convergence. When a genetic algorithm is said to have converged, its population has become stable – it is more homogeneous than it was at its starting point, and has reached a local maximum in the solution space that it cannot escape from. There is not enough variability in the population (measurable by the number of unique individuals) to allow mutation and crossover to take the algorithm over to a new "hill". Effectively, the process of convergence is the end result of the algorithm finding a peak in the solution space, hill-climbing[51] to a point near the top, and stopping. Avoiding premature convergence, in which a low-fitness local maximum is reached and got stuck in, is one of the main goals in the development of new GA techniques; the focus is generally on methods of ensuring variability in the population by preventing identical individuals (clones) from dominating the population.

To use a genetic algorithm in a useful way, there are a number of design decisions that must be made. First, one must decide how to represent the problem and design the "genome" used by the genetic operators. Next, the selection method(s) and genetic operators (such as mutation and crossover) must be designed. Finally, though far from least important, the fitness function that determines the fitness values used in selection must be designed. The process will vary slightly for different problems or specifications of a genetic algorithm (such as genetic programming) but the core concept remains.

3.2 Representing the Problem

The key to making good use of a genetic algorithm is choosing the proper representation for a solution to the problem. For example, if attempting to solve the Travelling Salesman Problem (TSP), a properly usable representation of the problem would be as a sequence of nodes, referenced by number and listed in the order in which they are visited; a poorer representation would be a linked list, which would be significantly less time-efficient for genetic operations. An unordered set, for example, would not work at all because of the ordered nature of the result for TSP.

In GP, a tree structure is a standard representation for programs in most languages. In the tree, the root node represents the entire program – subtrees represent statements and code structures like subroutines. Each leaf contains an atomic element of a statement; generally a parameter, primitive data type, or variable. The example below shows how to represent a simple program using a tree structure.

The following program, defined in LISP (one of the original languages used for GP) prints out "hello", "yo", and "whassup?" on separate lines, followed by the number 9 twice.

```
(if (> 3 2)
          (progn (print "hello") (print "yo")
            (print "whassup?") (print 9)
          (+ 4 2 3)))
```

For those familiar with LISP it is a very simple program; here it is described as a tree:

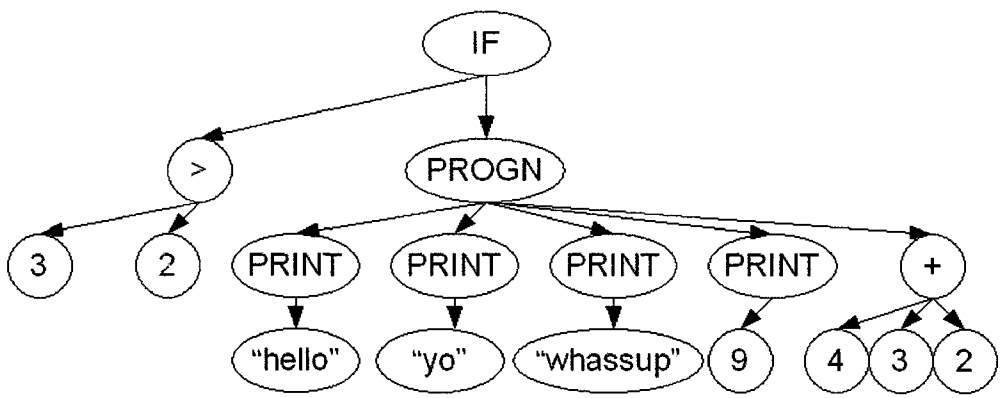

Illustration 1: An example LISP program tree

The same program can be written in SEBeLS:

```
STA 0
@?3 2 >;@"hello",@"yo",@"wassup",@?9,@?4 2 + 3 +
@Q
```

The SEBeLS program tree for the example follows:

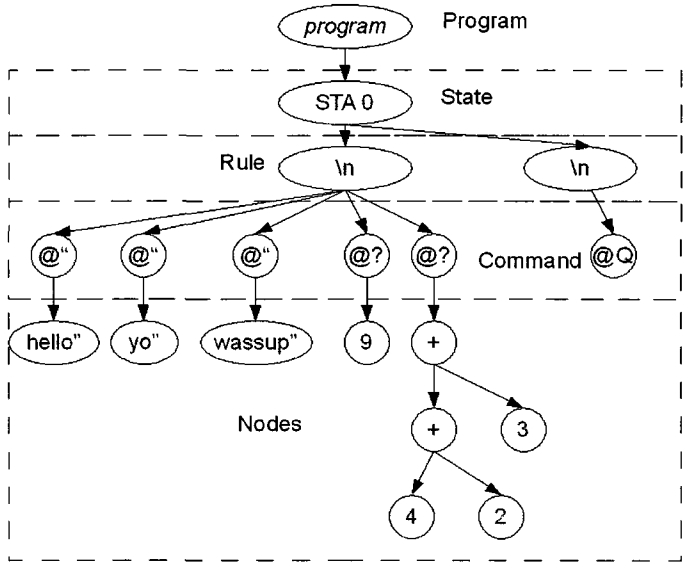

Figure 4: An example SEBeLS program tree, with levels defined

3.3. Reproduction Techniques – Selection

Another important choice that must be made when preparing any evolutionary system is the choice of which individuals will reproduce. The function that, based on the fitnesses of the individuals in the population, determines which individuals will reproduce during that generation is referred to as the selection method. There are many different selection schemes in the field of genetic algorithms; those considered in this project were Fitness-Proportional Selection, Rank Selection, Tournament Selection, and Hi-Lo Fit[1], a recent selection scheme developed in 2006 by Elnima (see section 3.3.4). The majority of related work found in Chapter 4 uses Tournament Selection or a variant thereof.

3.3.1. Fitness-Proportional Selection

Fitness-Proportional selection is a very random means of selecting an individual. Effectively it makes a "roulette wheel" with slices for each individual proportional to that individual's fitness. After determining the fitness values of every individual in the population, it generates a random number between 0 and the sum of all the fitnesses of all individuals in the population. This random number corresponds to a position in an array that the individuals occupy in a way that the number of positions in the array taken by one individual is equal to its fitness value; this ensures that the probability of an individual being selected is equal to its fitness over the sum of all fitnesses for that generation. Some

variants also choose to drop some subset of individuals with low fitness in order to save on processing time.

The selection algorithm is as follows:

```
1. for q in population:
2.     fitness[q] = getFitness(q)
3.     totalfit = totalfit + fitness[q]

4. randomnumber = random(0, totalfit)
5. temp = 0
6. for q in population:
7.     if randomnumber < fitness[q]+temp:
8.           return q
9.     else:
10.          temp = temp + fitness[q]
11.          next q
```

Algorithm 2: Fitness-proportional Selection

The biggest advantage to this selection method is that there is no need to sort the population by fitness before starting – the order of q is irrelevant. However, considering the increased randomness of selection and inability to vary selection pressure caused by this method, there is little reason to use it over more modern techniques.

3.3.2. Rank Selection

Rank Selection is another very simple method of selecting an individual. It simply sorts the population by fitness, starting with the highest, then iterates from most fit to least fit continually checking against a fixed non-zero probability p. Therefore the probability of the nth most fit element in the population being selected is simply p^n. This method has an advantage over the previous one because by varying p, selection pressure can be changed. The algorithm follows:

```
1. sort q in population by fitness(high to low)
2. for q in population:
3.     if random(0,1) < p:
4.           return q
5.     else:
6.           next q
```

Algorithm 3: Rank Selection

The advantage of rank selection is that it has the ability to vary selection pressure, by varying p. This allows it to have greater selection pressure than fitness-proportional selection, while being only slightly more computationally intensive. Using known optimizing techniques, the sorting becomes less of a problem, and rank selection can be a somewhat effective selection method.

3.3.3. Tournament Selection

Tournament selection [34] is a more complex, but much more effective means of selection. It is much more flexible in varying selection pressure than rank selection. The focus is the completely random selection of two individuals from the population, which are then "competed" against each other (their fitnesses compared). The most fit one is usually selected, with some small probability of choosing the other. The pseudocode follows:

```
1. indi1 = random(q in population)
2. indi2 = random(p in population)
3. if random(0,1) < p:
4.     return max(indi1, indi2, key=fitness)
5. else:
6.     return min(indi1, indi2, key=fitness)
```

Algorithm 4: Tournament Selection

When multiple individuals share the same fitness value, the max() function will choose one based on its implementation – different programming languages may select different individuals, but for the purposes of the raw algorithm, any pseudorandom or arbitrary selection method is sufficient, as long as it is known. Python, for example, simply selects the first argument.

There is also a variant known as k-tournament selection, where the tournaments are of size k rather than of size 2. This gives another means of controlling selection pressure; higher values of k become less random. With k-tournament selection, the algorithm looks like this:

```
1. for a=1 to k:
2.     indi[a] = random(q in population)
3. if random(0,1) < p:
4.     return max(fitness(q in indi))
5. else:
6.     remove(max(fitness(q in indi) from indi))
7.     return random(p in indi)
```

Algorithm 5: k-Tournament Selection

Again, it will usually return the winner of the tournament, with some chance that a completely random individual from the tournament will reproduce instead. This is intended to add some variability to the population and allow evolution to progress. Also, if two or more individuals in the tournament have the same highest fitness within the tournament, some arbitrary or pseudorandom selection method will be employed. Python, for example, selects the first argument in the list.

The greatest advantage to tournament selection is that selection pressure can be made as high or as low as we want. By varying the tournament size, k, the variability of selection will change; by varying p, the probability of selecting the winner of the tournament, we can change how much high-fitness individuals will be exploited, versus the reproduction of random individuals. With two variables, a much greater range of possibilities is opened up. A small additional advantage is that if a random individual is selected after computing p, no fitnesses have to be calculated, thus decreasing overall

computation time.

3.3.4. Hi-Lo Fit

Hi-Lo Fit [1] is a selection method that at first appears counter-intuitive, but has both been shown to be quite effective and fairly well represented in nature. It simulates a reproduction method in certain species whereby one dominant individual forces a weaker individual to mate, rather than simulating a fitness-based courtship process where like attracts like. This tends to result in one very fit individual and one less fit individual reproducing more often than two fit or two unfit ones. The scheme removes the concept of gender (though it can be used in genetic algorithms, it is far beyond the depth of this experiment) and abstracts the selection scheme into a very simple one: select one individual from the upper half of the fitness curve and one from the lower half, and mate them.

Unlike the other algorithms, this one cannot be used to effectively return a single individual: it only returns breeding pairs. In most cases this is sufficient; if select is called for fewer than two individuals, another selection method is used instead.

The algorithm follows:

```
1. sort q in population by fitness(high to low)
2. midpoint = length(population)/2
3. high = population[random(0, midpoint)]
4. low = population[random(midpoint, length(population))]
5. return [high, low]
```

Algorithm 6: Hi-Lo Fit

The advantage of this algorithm is in adding additional randomness to the process while still exploiting high fitness individuals considerably. By ensuring that an individual in the high-fitness half of the population breeds at every selection step, it is almost guaranteed through weight of numbers that the best individuals' genes will be passed forward. Additionally, by choosing a low-fitness individual to mate with the better one, randomness is introduced in the form of "bad" code segments which may become useful in a different context.

The function was tested by Elnima on a variety of math functions [1], though not tested further. The ability to create high variability in a population while retaining the highest-fit individuals shows potential for genetic programming, hence its inclusion in this thesis.

3.4. Genetic Operations

After a breeding pair or group has been selected (usually by running the selection method n times, or n/2 times in the case of hi-lo fit), the next step is obtaining the offspring to insert into the new population. There are two basic genetic operators that are used on a mating pair: mutation and crossover. Each occurs by a random probability, m for mutation, and c for crossover. One, both, or neither of mutation and crossover could occur in a single instance of reproduction. When neither occur, the result is simply the first of the selected breeding pair being cloned into the new generation,

retaining all of its genetic material.

3.4.1. Crossover

In nature, genetic crossover occurs when one strand of DNA physically breaks and the strand from the other parent reattaches when it repairs, rather than the original. In genetic algorithms, this operation is mimicked by a simple crossover function that uses the following algorithm, assuming the genome is an array or similar indexed list:

```
1. function crossover(self, other):
2.     start1 = random(0, length(self))
3.     start2 = random(0, length(other))
4.     length = random(0, min(length(self)-start1, length(other)-start2))
5.     temp = self[start1..start1+length]
6.     self[start1..start1+length] = other[start2..start2+length]
7.     other[start2..start2+length] = temp
8.     return self, other
```

Algorithm 7: Linear Crossover

In most cases of crossover only the first return value is used, and the second parameter is a copy of an individual in the population. The diagram explains crossover somewhat more elegantly:

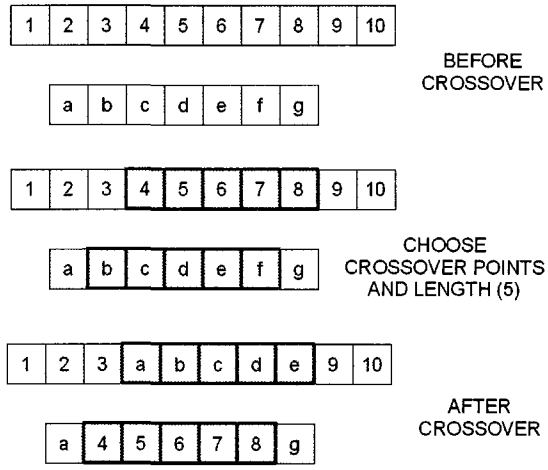

Figure 5: Examples of crossover using an array representation. Red boxes denote changes.

In the case of the tree representation used in genetic programming, crossover occurs on subtrees. This uses a different algorithm, below:

```
1. function crossover(self, other):
2.     node1 = random(0, length(nodes in self))
3.     node2 = random(0, length(nodes in other))
4.     temp = node1.parent
Remove the nodes from their parents' lists
5.     node1.parent.removechild(node1)
6.     node2.parent.removechild(node2)
Add the nodes to their new parents' lists
7.     node1.parent.addchild(node2)
8.     node2.parent.addchild(node1)
6.     node1.setparent(node2.parent)
6.     node2.setparent(temp)
8.     return self, other
```

Algorithm 8: Subtree Crossover

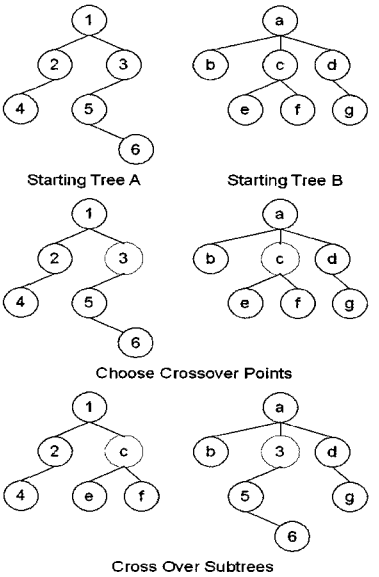

Figure 6: Crossover on a tree-based representation.

The diagram at right shows an example of this type of crossover:

There can be restrictions on this type of crossover, generally caused by the specific representation of the problem set or program tree. In SEBeLS, crossover can only occur within data types, which are represented by specific levels in the tree. A state cannot be crossed over into a command, which is a lower-level structure, because this would violate the syntax of the language. Multi-level commands, however, are able to cross over into each other as normal for a tree structure – in the implementation, each node has a type identifier describing itself and its subtree. For more information on SEBeLS crossover, see chapter 5.

This is hardly the only crossover algorithm in existence. Many others such as single-point crossover (where one subtree, or the tail of a list is crossed over), "cut and splice" (a variation of single-point crossover where the point can be different on each individual), and Uniform Crossover (where individual nodes are crossed over) [19] exist. These can be further explored in the reference text and will not be covered, nor used, here.

When crossing over components that can have different types (such as atoms in a sentence, or statements in a computer program), there is often the need for strict type constraints during crossover. This is especially true when some nodes in a representation are parameters for another node – such as a function call and its parameters before or after the function definition. The concept of strongly typed genetic programming (STGP) was developed in 1995 by Montana [37], and restricts the search space of genetic programming by eliminating invalid combinations of data types, notably within function parameters. These restrictions can greatly increase the power of genetic programming by greatly cutting down on the number of low-fitness individuals that can be generated, but reduce the possibility of positive mutations by pruning the search space.

3.4.2. Mutation

Mutation is a primary driving force of genetic variance in nature. Generally random in its effect and its breadth, it can transform an individual very slightly, or very drastically. In genetic algorithms, the algorithm for mutating a genome is entirely dependent on its representation. For a typical array representation, there can be several forms of mutation: insertion, deletion, swap, or modification. Insertion mutation results in a new (randomly generated) node being inserted into the genome at a random point. Deletion results in the opposite occurring: a random node in the array being destroyed. A swap mutation involves two nodes' positions within the same individual being exchanged; obviously, this is only relevant if position is important in the given representation. This operator should not be confused with crossover, which swaps one or more elements in sequence from different individuals in the population. Modification, usually the most complex form of mutation, involves a single node in the array being transformed in some way. An integer, for example, could be incremented, decremented, or changed to a random value. A command could have its parameters changed in similar ways, or the command itself rewritten to be a new one. The above diagrams demonstrate some examples of mutation in a simple genome represented by an array of integers.

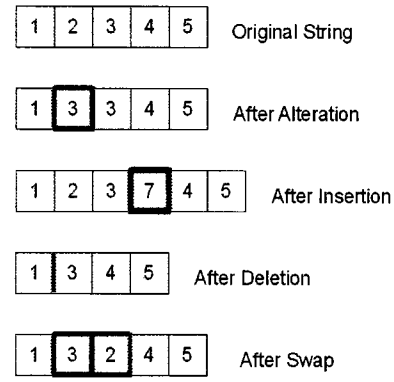

Figure 7: Examples of mutation in an array representation. Red boxes indicate changes.

In a tree representation, similar mutation operations are possible, but they must consider the hierarchical nature of a tree. A deletion operation at the root of the tree, for example, would destroy absolutely everything, while insertions below the leaves of the tree would do nothing at all without first modifying the leaf nodes. A few simple examples using trees of integers are shown at left.

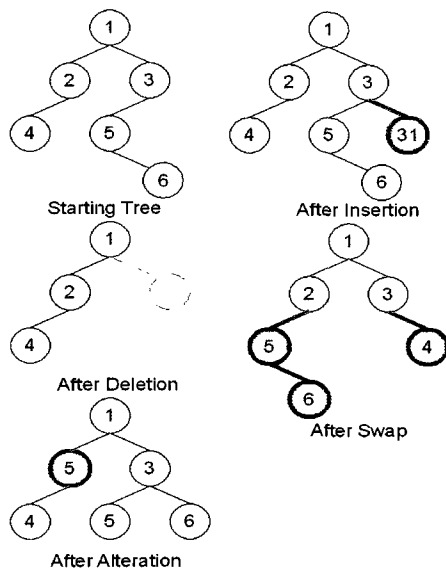

Figure 8: Examples of mutation in a tree representation. Red circles and lines denote changes.

3.5 Fitness Functions

The choice of fitness function is one of the most important parts of designing a genetic algorithm. If there is no proper way of comparing individuals against the problem being solved, the solutions generated will be wildly inaccurate. An individual's fitness needs to be measured as closely against the problem in question as possible in order for any selection method to be effective at all.

3.5.1. Designing a Fitness Function

The problem of choosing the function is generally of similar complexity to the representation of the problem being solved. Problems with easily-defined fitness functions, such as the Travelling Salesman Problem, whose solutions can be measured simply by the length of the tour (with shorter tours being better), can be solved with simple fitness functions. Other problems, such as the design of a more optimal antenna for a certain frequency, needs a more complex fitness function. The most complex fitness functions arise when a solution needs to be general, solving more than one problem at once; in general, a fitness function will have complexity proportional to the difficulty of representing the problem to be solved.

In the case of a solution for a real-world problem, the fitness function must employ some measure of simulation. Simulation of an environment and agents within it is generally complex and computationally expensive, so consideration must be given to simplifying the simulation to a point where it can be run as fast as possible while keeping its accuracy sufficient to generate a solution that will be valid the real world. This type of optimization is generally done through an iterative process of development and testing until a satisfactory design is achieved. Once the simulation is completed and tested for accuracy, some property or combination of properties from the simulation is used as the raw fitness measure.

3.5.2. Raw and Relative Fitness

Relative fitness is a way of expressing the quality of individuals within a population without resorting to the numerical output of the fitness function, known as the raw fitness. Often measured as a percentage of some maximum fitness, the relative fitness provides a clear and obvious measure of the effectiveness of an evolved solution. While it adds some computational overhead, it often makes comparing different representations of the same problem much easier. It also allows modification of the absolute fitness of individuals near the extremes of the possible fitness values, so that selection methods such as fitness-proportional selection are able to work more effectively. This is useful because in cases where large fitness values are very close together, the percentage difference in the raw fitness can be less than a thousandth of a percent. By using relative fitness and exaggerating this difference, such selection methods are still useful at the extremes of the fitness distribution. On the other hand, relative fitness is difficult to calculate while the GP system is running because there is no optimal measure of reference: the best achievable fitness for a problem of unknown solution is often uknown as

well, and using a percentage of best fitness can cause normalization problems when a new best fitness is found by the algorithms.

3.5.3. Selecting for General Solutions

In many cases, genetic programming is used to create programs that must solve more than a single exact problem. Parameters, both related to the solution and environmental, may change without allowing further evolution of the program. This means that the solution generated by GP must be general to the space of possible problems. The level of generality is almost entirely dependent on the choice of fitness function, and how well it selects for a general solution; this generality is often also referred to as the robustness of the solution.

3.6. Setting up Runs – Parameters and Generations

Once the problem representation (genome), fitness function, and genetic operations have been defined, the algorithm can be run and solutions can be generated. Before running, a few parameters depending on choice of genetic operations usually have to be set, however. This can be a very important contributor to the effectiveness of the GA; in some cases important enough to warrant developing a second GA just to evolve a usable set of parameters for the first [15]. A poor set of parameters can result in various behaviour, ranging from completely random generation and regeneration of individuals if random mutation prevails excessively over deterministic selection, to completely linear hill-climbing if the opposite happens.

The first parameter to choose is the number of individuals in a population, or population size. Genetic variability and the effectiveness of the algorithm increase significantly with larger populations, but so does memory consumption and computation time, due to the number of individuals being stored and fitness checks being done. The variation of these factors based on population size is dependent on the search space in question. Generally, the largest population the system running the GA can handle is the best choice. Alternatives to simply using large populations, however, do exist: the simplest alternative is making multiple separate runs with smaller populations in the same amount of time. Depending on the nature of the solution space, this choice can be more or less effective, or have no overall effect on the speed at which good solutions are found. Similarly, the use of multiple parallel populations can be chosen [14], allowing a group of normally-separate populations to intercommunicate through migration. This is a relatively new and unstudied technique that will be further discussed in section 3.7.1.

Next, mutation and crossover rates, known here as m and c, need to be chosen. Generally, the mutation rate has been kept low (between 0.1 and 0) and the crossover rate kept high (between 0.7 and 0.95)[28, 29, 39, 40, 41]. Examples of these choices of mutation and crossover rates are shown in the reference papers in chapter 4. These rates influence how much variability a population will have in several ways: by increasing the mutation rate, one increases the diversity in a population, and adds more new genetic material, but can also cause more random deletion of (potentially useful) parts of individuals. This can have the effect of destroying high-fitness individuals before they can spread

through a population, but it also injects vital new material to keep a population from stagnating. Decreasing mutation has the opposite effect, making high-fitness individuals persist longer, but making it more likely for a population to stagnate and converge prematurely. Increasing crossover, on the other hand, improves a population's ability to exploit high-fitness individuals by moving parts of fit individuals to new offspring. This is especially true in the case of hierarchical representations (such as genetic programming trees) where a good high-level structure could be crossed over entirely, greatly increasing the fitness of other individuals. Unlike mutation, crossover depends on selection: it is thus less random and much less likely to destroy high-fitness individuals. A higher crossover rate (as high as 90%) has shown to be advantageous in many situations [28, 29, 39, 40, 41]..

Additionally, any parameters for the selection method used need to be defined at this point. Of the selection methods described in section 3.3, only the rank or tournament selection are affected by selection parameters: when using rank selection, a higher p will increase selection pressure – less fit individuals are much more likely to "die" the closer p gets to 1, because it becomes more likely an earlier individual in the sorted list is selected. Lower values of p, on the other hand, will increase the randomness of the selection process, by lowering the likelihood that a specific individual will be selected before another. In tournament and k-tournament selection, varying p (now the probability that the best individual in a tournament will be selected) will again increase selection pressure, for similar reasons[30]. Since p is an inverse measure of complete randomness, increasing p removes randomness, and decreasing it makes the random selection of individuals more likely. Note that because the individuals undergoing a tournament round are selected completely randomly from the general population, a tournament winner being selected randomly is functionally equivalent to selecting it randomly from the population. Changing k, the tournament size, is the second way selection pressure can be varied. By increasing the number of individuals competing against each other for a given reproduction spot, the more likely it becomes that one of the best individuals in the population will be chosen, instead of simply a locally best individual.

Finally, a stopping point needs to be set, so the algorithm terminates after a given condition: this can be as simple as telling the algorithm to run for X generations, or until Y fitness has been achieved. Many stopping points relate to the possible convergence of the algorithm – no change after a number of generations, or some measure of variability in the population, such as average fitness relative to high and low fitnesses. Another very common measure of variability is the number of distinct individuals in the population – if out of a population of 2000 there are only 500 distinct individuals, then there is much less genetic variability, and it is less likely for a beneficial mutation or crossover to emerge. More unusual or complex stopping conditions, such as an average fitness value across a population, can be chosen, though the reasons for such a case would be very specific.

From start to stopping point, the execution of the GA a single time is known as a run. Most experiments are made up of many runs, in which some parameters will vary.

3.7. Advanced Genetic Algorithms Techniques

3.7.1. Multiple Distributed Populations

In nature, sometimes a species evolves in separate populations, with only occasional interbreeding between them. The most obvious example of such evolution is the finches on the Galapagos Islands[10], discovered by Charles Darwin. Because the finches evolved in different biomes, often with different conditions, each subspecies evolved a different set of characteristics. This concept has many names, but the broadest definition, known as spatially distributed evolution, states simply that evolution occurs in multiple regions separated by distance but at the same time.

Each of these populations undergoes the usual generation process, and is handled completely separately aside from the new migration phase. During this phase, individuals move from one population to another, usually randomly[14]. There are two main schemes for migration – equidistant populations, and spatially located populations. In the first case, all populations are considered to be the same "distance" from each other, so migration is equally likely to occur between any pair. In the second, some populations are closer than others and migration will – like in nature – result easier between the closer populations. In both cases, a migration probability is needed to determine how often migration events occur and how many individuals migrate per event. In the second case, a weighting value is also needed for each population pair, to indicate how far they are away from each other.

The benefits of this technique are largely speculative; in theory multiple populations would improve genetic variability while still allowing high-fitness individuals to be exploited, because they will do so in parallel. Ideally, multiple populations will also better allow a exploiting multiple high-fitness individuals at the same time. The migration rate is a very important parameter, however – if it is not set correctly, the run will simply behave as one large population, or many completely isolated small population, negating the possible benefits of migration but maintaining the extra overhead.

3.7.2. Elitism

Elitism is a concept intended to preserve high-fitness individuals, protecting them from the dangers of being destroyed by random mutation when new populations are generated. The concept is simple: define an "elite" subset of the population as a given percentage of the total population (typically 20%). Then, when selecting new parents for future generations, ensure that elite population is passed directly into the next generation, without a chance to lose its genetic material to mutation or crossover. After recombining the other 80% of the population, the population is then sorted again and the best 20% determined for the next generation; the 80/20% split may be adjusted.

The key advantage to using elitism is that there is no chance to lose the best individuals in a population to random mutation or bad crossover – a population's maximum fitness can only increase, never decrease, when elitism is being used. The downside, however, is that there is a massive decrease in randomness and variability, because the population is only recombining to generate 80% of the individuals it normally would without elitism. It is also quite likely that these high-fitness elite individuals will be selected by the ordinary selection method and recombined, which can lead to a large

amount of the population's genetic material coming from a small percentage of the population (because of the 20% automatic selection caused by elitism), further reducing randomness. Elitism is very much a double-edged sword and the pros and cons of its use should be carefully weighed.

3.7.3. Penalty and Bonus Functions

There are many cases where a genetic algorithm will produce many results with an incorrect structure, format, or along a poor path to a solution, repeatedly and without significant deviation. These cases generally indicate an incorrect or incomplete choice of fitness function, or that the obvious fitness function is insufficient for directing the progress of evolution. In such a case, one option is the use of penalty functions, and their positive counterpart, bonus functions.

A penalty function is an additional component to a fitness function that subtracts some value from the fitness of an individual based on some characteristic it possesses. The fitness function will still run the program; a penalty function will act on observing some characteristic of the run or of the program itself. For example, a penalty function commonly used is one penalizing overly-long programs, because they will have slower execution times, consume more memory, and be otherwise less efficient than shorter ones. This penalty function reduces the viable solution space, and will improve the quality of solutions, assuming the optimal solution is in the space of shorter programs allowed by the penalty function. If it is not, the poorly-chosen penalty function will have eliminated the correct solution, leaving a local maximum as the best approximation.

Bonus functions are simply the positive analogue to penalty functions; instead of subtracting from an individual's fitness, a bonus function adds to it. For example, a bonus function that increases the fitness of SEBeLS programs that make use of the Antbot's sonar rangefinders is a potentially useful tool to include in genetic programming on the Antbot.

3.8. Genetic Programming Concepts

The basic concepts of genetic programming are, as explained in the beginning of this section, very similar to those of genetic algorithms in general. The primary difference between an ordinary GA and GP is the additional level of indirection; while GAs construct solutions to problems, GP constructs programs that will solve problems. This adds significant complexity to the representation of most problems, but it can also vastly increase the power of a GA to solve said problems.

3.8.1. Representation of a Program

To represent a program as a genome, one must consider the basic hierarchy of a program. Programs are composed of classes (or modules, or even files in a filesystem for e.g. shell scripts), which are in turn composed of functions, which are composed of commands; each command has its parameters and components. This type of representation looks very much like a tree, as there should be no data path between lower-level components and higher-level ones – a command cannot include an entire class as part of itself. At best, classes could contain internal classes (such as in Java), offering additional levels to the tree. An example tree representation of a simple Java program follows:

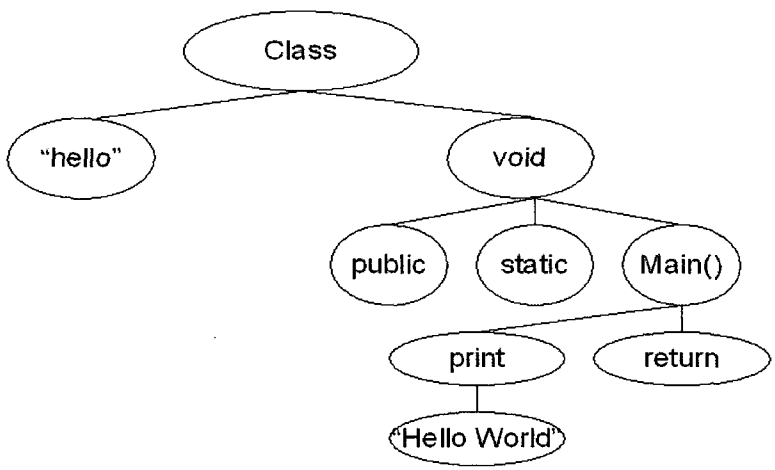

Figure 9: A tree representation of a very simple Java program.

The mechanics of mutation and crossover on trees, discussed previously in Section 3.4, apply directly to genetic programming representations, which is their primary use. All of the techniques from section 3.6 will also apply to genetic programming, as will nearly all other GA techniques, though some may require adaptation to work on the tree structure of a GP genome.

3.8.2. Automatically Defined Functions – ADFs

The use of functions and subroutines in programs allows programs to be more space-efficient through the reuse of code. In the modern day, the matter of a program's size on most systems is irrelevant to performance. In the realm of genetic programming, the purpose of functions is the improvement of the representation of the solution, by defining a standardized structure for the program. This standard structure allows evolution to take a path closer to the human-designed solution, which is especially useful when the program has been previously solved and is simply being further optimized by GP. They can also reduce the size of the search space, if a repetition of an action is required. The evolution of a loop structure can accomplish the same feat, but the human-guided structure of an ADF ensures this function is contained in all valid parts of the search space.

The concept of ADFs (Automatically Defined Functions) was defined by John R. Koza [27] as follows: ADFs are predefined substructures of a program, not unlike functions or subroutines, that may be called as atomic instructions. For example, a program may be composed of three blocks: function 1, function 2, and the main program. Function 1 and function 2 would be considered ADFs – the code in them would not execute unless called by the main program, or by each other. In some cases, an ADF can also call itself (and is then known known as a recursive ADF). This recursive property, in most cases, becomes a parameter, which can be enabled or disabled based on the hypothesis of how well it will benefit the run in question.

The benefit of using ADFs in genetic programming is to make the algorithm more scalable to larger problems. The intent is, through constraint of the possible solutions, to direct evolution in a positive direction using knowledge gained from manual attempts at solving the problem, or from previous GP experiments. Additionally, the ADFs allow for some "scratch-pad" evolution – the development of genetic material that can be turned on and off. By evolving a function that can be called at will, the still-evolving genetic material of the function can be called, or not called, depending on its overall fitness, using a single operation in the main program. This is similar to promoter genes in biology, which regulate the transcription of otherwise non-coding DNA. This non-coding DNA can then evolve either randomly (when not being used, and therefore not under selection) or selectively (when being used, and under selection). The same is true for ADFs.

3.8.3. Module Acquisition

The idea of module acquisition was developed by Peter Angeline and Jordan Pollack[4]. In their paper, they show that the concept of shared modules is very similar to that of ADFs – the division of a program into a group of sub-programs, rather than a single monolithic element. Like ADF usage, module acquisition gives a similar benefit to promoter genes in nature – a function call to a module can be turned on and off as a single atomic instruction. This allows code to evolve under selection when active, and randomly (not being selected for or against) when inactive. Unlike ADFs, modules are not isolated to the individual that evolved them, but are instead global and reside in a library shared among the population. As such, it becomes extremely easy for a call to a module to cross over between two individuals, allowing highly-fit code from one to be quickly called, in bulk, by another. Additionally, it allows a module to be under very high selection pressure from multiple individuals – if it is not used, it will simply begin to evolve randomly, or not at all (and eventually be deleted for a new fitter one). In

theory, this will create drastic improvements in fitness over a short time, allowing genetic programming to scale to much larger problems.

3.9. SEBeLS and Genetic Programming

One of the goals when designing SEBeLS was its suitability for use with GP. In terms of being able to evolve useful programs effectively, not all languages are created equally. Because genetic programming is in many ways dependent on a language's tree structure, most extant genetic programming techniques are designed around the "classical" tree representation – by operating on subtrees, it becomes easy to operate on coherent subsections of a program rather than random lines of code [42]. The mutation and crossover operations in section 3.4 are, as shown, a different means of manipulating trees.

The main benefit of SEBeLS in GP is that the hierarchy of the language is designed to suit this tree structure as efficiently as possible. Levels are explicitly defined (program, state, rule, command, command node), so that crossover algorithms can be designed to exploit these without difficulty in determining where level divisions should be made. It also limits this crossover to within specific levels, again reducing the chance of useless crossover. Other languages, especially LISP, do have this property, but are less suitable for writing rules and defining AI than SEBeLS, due to its specialized design. In addition to having a defined hierarchy, SEBeLS has explicit definitions of states and rules, so that AI state machines can be quickly translated into code. Chapter 5 discusses the matter of designing a genetic programming framework in more detail, and its direct application to the SEBeLS language. As a note, while the problem never presented itself in simulation or tests, the SEBeLS implementation on the Antbot allows skipping over commands with malformed syntaxes. In the next section, the current state of genetic programming research as applied to mobile robots will be explored in detail.

4. Genetic Programming – Related Work

Genetic programming is a vast field of research, and there have been many attempts in the past two decades to apply the principles of GP to robotics. The original proposal for using genetic programming for robots was written by John R. Koza, in 1992 [28]. It ran a simple box-moving experiment, using simulations and LISP to program a robot that would push boxes from the middle of a room against the walls. No real-world tests were ever performed, but the robots were able to accomplish their simulated tasks, providing basic proof that genetic programming could be used for robotics, at least under ideal conditions.

4.1. Genetic Programming on Single Robots

The question of whether any single robot can be programmed by genetic programming has been studied many times, beginning with Koza and continuing forward to this date. The majority of work, however, has not been on navigation, or aimed at any form of generality, but using genetic algorithms to adapt the gait of humanoid or quadruped walking robots, optimizing machine vision, or otherwise improving the component tasks of robot operation. These topics are not relevant to core navigation, because once these algorithms have been designed, evolved, or otherwise generated, control returns to the human programmers and operators. A more holistic approach, where the genetic algorithm has full control of the robot removes more need for human programming, as is discussed in chapters 1 and 2. There are two subsets of a holistic GP approach on single robots: general problem coverage, and specific task optimization. In the latter case, a specific task problem (such as playing soccer, following a light, or navigating a corridor) is presented, a genetic programming experiment designed, then run and the results described. In the former case, one or more test tasks are used in the above manner, but the task is not the key to the project: the real test is to see how the genetic programming framework itself performs on the platform, with a simulation environment, or under other specific conditions, in order to determine its usability on a larger problem space.

4.1.1. General Experiments

The groundwork for the task-specific experiments is set through the choosing of techniques and the development of generally-useful genetic programming frameworks. The intent of these experiments is to further the science of genetic programming on robotic platforms in general. Most of these experiments, however, have one failing in common: a lack of field testing in environments where a robot is expected to operate, existing only in simulation. Because of the generality of these experiments, it is difficult to say what the true state of the art is, but a few experiments will be

explained here, and their contributions and shortcomings analysed.

One such experiment by Miller and Harding uses Cartesian genetic programming [35], a technique that replaces the usual tree-based representation of the program with a graph in the Cartesian plane. The test task is the classical problem of obstacle avoidance, and though he does not conduct field tests, his simulated robot is based on the Khepera robots [24], which have considerably more rangefinders, but less computing power, than the Antbot. His test is extremely successful, and shows the effectiveness of Cartesian GP for programming robot controllers. The problem, however, is that Khepera robots are designed as testing tools, and have no function outside of a laboratory environment; Antbots are used in teaching and surveillance, and are being evaluated for bomb squad duty.

Some very interesting general-interest work was conducted by Gregory Barlow in 2006 using unmanned aerial vehicles (UAVs) and genetic programming [6,7]. His experiments endeavoured to refine the simulation of the real world to include noise that simulates real-world inaccuracy of sensors, though he limited his experiment to radar detection and high-level navigation, leaving low-level flight control to a human-engineered autopilot. This is a very similar choice to leaving the NAVCOM or Antbot motor controller in charge of vehicle operation, and using a SEBeLS script for navigation. Once the selection for flight to a radar point and circling of said point in a stable manner was completed, though, a test for robustness was performed on the best 10 individuals created. These tests were run using the same simple simulation used for evolution, but a variance on angle of approach, radar amplitude, airspeed, heading, and wind effects were all tested. These tests added considerable difficulty to the task for the evolved programs, and showed that there were many cases of overfitness to the simulated environment that could have been avoided if selection pressure had been lowered. Note, however, that the hand-written controller used as a control still had significant problems with some of the robustness tests. From experience observing NAVCOM AI tests in sea and land vehicles, this shows a poor expectation of overall robustness. Still, the fact that real-world tests were scheduled after the publication of the paper shows that the results will at least be meaningful and can be used to further improve UAV navigation outside of simulated environments.

Peter Nordin, along with Wolfgang Banzhaf, have also done some useful general work on adapting GP to robotics problems [39, 40, 41]. They took the unusual step of employing real robots for their generational tests, necessarily with a very small population size. Their results were extremely optimistic for future developments in genetic programming on mobile robots, despite a tiny population. In only a few hundred generations, the system has a relatively general obstacle-avoidance system, evolved from real-world robot testing. These results are nothing short of miraculous. The drawback is that the system used is specific to the Khepera robot used, and the robot is very capable of damaging itself during evolution: for example, they encountered several problems with overstressed motors due to the robot trying to drive through a physical barrier (the Antbot platform has a low-level emergency stop system to help prevent this). Later, they repeated the experiment allowing the robot's memory to persist between generations, in a way that meshes with other forms of online machine learning [2, 40], such as Stanford's STAIR project [38]. Though the results of this method demonstrated perfect obstacle-avoidance, the use of real-world learning is still questionable in field robotics due to the obvious risk of damage to the hardware, as in [39].

Along with Markus Olmer, Banzhaf and Nordin created a genetic programming strategy for robots that first uses GP to learn subtasks, then evolves a higher-level strategy of selecting subtasks to perform to solve a larger problem [41]. Again, the Khepera robot and its simulator are used, and even though functional controllers were evolved the results demand further work.

Related to Nordin and Banzhaf's experiments is Marcin Pilat's repetition [42] of the simulated portion of the experiment, using an improved version of the Khepera GP Simulator [33] used by Nordin and Banzhaf in [39] and [40]. He also conducted the experiment using different representations, using both linear and tree-based genomes, along with genomes using ADFs [28], Module Acquisition [6], and a more recent technique known as Adaptive Representation [50]. All of the hierarchical representations performed more effectively than the linear genome, but the wall-following and obstacle avoidance did not match the "perfect" results quoted by Nordin and Banzhaf.

The last paper to mention is a recent one by Doncieux and Mouret [11]. In it, they describe the abstraction of tasks into a series of independently evolved subroutines, evolved in parallel and assembled in a human-designed main program. This allows for considerable increases in performance for complex multi-task problems such as a simple game of basketball. Though the idea of a human-designed main routine breaks one of the main goals of genetic programming, the significant increase in performance shown gives it merit against purely automatically-generated programs.

4.1.2. Task-Specific Experiments

Task-specific experiments make up the bulk of genetic programming experiments on robots, and for good reason: many robots are specialized to individual tasks, and even in the case of more general purpose robot platforms such as the Antbot, every GP experiment will inevitably be selecting for the ability to perform the task under exam.

One of the first successful task-specific GP experiments was the wall-following experiment done by Koza as a follow-up to his earlier box-mover, in 1994 [30]. Again, a simulated robot was programmed using LISP, and it executed its tasks extremely well, achieving perfect fitness scores. The fitness function used, however, was extremely strict in selecting strongly for robots that press closely to the wall and follow a specific path. Additionally, no physical test was conducted to ever confirm that the results of the experiments had any relevance outside of simulation.

Two similar experiments were conducted four years later by Robert A. Dain, and separately by Marc Ebner. In Dain's paper [9], wall-following is evolved using a set of pre-set functions, including functions to atomically turn toward and away from the closest wall. Good results were achieved – this level of human assistance shrank the program to a handful of decisions, and there was no sensor noise in the simulated system. Ebner's work [12], on the other hand, is closer to Koza's, using a more realistic command set of turns, sensor readings, and forward/reverse motion. Like Koza's experiments, the simulated robot is programmed in LISP. The simulation map and robot chosen were designed to mimic the real environment of the test lab and a Real World Interface B21 robot, a rather large mobile robot

with many rangefinders. With short (50-generation) runs, and small populations of 75, they achieved a best-case scenario of 40% success (8 good individuals out of 20 runs) in simulation. A real-world test using the physical robot in the lab was then performed, allowing the program to evolve on the physical form rather than using the best simulated individual. After two months of work, the best evolved result produced reasonable results. The primary issue with this experiment is that it is not general, but designed to be very fitted to the test environment – it is possible that the evolved navigation program used dead reckoning, rather than being sensor-driven.

Yet another track follower, this time for the purpose of simulated auto racing, was developed by Togelius and Lucas [58] to show the possibilities in scaling-up of genetic programming. They describe a series of racetracks using waypoints and walls, and simulate a car equipped with a position sensor, heading sensor, and rangefinder, very similar to an Antbot or similar vehicle with a compass and GPS module installed. The results on the general navigation experiments showed promise, though the algorithm was unable to devise any controllers capable of navigating the most difficult tracks. An attempt at parallel evolution was an outright failure, never able to complete a lap of any of the tracks. Next, sequential evolution was tried; a single track was used in the training set, then another was added after the population reached a given average fitness. Performance here was quite good, but still failed to solve the most difficult tracks. The paper makes many good observations about GP on mobile robots, such as noting how commands from the robot controller to the motors tend to "thrash", changing almost every time step. Another interesting thing about this particular experiment is how the genetic algorithm was configured: 50% elitism, and no crossover at all, only mutation and copying. This paper shows that flexibility and genetic variance are very important for robotics problems, through its complete disuse of crossover.

More recently, more specific tasks than navigation are being selected for, and navigation is relegated to test tasks for new GP techniques. One of the most important tasks is cooperative control of networked robots; now that a newer version of the Antbot (and therefore SEBeLS) with mesh networking capabilities has entered production, these papers have become related to possible followups of this work. The first of two key works found deals with cooperation between UAVs [7], by Barlow, the other with land-based rovers and specializing them for object detection [38]. In both cases, genetic programming was used to evolve effective co-operative strategies so that these robots may interact with one another.

Another interesting task for autonomous vehicles is path planning, and genetic programming has yet again been applied to attempt to optimize it without human intervention. Two interesting papers have been written on this subject; the first, from the University of Maribor [26] details very briefly path-identification by a robot similar to the Antbot, using simulated rangefinders for vision. The sensors are more powerful and abstracted than those on the Antbot, however (in order to reduce the impact of sensor noise) and the actual results terrible, with only 4% of tests passing at all. The second paper, by Carl Hein and Alex Meystel, deals with robots in space navigating using thrusters, and optimizing for a minimum amount of travel time in the obstacle field [20]. This is mostly unrelated to the current work, and the results were entirely theoretical, without even a physics simulation to prove the paths developed would be navigable in real-time. Additionally, it appears to assume an omniscient robot as far as obstacle awareness goes, something impossible without a second system adding to the

overall complexity of the mobile agent and requiring more human-derived code.

A final specific task that has seen significant genetic programming work is RoboCup Soccer [46]. Most soccer robots are human-designed, though many attempts at designing them with genetic programming have been made. Some cannot get past simple light-following, or in this case ball-following [8], but others make excellent progress towards evolving soccer strategy [31]. The RoboCup games should provide a good source of fitness functions for improving the overall state of genetic programming on groups of mobile robots for some time to come.

Interestingly, not much research in this sense has come out of robot combat, despite the high budgets and media profile; Robot Wars champion Reason Bradley (Team Toro / Inertia Labs) confirmed when asked that most combat "robots" are either entirely teleoperated or use only basic sensory aids such as gyroscopic accelerometers.

4.2. Other Approaches

Aside from genetic programming, there are other approaches for programmatically creating software for adaptable robots. The two most promising approaches are online machine learning via database, and replacing the single robot with a swarm of interconnected small robots that handle simpler tasks.

The online machine learning approach, however, is computationally intensive beyond the performance of most embedded systems, and certainly that of lost-cost ones such as the Antbot. The state of the art in online robotic machine learning at the time of this writing is Stanford's STAIR project [38]. It is able to learn to identify and manipulate new objects, and mimic human tasks such as cooking, organizing tools, or operating simple machines such as opening doors. The problems with STAIR are twofold: first, the need to retain access to the recognition database and training algorithms, which require significantly more computing power than available on embedded processors such as the Antbot's Propeller chip, or even more computing power than the mobile robot can carry. The second problem is less serious, but part of the nature of machine learning: the need to train the system in the field. If the task is potentially destructive to the robot, this makes training the robot difficult, if impossible, and a simulation-based interface would need to be developed, then field-adapted, which is an entirely new project. Overall, STAIR, and other machine-learning robots, are still many years away in computing advancement before becoming viable in the field.

Swarm intelligence, exemplified by the very successful Swarm-Bots [36] project, and its successor Swarmanoid [57], provides a good alternative to using a single robot. These intelligent swarms of small networked robots are able to assemble themselves into different forms in order to do jobs intended for larger machines. These swarms use genetic programming to evolve a neural network, rather than a traditional computer program. This is a very useful approach for full autonomy, but for navigation systems such as the NAVCOM, the lack of human readability can make the system unsafe for tasks that are not completely automated. Though Swarmanoid only exists in simulation aside from the basic docking/assembly function, Swarm-Bots has shown that genetic programming can be used to

evolve swarm AI that can assemble robots to perform a wide variety of tasks. They are capable of climbing tiny stairs, mounting ramps, crossing gaps, and moving larger objects by assembling themselves together in ways determined by genetic programming using simulated runs. The main difficulties with the Swarm-Bots are their poor ability to handle objects not designed specifically for them: real-world terrain (rather than steps and slopes a few cm high), and objects not designed for Swarm-Bot docking. The project proves the concept that genetic programming is suitable for swarms as well as single robots, and is a potential alternate solution to the problems this thesis tries to solve.

4.3. Robotic Simulation

In many of the experiments in 4.1, instead of developing a simulator based on a new robot for each experiment, the Khepera robot was used, simulated by Olivier Michel's Khepera Simulator [33] or Pilat's recent Windows port of the same [42]. The problem with using Khepera for a real-world experiment is the simple fact that the robot's design is suited only for laboratory experiments. In any kind of real-world terrain, the Khepera robot's drive system would become bogged down, and the robot would topple over easily. Finally, the prohibitive academic license for the Khepera simulator completely removes many avenues of future work on SEBeLS using the simulation, confining it to academic work alone. These factors together make it clear that a new, free simulation system must be designed for a less expensive, more modern platform, such as the Antbot, below.

The use of the wall-following and obstacle-avoidance problems in these simulations is relatively ubiquitous. The reasoning for this choice of problem is that any mobile robot needs to, at the very least, be able to move safely and quickly around its environment so it may perform its assigned tasks. One of the original genetic programming problems for robotics was wall-following, used in Koza's 1994 experiment [30], and was repeated in many later experiments in genetic programming for mobile robots as a simple test problem [9, 12, 39]. Finally, evaluating the fitness of a wall-following program is a simple problem to solve – the distance travelled in a "good" direction without contacting any of the walls is an effective measure of fitness for such a robot, given a common time limit between individuals. Using an appropriate simulator, this fitness value can be computed cheaply, using data already required by the simulation.

4.4. The Robots Everywhere Antbot

The Antbot [48] is a product by Robots Eveywhere, designed for hobbyist and student use. Its most attractive feature is that its software architecture is almost identical to the NAVCOM's, but much simpler. It has been chosen because of this similarity, along with its low cost, and access to some features that have not yet been implemented on the NAVCOM, such as the NOSIC (Navigation-Oriented Symbolic Instruction Code) scripting language, which is a language similar to BASIC. This allows for more scripting flexibility, and enables a greater degree of field-programmability than with the current NAVCOM system. Robots Everywhere has assured us that there will be comparable features on the NAVCOM in the near future, giving the experiments further validity.

4.5. THE NAVCOM AI

The NAVCOM AI, developed by M.K. Borri, is a modular, low-cost general autopilot for surface vehicles, boats and airplanes. It can be deployed on any remote controlled vehicle. Coupled with a serial packet radio, a GPS and at least one other position sensor (another attached GPS at some distance from the first, a compass or an accelerometer, rangefinders, or other sensors) the NAVCOM adds autonomous capabilities, as well as the ability to return telemetry to the base station, from any remote vehicle; manual radio override, as well as a NAVCOM-driven "jog mode" for testing, are provided as well. The NAVCOM can receive, process and record NMEA input from standard and user-defined sensors for measurement, obstacle avoidance and data logging to an optional SD card, and generate NMEA output for other instruments as well as PPM and servo pulses to drive actuators directly. It uses a very powerful microcontroller designed by Parallax, called the Propeller[43]. The Propeller's 8-core architecture allows realtime handling of all acquisition, navigation, actuation and logging tasks [52]. The current version of the NAVCOM runs a user-defined state machine with limited over-the-air runtime reprogramming. The next version will allow the entire state machine to be modified at runtime through the packet radio. Waypoint and position data is transmitted and received using either NMEA or KML(Google Earth) format [48].

So far, NAVCOM-equipped drones have been able to sail, fly, and drive on both land and water to accomplish various tasks, such as survey and payload delivery. More complex tasks are easily accomplished by having the NAVCOM interface with another piece of equipment, such as a robotic arm, controlling it via servo pulses. Part of the NAVCOM's modular AI function code's job is to allow programmers to write an interface to this sort of hardware, or even more complex hardware, such as another microcontroller, communicating via RS232 serial communications. The NAVCOM can be field-reprogrammable to a limited extent; its state machine must be programmable offline but parameters and expressions can be altered on the fly.

What would benefit the NAVCOM greatly is the ability for its AI functions to be adapted more easily to different mechanical systems, even using the field-reprogrammability of the AI to control them. In addition, if it were made possible for the AI to adapt itself to a given mechanical platform, if given a description of the platform, it would make the NAVCOM much more usable for consumer devices, education, and other non-industrial applications where the engineering budget to develop an effective AI function cannot be raised. SEBeLS programs can be sent to the Antbot on the fly – in fact, it is possible to modify a state while the state machine is running if it is so desired.

5. A GP Framework for SEBeLS

Upon completion of the SEBeLS parser for the Antbot, the next step was the creation of a framework to run on a more powerful computing platform, designed to apply GP techniques to the new language. The primary goals of this framework are portability, fast execution speed, and modularity, so that algorithms can be changed with relative ease. The modularity quickly became the most interesting of these goals to fulfil: speed and portability can mostly be satisfied by a good choice of programming language, and an optimized implementation.

5.1. Outlining the Requirements

The goal of modularity is a rather abstract one to make into a concrete set of system requirements. Effectively, what is required is the separation of four things into subsystems:

- The genome representation of the SEBeLS programs and its specific genetic operators
- The selection methods
- The fitness functions
- The experiments themselves

With this division made, we can begin to consider a set of requirements that will formally define this division, along with the other needs that must be met by the GP framework. We know the functional divisions of the system, which are a significant portion of the system's requirements; these requirements are relatively simple, and explained below.

5.1.1. Requirements for the GP Framework

1. The genetic programming system shall use a genome representation that can represent all possible SEBeLS programs, given a specified set of external commands.

   1.1. The system shall have a means of creating new external commands in a standardized way.

   1.2. The system shall allow any number of external commands.

2. The system shall perform the basic genetic algorithm on this SEBeLS genome, for the purpose of genetic programming. For a description of the algorithm itself, see chapter 3.

   2.1. The system shall allow extensions to the genetic algorithm, which should be separate from the genome, selection methods, fitness functions, and genetic operators.

3. The system shall never restrict itself to a specific set of genetic operators.

   3.1. The system shall have a means of creating new genetic operators in a standardized way.

3.2. The system's genetic operators shall be independent of each other, and any fitness functions, selection methods, or experiment structures.

4. The system shall never restrict itself to a specific set of fitness functions or problems.

   4.1. The system shall have a means of creating new fitness functions in a standardized way.

   4.2. The system's fitness functions shall be independent of other fitness functions, selection methods, experiment structures, and the genome.

5. The system shall never restrict itself to a specific set of selection methods.

   5.1. The system shall have a means of creating new selection methods in a standardized way.

   5.2. The system's selection methods shall be independent of other selection methods, fitness functions, experiment structures, and the genome.

6. The system shall have a means of executing the genetic program at any level of the SEBeLS structure.

7. The system shall have a means of executing the genetic program from any point in the SEBeLS code.

8. The system shall be functional on Windows, Linux, and OSX-based computers.

9. The system shall be functional on x86, amd64, and PPC processor architectures.

5.1.2. Analysing the Requirements

With the requirements formally written, the next step in the development of the framework was the conversion of these requirements into the fundamental decisions that outline the design of the application. To determine the main design decisions for the framework, we must consult the requirements and determine which should be prioritized as core requirements, and which should be added to those as features. Upon looking at the requirements, it seems requirement 2 is the key core requirement, and should be considered immutable in the design; this is a simple functional requirement and does not require any design decision to be made at all – it is just code that needs to be implemented.

The next significant requirements are requirements 1, 3, 4, and 5, and their sub-requirements. These are the defining requirements of the system architecture, and will be responsible for the top-level design. What these requirements suggest is that there is a group of selection methods, fitness functions, and genetic operators independent of each other and of the genome defined in requirement 1. This suggests the package arrangement should centre around these four key concepts: the genetic algorithm, the genome, the fitness functions, and the selection methods. This gives us a potential package structure, but no arrangement.

Determining the arrangement of these packages primarily comes from observing the functional coupling of the four elements in the genetic algorithm itself. The algorithm sets up the population using the genome, and accesses the selection methods which in turn make use of the fitness function. This creates a relatively coupled package architecture, meaning there will be a significant need for interfaces or abstractions to allow for the flexibility required by requirements 3 through 5. The package architecture will become rather arbitrary, but given the small number of packages, and likelihood for single-class interfaces, the design should still be relatively sound.

The next requirements that need to be addressed in the design are 6 and 7; these requirements require that any part of the genome must be executable, and that execution will proceed from that point. This provides two potential methods of execution: tree-based execution, where execution will only proceed in the subtree of the program, or start-point execution. Both of these need to be provided, but thankfully this is relatively simple given the design decisions already made. All of the abstract classes for the genome will require an execute() method, letting them run once as their own subtrees. Additionally, the starting state of the entire program will be given public access methods, so it may be changed before the beginning of a run. This should satisfy these two requirements. The remaining requirements are implementation requirements and will be addressed later.

5.1.3. GP Framework Architecture

With the basic package architecture outlined, we can begin the actual design of the GP framework. In section 5.1.2, we decided that the system would be composed of four main packages: the genome, the selection methods, the fitness functions, and the genetic algorithm itself. The final package added was called SmallStuff; it is simply a library containing the logging function and other useful functions not bound to any of the other packages. Their intercoupling can be shown by the UML package diagram at right:

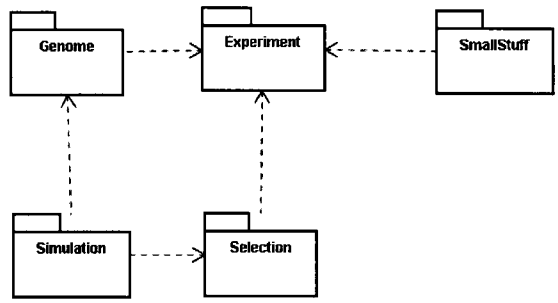

Figure 10: Final package diagram for the GP framework.

The Experiment package contains all of the different experiments to be run on the GP framework. The actual genetic algorithm is contained within this package, defining the basic structure of each run. The architecture of this package is a simple bag architecture, because each version of the algorithm is an independent entity of its own; and the package simply exists to organize them by function. No UML diagram is shown for this package because there is no organization worth noting about a bag structure.

5.2. Designing the Representation

The first step towards designing a genetic algorithm is to determine how the problem the algorithm must solve can be represented: without a representation, the genetic operators and fitness function cannot be designed at all. The representation for SEBeLS, however, is simple and needs little design work to fit it into a genetic programming framework. The SEBeLS language, according to the grammar in 2.4, is naturally hierarchical, with a program decomposing into states, which decompose into rules, which decompose into commands. This produces a natural tree structure, as shown in section 3.2; this basic representation can be used for genetic programming without any changes. The console commands (commands beginning with @ and part of the NAVCOM and NOSIC command set) are further decomposed into their component trees, called "nodes". Each node contains an element of the command, such as a parameter or operator, which combine with the parent command node to form a functioning command.

Within the representation, the generation of new random individuals (such as at the beginning of a run) is also a necessary consideration. The creation of these individuals is not truly random, due to no hard-coded limit on the depth of an individual or length of a rule – the generation space would be infinite. Instead, an individual with a single state, node, and empty rule is created, then mutated at the program level a random number of times from 1 to 5, with the number chosen from a uniform distribution. For random floating-point and integer values, they are initialized to a random number between -180.0 and 180.0, due to these values being normally used for a compass heading on the NAVCOM. On the Antbot, this was mostly arbitrary initialization, but was kept to see if GP would find a use for these large numbers, due to it having no heading sensor. The following example shows a SEBeLS program within the representation. This example program tree will be used throughout this section for further examples, in part or in whole.

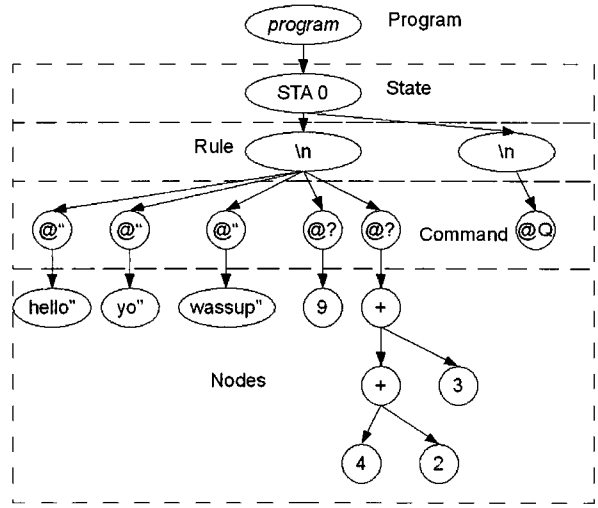

Figure 11: A SEBeLS program as organized in the GP framework

The following UML class diagram describes the Genome package more clearly:

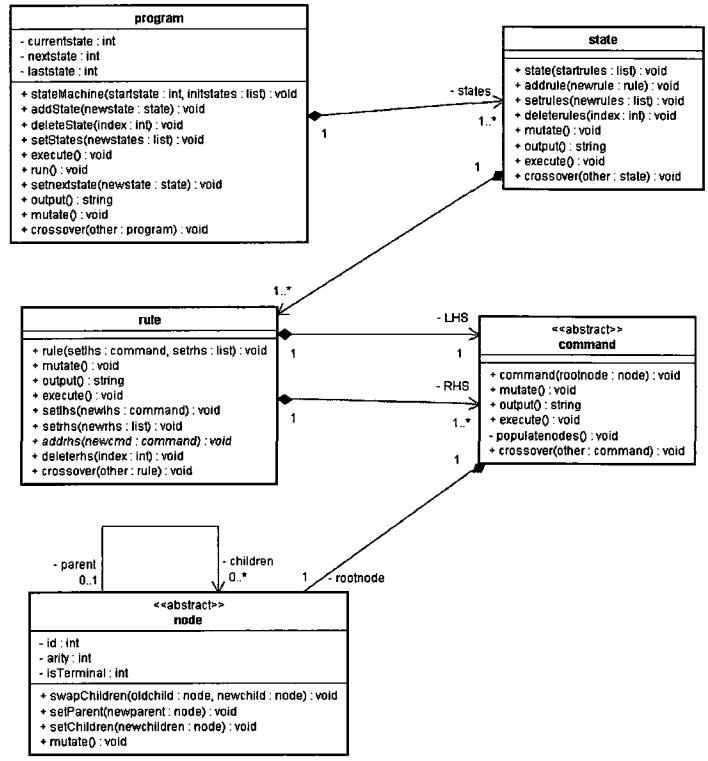

Figure 12: Class diagram for the Genome package.

With the representation clearly defined, the next step is to decide on a selection method.

5.3. Designing a Selection Method

The selection method, and its associated package, the Selection package (below) is simple: a single SelectionMethod interface defines the properties of a selection method, then specializations (often with additional parameters) can be extended from it.

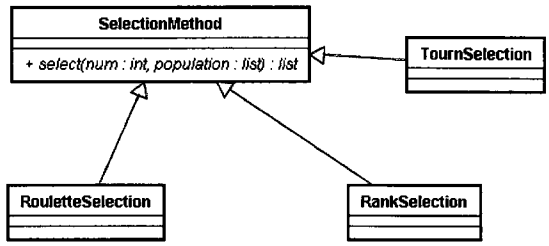

Figure 13: Class diagram for the Selection package with a sample of possible selection methods

The four selection methods shown in 3.3 were all used in the Selection package, as different designs of genetic algorithms may use different selection methods in the future, and such flexibility is good practice. It is expected either Hi-Lo Fit or Tournament Selection will perform well, and the older selection methods of Fitness-Proportional Selection and Rank Selection will likely go unused for this purpose.

5.4. Designing the Genetic Operators

A very important part of genetic algorithms is the design of the genetic operators – normally, mutation and crossover. When using a tree representation, such as in most genetic programming applications, different crossover and mutation schemes must be used, as explained in section 3.4. The crossover and mutation operations designed for SEBeLS were chosen based on the hierarchical structure of the language, and the idea that linear crossover within a segment of a program could be highly beneficial to forward evolution.

5.4.1. Designing the Crossover Operator

The crossover operator for SEBeLS has several necessary restrictions placed on it. Firstly, crossover is only able to occur between two nodes at the same level of the program tree: states can only be crossed over with other states, rules with other rules, and commands within a rule with commands within another rule. This needs to be ensured true in all cases so that programs can run. Secondly, it needs to be possible for a series of nodes that share a parent (such as a series of rules within a state) to cross over simultaneously. Thirdly, crossover needs to be weighted toward smaller changes, rather than large crossovers of multiple whole states; crossover should focus on rules, commands, and nodes. This may be flexed somewhat at the discretion of the designer of the particular GA, so the probability of crossing over on states, rules, commands, or parts of a command are not fixed. This creates a somewhat unique crossover operator that follows the following algorithm. Note that variables a, b, and c are cumulative probabilities representing each level of the hierarchy, explained following the algorithm.

```
1. function crossover(self, other):
2.     q = random(0,1)
3.     if q < a: // crossover on states
4.           starta = random(self.states)
5.           startb = random(other.states)
6.           length = random(0, min(len(self.states) - starta, len(other.states) -
                  startb))
7.           self.states.swap(self.states[starta..starta+length],
                  other.states[startb..startb+length])
8.     elseif q < b: // crossover on rules
9.           statea = random(self.states)
10.          stateb = random(other.states)
11.          starta = random(self.statea.rules)
12.          startb = random(other.stateb.rules)
13.          length = random(0, min(len(self.statea.rules) - starta,
                  len(other.stateb.rules) - startb))
14.          self.statea.rules.swap(self.statea.rules[starta..starta+length],
                  other.stateb.rules[startb..startb+length])
15.    elseif q < c: //crossover on whole commands
16.          statea = random(self.states)
17.          stateb = random(other.states)
18.          rulea = random(self.statea.rules)
19.          ruleb = random(other.stateb.rules)
20.          starta = random(self.statea.rulea.commands)
21.          startb = random(other.stateb.ruleb.commands)
22.          length = random(0, min(len(self.statea.rulea.commands) - starta,
                  len(other.stateb.ruleb.commands) - startb))
23.          self.statea.rulea.commands.swap(self.statea.rulea.commands[starta..
                  starta+length], other.stateb.ruleb.commands[startb..startb+
                  length])
24.    else: //crossover on nodes
25.          statea = random(self.states)
26.          stateb = random(other.states)
27.          rulea = random(self.statea.rules)
28.          ruleb = random(other.stateb.rules)
29.          commanda = random(self.statea.rulea.commands)
30.          commandb = random(self.stateb.ruleb.commands)
```

```
31.         starta = random(self.statea.rulea.commanda.nodes)
32.         startb = random(self.stateb.ruleb.commandb.nodes)
33.         length = random(0, min(len(self.statea.rulea.commanda.nodes) - starta,
                        len(other.stateb.ruleb.commandb.nodes) - startb))
34.         self.statea.rulea.commanda.nodes.swap(self.statea.rulea.commanda.nodes
                [starta..starta+length], other.stateb.ruleb.commandb
                .nodes[startb..startb+length])
35.   return self
```

Algorithm 9: SEBeLS restricted crossover

Note that generally, a < b < c. In this crossover algorithm, there will only ever be crossover within one part of the tree, so that syntax is retained, but groups of adjacent nodes (lines of code, sequential states) can be crossed over in a single operation. This is potentially useful because it allows the main benefit of a linear representation (the ability to cross over multiple sequential lines of code) to apply to a tree representation as well. It is often the case in a system that one component of something considered inferior (such as linear representation for problems like this, as shown by Pilat [42]) can create an improvement in a more advanced system (such as subtree representations). An example of the use of this crossover follows.

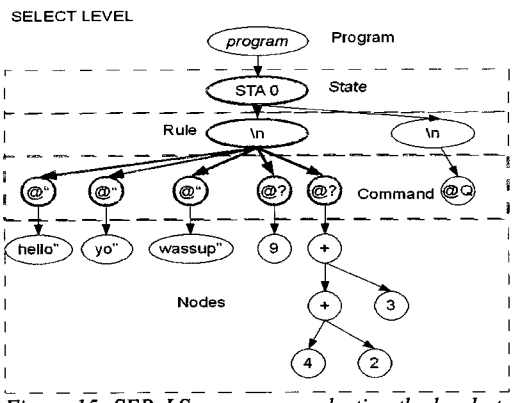

Figure 15: SEBeLS crossover - selecting the level at which crossover occurs

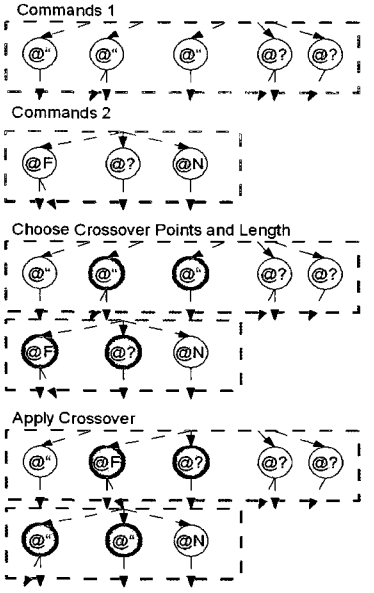

Figure 14: SEBeLS crossover - select the crossover points and cross over

5.4.2. Designing the Mutation Operator

The other key genetic operator in addition to crossover is mutation. When operating on a SEBeLS program, mutation –like crossover– needs to be able to retain the existing program structure and not for example corrupt states into commands. This operation is much simpler than the crossover operator since each level of structure can simply have its own mutation operator; in addition, when an individual is mutated at one level, there is a probability (again fixed by the user) to mutate a random lower level instead.

The types of mutation available for SEBeLS depend on the level of structure being considered. At the higher levels (program, state, rule) entire nodes may be added, subtracted, or changed in position. At the lower levels (commands and their component nodes) values are shifted – a @? command (output a value) may be changed to a @F command (set a variable), and numbers or arithmetic operators may also be moved or changed in value. As a way to reduce the range of numeric values, at this point it was decided that floating-point values (necessary for any navigation system) would be limited to one decimal place of precision, and the values -180.0 to 180.0. This is because angles are considerably more sensitive than distances or times in the NAVCOM, and the idea of using this system to evolve NAVCOM programs later was considered very important. For the Antbot, the results do show that high fitness programs do not commonly have values approaching these limits, so the reduction of the search space here is not deleterious; it also removes possible issues with floating point or integer rollovers. Values higher than 180.0 or with finer granularity than 0.1 can still show up in programs as results of arithmetic operations, but aren't used as starting values for numbers.

A visual example of the mutation function follows:

Before **After**

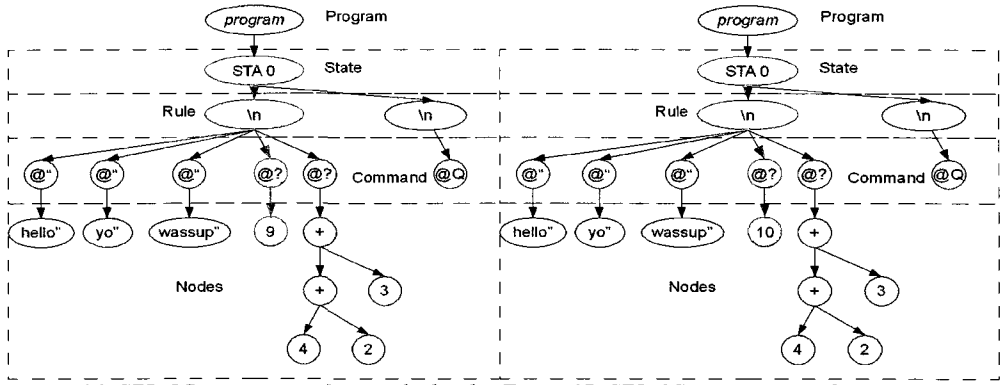

Figure 16: SEBeLS mutation - selecting the level where mutation occurs

Figure 17: SEBeLS mutation - performing an incremental mutation on the selected integer

Before **After**

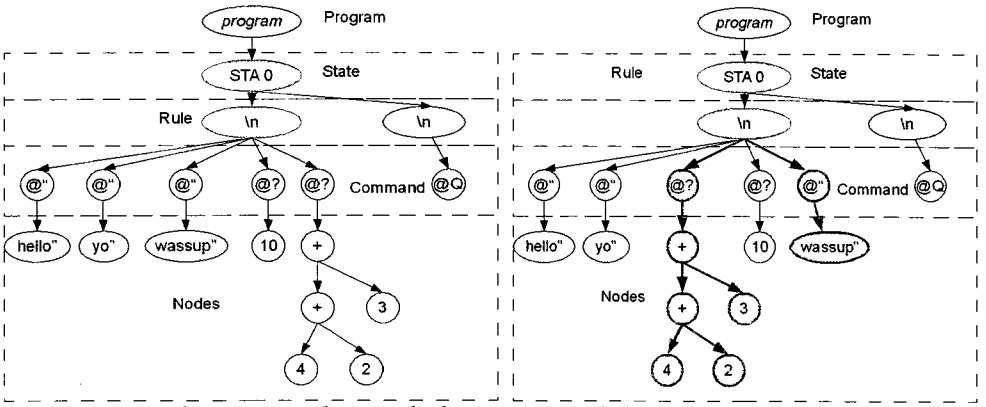

Figure 18: SEBeLS mutation - selecting a higher level (a rule)

Figure 19: SEBeLS mutation - after selecting a rule, swapping the order of two commands.

The complete algorithm for SEBeLS mutation follows, where the all of the single lower-case letter values are between 0 and 1, acting as probabilities for given levels and types of mutation, and all object references (ie self.states) are given following Python conventions. See the UML class diagram depiction of the genome in Figure 11 for details.

```
1. function mutate():
2.     if random(0,1) < p:                          // mutate state
3.           random.random(self.states).mutate()
4.     else:                                  // mutate self: insert or delete state
5.           x = random(0,1)
6.           if x < q:
7.                 self.states.append(new state("random"))   // add a new state to
                                                               // end of machine
8.           else:
9.                 self.states.delete(random(self.states))   // delete a random state
```

Algorithm 10: Mutation on Programs

The other mutation algorithms for smaller elements within a program work in a similar manner, with the differences mostly occurring at the rule and command levels, as shown in the following algorithms. Note that variables and object references follow the same conventions as the above algorithm, and simple operation such as swapping child nodes in a tree, are not detailed.

```
1. function mutate():
2.    if random(0,1) < p:                                    // mutate a rule
3.          random.random(self.rules).mutate()
4.    else:                                                  // mutate self: insert
                                                             // or delete rule
5.          x = random(0,1)
6.          if x < q:
7.                self.rules.append(new rule("random"))      // add a random rule to
                                                             // end of state
8.          else:
9.                self.rules.delete(random(self.rules))      // delete a random rule
```

Algorithm 11: Mutation on States

```
1. function mutate():
2.    if random(0,1) < b:
3.          if random(0,1) < t:
4.                alwaystrue.Toggle                    // toggle the boolean value
                                                       // that determines whether to
                                                       // use or skip conditional
5.          else:
6.                self.condition.mutate()              // mutate the guard statement
7.    else if random(0,1) < p:
8.          random.random(self.commands).mutate()      // mutate one of the commands
                                                       // executed after the guard
9.    else:                                            // mutate self after the
                                                       // guard
10.         x = random(0,1)
11.         if x < q:
12.               self.commands.append(new command("random"))     // add a random
                                                       // command to end of sequence
13.         else:
14.               self.commands.delete(random(self.commands))     // delete a random
                                                       // command
```

Algorithm 12: Mutation on Rules

```
1. function mutate():
2.    if random(0,1) < p:
3.          random.random(self.nodes).mutate()         // mutate a node
4.    else:                                            // mutate self
5.          x = random(0,1)
6.          if x < q:
7.                self.nodes.swapchildren(random(self.nodes))     // swap children
                                                       // of two random nodes
8.          else if x < r:
9.                self.type(random(commandtypes))      // mutate command into new
                                                       // type - ignore arity, extra
                                                       // children are non-coding
10          else:
11.               node = random(self.nodes)
12.               self.node.children.append(new node (random))    // add a random
                                                       // node as a child of another
                                                       // node
```

Algorithm 13: Mutation on Commands

```
1. function mutate():
2.     self.value = random (self.possiblevalues) // mutate to a random possible
                                                  // value based on the node type
```

Algorithm 14: Mutation on Nodes

With the mutation functions designed, the next step is to design the fitness function, and the simulation environment that surrounds it.

5.5. Designing the Fitness Function

5.5.1. Corridor-Following Simulations

As stated in chapter 1, the problem considered in this work as a good candidate for solving by genetic programming is that of corridor-following. This, is a very common problem in robotics, with an easy measure of success: the effectiveness of a corridor-follower is simply measured by the distance it has travelled down the corridor in a fixed amount of time. This distance makes an excellent choice for a fitness measure.

5.5.2. Design of Simulation

Simulating the environment in which a corridor-follower may develop is simple. The fitness functions themselves were designed as turn-based simulations of a square Antbot moving on a Cartesian plane, with a pre-programmed set of walls to act as obstacles (the actual Antbot is rectangular with roughly a 3:2 aspect ratio). The simulation feeds rangefinder data to the variables g, h, i, and j, which store this same data on the physical Antbot. These rangefinders (accurate to within about 1cm) are arranged pointing outward from the Antbot's corners, with a 10-degree detection cone representing ultrasound scattering. Each simulated rangefinder will return the distance to the closest object within its cone of sight; the simulated measurement can be instantaneous since the Antbot does not move at an appreciable fraction of the speed of sound.

The joystick commands, @J2, @J4, @J6, and @J8 are simulated to handle Antbot movement – other joystick commands are left as non-coding genes. The forward and reverse commands move the Antbot forward one unit, which is arbitrarily set to equal 5 times the length of the Antbot. The rotation commands were measured to rotate the Antbot roughly the angular difference a physical Antbot would rotate in the amount of time it takes to move its own length forward or backward, which was approximated to 10 degrees after testing with an Antbot chassis operating purely under remote control. The emergency-stop function of the Antbot is also implemented in simulation, so that it shuts down before it can impact with a wall: in both the simulation and in real life, the emergency stop will halt the running program and stop the motors when an obstacle is detected closer than a distance which can be set as a parameter.

The fitness of the individual is equal to the distance travelled in a direction away from the starting point, after a user-defined number of turns, or if the simulation is interrupted by an emergency stop. A penalty function is applied for emergency stops. In an infinite straight course, this is very easy to measure – in closed courses, a series of checkpoints are used, with weighted fitness values comparable to those of the infinitely-long, straight corridor.

The raw numerical fitness of an individual in the straight corridor is equal to:

```
f =  (d+ (1-e)*w) - 100*e/((1-e)*(p) + 1) + ((1-e)*d)
```

Where d is the distance from the starting location, and w is the perpendicular distance to the centreline of the course from the final position of the Antbot.

In the checkpoint-based tracks:

```
f = (v * c + (1-e)*w) - 100*e/((1-e)*(p) + 1) + ((1-e)*d)
```

Where c is the number of checkpoints crossed, v is a set value for each checkpoint, and w is the perpendicular distance to the centreline of the course (a circle between the inner and outer ones) from the final position of the Antbot.

Two penalty functions exist in the fitness functions above. If the conditions are true, the variable will resolve to 1, otherwise 0. The first penalty flag, e, is applied when the Antbot emergency stops, instead of completing its allotted number of turns. This penalty subtracts a fixed 100 fitness from the individual, marking it marginally worse than one that goes a similar distance without emergency stops. During system testing using the simulations, this value started at a multiplication by 0.5, and was eventually changed, after various iterations, to a subtraction of 100. The other penalty function (p) is more serious, and hinders individuals that do not use the lower-case variables (system data) in the conditions of any rule. These variables, as mentioned previously, contain data from the Antbot's sensors – for the purposes of the simulation, its rangefinders. Without using the rangefinder data, the fitness of any individual in a simulation is limited to a scripted path around the track using dead reckoning, which can obviously never be a general solution for an arbitrary track shape. Therefore, the penalty reduces the fitness of any Antbot without at least one rangefinder variable in its conditions by half. This penalty function only applies if the Antbot does not also emergency stop – if it does emergency stop, it is still significantly possible a beneficial change will involve adding a rangefinder-based condition to the script.

Additionally, a bonus function exists; if the Antbot does not perform an emergency stop, it gains bonus fitness equal to the perpendicular distance between its final position and the centreline of the course. This, again, is a small bonus, but offers a small incentive toward individuals that guide the Antbot down a safer path with no obstacles nearby. This is shown in the above functions with the value d for the distance to the centre line.

5.5.3. Simulation Courses

The simulation courses chosen for the experiments were an angled corridor (a straight line where the starting position of the Antbot faces at an angle to the walls) and a circular track (again, where the Antbot does not face directly down the course). These appeared to be sufficiently different from one another to create some generality in the population, while not consuming too many computing resources by adding more simulations. Each simulation will be run once per individual as the results are completely deterministic: there are no other moving objects on the simulated track. Note that for most tasks, evolving to a specific course (i.e. overfitting) is something to avoid, and a more general solution should be the goal (see section 5.5.4).

Angled Corridor

The first simulation given to the Antbot is an angled corridor, 40 units wide and infinitely long. The choice of an angle rather than a straight corridor is objectively irrelevant to the Antbot, since it is simply a rotation of the frame of reference to a straight corridor. The reason the angled corridor was chosen was to avoid allowing the Antbot to find a round number for a heading, and simply travel along the corridor without using its sonar rangefinders. This program would be an overfit individual, guaranteed to fail future trials. While the initial fitness functions were designed not to overly penalize overfitting, it is still a negative condition that should be avoided when possible. Since the angle of the corridor is 1.1 degrees from vertical and the simulated Antbot uses 10 degree increments, it will be required to make course corrections even on a perfect trajectory.

The sample wall-following program in section 2.4.4. has a fitness of 241 in this test. The best possible fitness for this test is considerably higher (approximately 4000) – the wall-following program used in 2.4.4 is designed for much higher granularity in the real world, whereas the simulation, as stated above, works in terms of Antbot-lengths and 10 degree angles. On the other hand, a program that IS effective under this lower granularity should also be effective in the real world, albeit with some "hunting" behavior on part of the mobile robot.

Annulus

The second simulation is an annulus, a two-dimensional ring much like a typical circular racetrack. The inner diameter of the track is 10 units, the outer diameter 50 units, making the track of equal width to the straight corridor. The choice of this simulation is obvious: it forces the Antbot to change direction rather than go in a straight line, and provides a very different environment from a straight corridor. It uses the checkpoint method explained in the previous section, with each checkpoint being a crossing of the x or y axis, in a clockwise direction from the starting point. The sample wall-following program in section 2.4.4. has a fitness of 225 in this test, using a checkpoint value of 62.5. This is again just a granularity issue and will not affect the results of the programs created using GP. The maximum attainable fitness of this system is also about 4000.

Observations on Simulation

It is clear that the simulation's accuracy leaves something to be desired, though it is also visible that the concepts were present and that the slight inaccuracies would not have any significant effect on

proving the concept that genetic programming can create effective robot AI. Because most of the errors are related to granularity, a good AI in simulation will likely still be able to translate well in the real world – 10 degree increments do exist in reality, whereas 1 degree increments do not exist in simulation, preventing real-world solutions from being as effective. In addition, any simulated solution that relied on excessively fine adjustments would not work in reality due to mechanical factors such as slight differences in motor outputs, which tend to be much more visible after small adjustments.

The choice of values for the checkpoints and penalty functions came from trial and error during the testing phase. Originally, fitness was reduced far more for an emergency stop, but this was found to cause simulated Antbots to earn high fitness by doing nothing, or by moving forward a few spaces then halting. By reducing the penalty, Antbots are now encouraged to move first, and avoid walls second – a behaviour that should, by logic, lead to a better program evolving once competition at higher fitnesses begins.

The value of the checkpoints was simply extrapolated from the length of the track, accounting for the number of moves needed to be spent on turning each 90-degree corner (nine), so that fitness for the annulus would be directly comparable to fitness for the linear track.

5.5.4. Generality of Simulations

As mentioned in 3.5.3, the generality of a genetic programming solution is very important, especially in the case of robotics. A robot in the field needs to be able to perform its task(s) in a variety of environments, rather than just the simulation(s) it was trained with. The definition of generality used throughout this thesis is the amount of variation in the environment that an individual can sustain before failing to operate in a useful manner. In the case of wall-following, this means that there must be more than one course chosen for the fitness function, and these courses should combine to create a relatively general environment. The generality of an individual is then measured by how well it does in another environment – in the case of these experiments, a physical, real-world one on an actual Antbot.

For the experiments explained in the next chapter, two simulations were chosen: a straight, infinitely-long track, and a circular one. These two simulations were chosen to increase generality – more simulations would be ideal, but these two were chosen due to the inherent difference between lining up to a straight corridor and navigating around a circle. These simulations were explained in the previous section. The total fitness of an individual will be the product of the fitnesses of the two runs. The values for the annulus' checkpoints are done in such a way that comparable distance equates to comparable fitness in each simulation, so one does not significantly outweigh the other. Note that because these are discrete simulations, and gain no benefit from being run multiple times (the result is always the same), the resulting fitness is from one run of each simulation.

5.5.5. Package Design

The Simulation package is partner to the Experiment package, carrying the fitness functions that simulate the actions of a robot running the SEBeLS program. Fitness functions are represented by a simple abstract class, with which all of the experiments will interact. The actual fitness function classes

will all implement this abstract class, and be contained within the package in a single-level hierarchy. An example UML diagram of the Simulation package, with some sample fitness functions, follows:

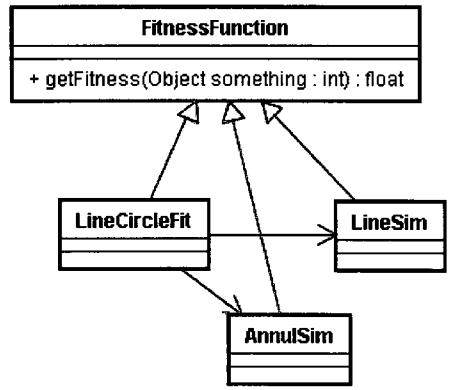

Figure 20: Class diagram for the Simulation package.

5.6. Implementing the System

When considering the implementation of the GP framework, additional requirements come into play. These are requirements 8 and 9, which define the required operating system and architecture compatibility. In this case, requirement 9, the architecture compatibility list, is mostly irrelevant – the framework does not require any direct interaction with hardware, and can work entirely through API calls to the operating system. As long as these architectures support all of the operating systems defined in requirement 8 (which is the case), requirement 9 is satisfied.

5.6.1. Choosing a Language

When considering the language of implementation, there were several concerns raised. First and foremost is the issue of version control – for multiple architectures and operating systems, how many versions of the program will be required, and how significant will the differences between versions be? This became the most pressing question in the end, and became the major source of our decision. The next question was of performance – speed and memory efficiency. Many high-level languages have serious problems in this area (such as Java), and are often considered unusable due to their poor speed. On the other hand, a low-level language such as C would provide version control issues because of different compilers for different architectures and operating systems resulting in different behavior for floating point numbers, random number generation and the like.

Once the tradeoffs were all considered, the implementation of the framework was done in Python, due to its similarity with the Spin language used on the Propeller, its especially excellent portability, and quite reasonable speed, especially compared to other high-level languages such as Java.

The single-core power of Python, plus its ease of writing an efficient and optimized implementation, has made it a serviceable choice for this version of the framework.

Much of the loss of speed generally attributed to an interpreted, high-level language such as Python has been made up for by the use of the Psyco libraries[44]. Though the optimization of the code is questionable at best, Psyco has still yielded between a twofold to threefold increase in speed from implementations without it. On non-x86 compatible architectures, however, the framework's performance is extremely lacklustre, due to the lack of Psyco having implementations on these systems. Though requirement 9 is met by the system, its performance is largely inadequate, and at this time experiments are only being performed on x86-compatible systems, and systems that can natively emulate x86, such as amd64.

An additional note on Python is how the random number generator functions. Since genetic programming requires a lot of random values, for everything from selecting individuals to choosing the type of crossover or mutation, knowing the details of the system's random number generator can be important. The basis of the pseudo-random number generator used in Python is, like in many languages, a uniform randomization of a floating-point value between 0 and 1.0, inclusive at the high end [45]. This uniform function comes from the Mersenne Twister [32], a common pseudorandom number seeded by current system time. This uniform floating-point value is in most cases then used directly, or multiplied to create larger integer or floating-point values.

5.6.2. Testing the GP Framework

A few simple tests were used to ensure the GP framework's correctness and that it meets adequate performance and compatibility standards according to the requirements. These tests were designed in order to ensure the GP framework would perform accurately in the simulations and not be the cause of experimental error, and to verify that it is optimized for proper use of CPU cycles. The tests also ensured that development of simple runs can be done in a reasonable amount of time.

The fine details of the unit-level tests will be omitted for the sake of the readers; every node, command, and genetic operator was individually tested to ensure correctness, and the code inspected for algorithm efficiency. The system was then integrated, piece by piece, and the tests on every element repeated until the full system was placed together. The unit tests were performed in the following order, on each of the relevant classes:

- Internal Methods
- Execution Correctness
- Output Correctness

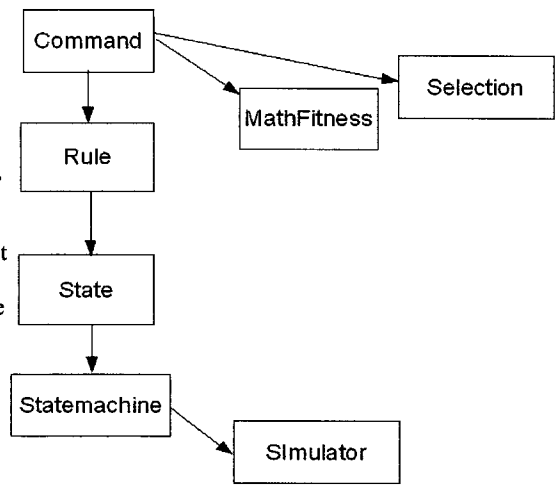

Figure 21: Testing hierarchy for the GP framework

When testing internal methods, the test cases were selected so that boundary conditions would be tested, and any potential failure conditions located. For example, when testing the mathnode class in the Genome.mathlib module (a module in Python is essentially a sub-package), the first test to be performed was to test the constructor (the __init__ function, in Python), then test its constant(), variable(), and mutate() functions for correctness. Mutate was run repeatedly until all of the correct cases materialized themselves – the others were simply tested once to ensure absence of memory errors, since they are very simple functions.

One important error found during testing in the mathlib module was the error caused by multiplying two large numbers producing infinity in some cases. This infinity is similar to not-a-number in the Python language, and is difficult to do anything mathematical with, given that large numbers nowhere near infinity cannot be treated as infinite in any sort of precision instrument. A change in the implementation, where any time a constant or variable is called, it will force a value less than kinds.default_float_kind.MIN, which is the maximum processable number Python can handle, solved the problem reasonably quickly. The same will occur for negative values and kinds.default_float_kind.MIN, to avoid negative infinities. A similar safeguard against overflows and divisions by zero exists in the Antbot's firmware.

5.6.3. System Testing Using Symbolic Regression

Once the unit and integration tests were completed, the first system test consisted of performing symbolic regression on a single rule. This test repeats the classic GP problem of finding an approximation or exact replication of a mathematical function. In this case, it makes use of the Rule and Mathnode classes to create programs in SEBeLS to represent these functions. Three functions were chosen: $5x+4$, $-5x+4$, and $5x^2+4$. Each of these tests were repeated 30 times. The tests passed, and in all three cases at least one-third of the runs found an exact replication of the function, showing that the GP framework is able to perform properly on a single rule. Generation times, however, were highly variable – sometimes runs would complete with an exact copy of the function within a few dozen generations, in other cases it would take several hundred (about 30% of the time), or converge without finding anything good at all (about 10% of the time). Refinement of the parameters for the specific problem would be needed to achieve better results, but for testing purposes the results were sufficient to be deemed acceptable, and for further testing to proceed.

The next step is to ensure the GP framework can operate properly on groups of rules collected into states, and groups of states collected into programs. Again, the problem of symbolic regression was chosen, and the desired end results the same: $5x+4$, $-5x+4$, and $5x^2+4$. The difference, however, is that the GP framework will have a three-state machine to work with, rather than just a single line of code. The states will be limited to 20 rules each. This is obviously less efficient than using a single command, and it is possible that the additional chaff of having to use three states (though only one will likely be accessed by optimal programs) will cause the GP runs to fail. The optimal result in all of these cases is a single initial state containing the relevant rule, two states with nonrelevant output in them (or nothing at all), and the single rule in the first state being an exact replication of the desired function.

The regression progressed effectively, though much less so than using a single rule. This is to be expected, as a much larger search space means mutations can be further spread out, causing the same result to be achieved much more slowly. The reasoning behind this seems to be that because of the

larger genetic representation of the same solution, and the allowance of genetic elements not allowed in the previous experiment, many more neutral mutations (such as illegal next-state operations, math output commands, and joystick commands, which act as no-ops) will accumulate rather than mutate away into beneficial ones in a short time. By changing the "MathFitness" function used in the test to add a penalty function for the program length, a single monolithic state was more likely to have the highest fitness, because the optimal solution is a single command. The symbolic regression was optimized by removing the neutral operations such as joystick commands (allowing only variable assignments) and adding a penalty function for total program length. The modified system still was not very effective at symbolic regression, but the representation was absolutely terrible for the job given the constraints it was given as part of the test. The test was deemed passed, since a fairly steady increase in fitness could still be observed and at some generation point satisfactory results were obtained for each test. The first test, with a single math command, proves that with a more suitable representation, good results can be fairly easily obtained. The issue in this case was simply the size of the search space being expanded by adding a maximum of 3 allowable states with up to 20 rules, instead of one rule, a sixtyfold increase in possible programs.

In all of the above tests, the results proved superior to a random set of 2000 individuals generated using the same means as the initial populations. This shows clear evidence that evolution is occurring and the genetic programming framework for SEBeLS is effective.

Upon completion of these tests, the implementation of the framework was deemed sound, and construction of a task set for testing the Antbot could begin.

5.6.4. Establishing Parameters

The next phase of testing was done to establish parameters. Runs were conducted to quickly determine an effective parameter set for the mutation rate, crossover rate, and tournament selection parameters p and k. There were two sets of these runs: one set with a population size of 50 and 500 generations of execution, and the other with a population set of 2000 and 50 generations of execution. Values were tested from 0 to 1 in increments of 0.1 near the middle, with increments of 0.05 being used when values became less than 0.2 or more than 0.8. For k, values ranging from 2 to 9 were tested. The problem to be solved was the same wall following problem, but using only the straight corridor simulation, without the annulus – again, this was to improve the speed of the simulation so that good parameters could be quickly retrieved. Each run was repeated three times to reduce the possible margin of error.

In the end, the best parameters established were a mutation rate of 0.5, a crossover rate of 0.05, a value of 0.5 for p, and 5 for k. These yielded the highest average fitness out of all of the test runs. The other parameters, such as probability for mutation at each level, were determined arbitrarily, due to their lower impact than these main parameters, and the amount of time it would take to test all of these possible combinations with any degree of granularity.

5.6.5. Future Implementation Work

The Python implementation of the framework has reached the end of its usefulness due to a lack of speed and portability to multiple-core systems. Upon proof of concept, a port of the implementation to a lower-level, more optimal language such as C or C++ will be developed. Though immediate portability is lost, the language chosen will be one whose compilers are widely available on different architectures and operating systems, so experiments can simply be compiled from source on every new machine. The greatest advantage gained by using such a language is the ability to optimize for multiple-core systems to operate on a single run, improving speed and allowing for larger populations and more complex problems to be solved. The possibility of using distributed computing to further improve experiments also exists if a new language is chosen.

Additionally, the issue of simulation noise is one for future implementation. Currently, the simulation has been chosen to be noiseless, due to the fact that the language is new and the robustness of the GP framework are beyond the scope of initial research. Further work will be done on simulation noise in the SEBeLS GP framework.

6. Experiment Design

In this section we experiment with various GP techniques to explore improvements to the framework in application to the SEBeLS language. These improvements will hopefully allow for more useful programs to be created, eventually leading to functional AI programs being developed purely by genetic methods for various real-world applications. These techniques will be applied to the wall-following fitness function described in the previous section, beginning with a baseline of a "standard" genetic algorithm, followed by the addition of single techniques described in section 3. Afterwards, these techniques will be used in combination with one another to create more useful results.

6.1. Experiment Structure

6.1.1 Initial Baseline

The first task was optimizing the basic crossover/mutation based framework: that is, finding a set of mutation operators that works nicely, a crossover method that applies well to SEBeLS code, and a selection method and set of parameters that takes well to it. This gave starting points for the various selection and recombination parameters. The reasonably best parameter set found during the tests in section 5.6 was a mutation rate of 0.5, a crossover rate of 0.05, and a population size of 2000. In all of the experiments performed, the initial population is generated randomly by using the add-random-new-node mutation function a random number of times. The selection method chosen for this baseline run was k-tournament selection, with k equal to 5 and p equal to 0.5. A larger population would contribute significantly to the results, but the speed of execution would become prohibitively slow. This compromise allows more experiments to be run and a better view of the possible ways of obtaining a solution to be seen.

Each of the experiments shown below will be repeated for five runs, stopping after 500 generations, with the following parameters:

Population Size	2000
Mutation Rate	0.5
Crossover Rate	0.05
Selection Method	Tournament Selection (k=5, p=0.5)
Stopping Condition	500 Generations Evolved

Table 1: Basic GP Parameters

Additionally, the crossover operator described in 5.4.1. uses the following raw probabilities to determine at which level of the program crossover will occur:

Level	Probability
States	0.1
Rules	0.15
Commands	0.25
Nodes	0.5

Table 2: Crossover parameters

The mutation operator uses a similar parameter set, along with the description of which mutation operators are allowed at each level. The raw probability of each different action occurring at a given level is equal. Again, these probabilities are the raw probabilities, and not the cumulative values used in algorithms 10-13.

Level	Probability	Possible Actions
Program	0.05	Add state, delete state
State	0.1	Add rule, delete rule
Rule	0.19	Toggle use of conditions, add command, delete command
Command	0.18	Swap subtree parents, change type, insert node
Node	0.53	Change node value by 1, change node value randomly

Table 3: Mutation parameters

Two simulations were chosen for the experiments: a straight track, and a circular one. The raw fitness of the simulation is set to the product of the two simulation fitnesses, to avoid overfitting. The following parameters set the properties of these tracks:

Width of Tracks	40 units
Starting heading of robot (straight track)	60 degrees
Starting heading of robot (circular track)	90 degrees
Heading Angle of Straight Track	91.1 degrees
Radius of Outer Circle	80 units
Radius of Inner Circle	40 units
Speed of Robot per "turn" (rule parsed)	5 units forward/back, 10 degrees turned
Emergency Stop Distance	5 units
Emergency Stop Penalty	100 Fitness
Checkpoint Value	62.5

Table 4: Simulation parameters

Finally, a subset of the external Antbot commands were chosen. See section 2.5 for a reference of Antbot commands.

Command	**Possible Parameters**
@J	1-9 as normal, but 1,3,5,7, and 9 set to Stop
@F	X, Y, Z, A, B, C
@E	Math Operators (only post-condition), Boolean Operators, Variables from @F, Lower Case Variables, Constants
Math Operators	Negation, absolute value, sine, cosine, square sine, square cosine, angle format, multiply, divide, reverse divide, add, subtract, reverse subtract, deadzone, clamp, modulo division, angular subtraction
Boolean Operators	Boolean negation, greater than, less than, equality, multiplication, addition
Constant	-180 to +180
Lower Case (read-only) Variables	g, h, i, j
@N	01/03/09

Table 5: Antbot commands

The reason additional stops were used instead of additional movement commands was twofold: first, to allow for mutation to create temporary stops on a similar likelihood to movement commands, and second, to simplify the simulation and avoid the need for the higher precision required by the circular paths created by these commands.. Though it reduces the performance of the wall follower in 2.4.4, the performance of good individuals generated by GP will not be affected by this decision, as the

subset of commands chosen is available on the physical Antbot. During testing with an Antbot, it was seen that disabling these commands in the firmware led at worst to more "hunting" behavior by the robot while navigating, and higher power usage.

The baseline of traditional genetic programming, as defined in chapter 3, were used as a comparison point for further experiments using the techniques described below. Five runs were executed.

6.1.2. Elitism

The elitism method was tested by adding provisions for an elite population of 20%, or 400 individuals. This sizeable preserved population would ensure that the majority of high-fitness individuals will reproduce at each generation, rather than having a random chance that few (if any) of them, would. By retaining more high-fitness individuals, the overall fitness of the population should increase faster. That said, destroying more low-fitness individuals would likely lead to high-fitness individuals reproducing more than once per generation, reducing diversity within the population.

6.1.3. Multiple Distributed Populations

Adding distributed populations to the basic method should introduce additional diversity into the population. Five sub-populations (each of size 400) were chosen as an initial test – it is likely this number is far from optimal, but it was hoped some indication of whether or not this method is effective would still show. It was expected that the diversity of the population would increase, causing more rapid changes in fitness (for better or for worse). The migration method used for this experiment is a direct swap of two individuals of two separate populations, with an equal chance of any pair being selected over any other. The number of individuals swapped at each generation is random, between 0 and 10% of the total population size. This is larger than some of the chosen values in [14], but in the paper, the authors state due to low migration in some tests there may be random effects from the initial population, therefore additional migration was made possible. The average case of migration is, given the normal distribution of the random number generator, 5% of the population per generation.

6.1.4. Hi-Lo Fit

This selection method is designed to not only prevent cloning, but to prevent multiple similar individuals from ever reproducing at all. In doing this, randomness is increased while maintaining the exploitation of high fitness (using this selection method as explained in section 3 implies something similar to 50% elitism). Additionally, once the population has been initially sorted, this selection method is computationally faster than tournament selection by a significant margin, allowing for the faster accumulation of results.

6.1.5. Module Acquisition

The use of modules is expected to improve the overall ability of programs to evolve, by transforming several rules within a state into a module, which can be crossed over between programs as a single rule would. The intent here is to allow a group of rules that increase a program's fitness greatly to propagate as a whole rather than as single rules, as often as possible. This should increase overall fitness and decrease the number of generations between significant fitness increases. In this section of the experiment, the possibility of module acquisition will be added to the framework as groups of rules. Modules will have a probability of 5% per mutation at the state level of being created. The modules themselves, when chosen to be mutated at the rule level (since they act as groups of rules) will mutate as a state, with the possibility of adding or deleting new rules, or mutating an existing rule within the module. Mutation will propagate to smaller structures as described in section 5.4.2.

6.1.6. Combining Techniques

After all of the techniques described in this section have been tested individually, further experiments were run on combinations of these techniques that appear useful or promising. If possible, the majority or all of these combinations of at least two techniques will be tested – combinations of three or more proved too numerous for the computing resources available, and will be left for future work.

6.1.7 Physical Testing

For each run, the programs were physically tested on a real Antbot in a confined environment similar, but not identical to the simulation environments. The test environment is a straight corridor 150cm wide, and long enough that the Antbot will stop (or be manually stopped) before the ends are within sonar detection range. The results of these tests will be recorded in chapter 7 along with the rest of the results.

6.2. INCREMENTAL OBSERVATIONS

6.2.1. Proof of Concept

When first testing just the straight-corridor simulator, a few exceptional individuals that used the sonar sensors appeared. This was somewhat surprising, as it was expected that only overfit individuals would have high fitness in this simulation (individuals that do not respond to stimulus but instead "memorize" the course and move by dead reckoning). Many such individuals occurred, but there were several runs where the best individual was much more sophisticated, such as the following (note that the initial value of B is nonzero):

```
1       STA 0
2       @?0.0 ; @FX A | (,@?j,@J4,@?-44.0 ]
        This line does nothing
3       @?150.0 0.0 0.0 + - ] ; @N1,@?-140.0 | 16.0 -,@J1,@FX -126.0,@J4
        Always start turning counter-clockwise. Set X to -126.0
4       @?123.0 ; @?39.0,@FZ -89.0,@J2
        Always start reversing, set Z to -89.0.
5       @?B ~ ; @N1,@J2
        Go to state 1 if B is 0.
6       @?j 36.0 < ; @J6,@N0
        If right rear sonar is less than 36, turn clockwise.

Next state will always be 0.


7       STA 1
        This state never executes.
8       @?Y ; @J5,@N0


9       STA 2
        This state never executes
10      @?126.0 71.0 > ~ ; @FB -89.0 51.0 -,@N2
11      @?X ~ ; @FA Z,@J4,@N1,@J6


12      STA 3
        This state never executes
13      @?88.0 ~ ; @N2,@J8,@N2
14      @?35.0 | | ; @?h
15      @?98.0 ~ ~ ~ ; @?-142.0 91.0 {
```

The fitness of this individual is 7622.88. The fitness appears low for such an outstanding individual, but because of the backward start, it will earn a low fitness on the clockwise circular track, and likely hit a wall before encountering any checkpoints. By reversing the orientation of the Antbot, and testing the same program, fitness increases considerably, to 775410.73. Because of this major increase in fitness, the importance of a good choice of environments to obtain generality rather is clear. In some situations, even a slight change in an environmental parameter can severely change the performance of an individual.

In this simple program, consisting only of one simple state, a simple navigation system, with the robot moving in a zigzag pattern in the reverse direction, is run. The Antbot will turn counter-clockwise briefly, reverse, then, if the right rear sonar sees a wall that is too close, it will turn the Antbot away from said wall. It may overcompensate, but at every loop iteration, the Antbot turns back counter-clockwise, repeating this loop: this represents a full implementation of a wall-following "mouse" toy robot, using a sonar instead of the more common switch.

6.2.2. Optimization of Performance

Later, when the annulus simulation was added, the GP framework's performance became a much more significant concern; in the amount of real time it previously took to run 500 generations, only about 200 could be simulated. The circle course was significantly more computationally complex, and needed to be optimized. The circle course involves much more trigonometry than the straight corridor, and requires multiple possible intersection points to be calculated. In the end, to reduce processing time the entire sonar ranging simulation was implemented by way of a lookup table – the granularity of 0.1 unit for position, and 0.1 degree for distance is acceptable considering the non-ideality in the performance of both sonar sensors and motors in a real Antbot. The level of granularity used for the commercial implementation of the NAVCOM system when performing calculations on headings and other angles is 0.1 degrees, while 0.1 units in the simulation would correspond approximately to 0.15 centimeters (the typical hobbyist-grade rangefinder is accurate to about 2 to 4 centimeters [49]).

Populations of 1000 individuals, tested briefly to allow more runs to be done within the time frame, produced results that with greater variance and difficult to measure the quality of: populations of 2000 were therefore retained. A single experiment thus constituted of 5 runs of 500 generations on a population of 2000, and results gathered from these as outlined in the previous section.

During these experiments it quickly became apparent that there were significant speed differences in the various methods being tested – most interestingly, the Hi-Lo Fit selection method produced runs much faster than tournament selection despite the sorting requirement; 500-generation runs would take, with the computers described in 6.2.3, approximately 60 hours, from about 72 for tournament selection. Similar speed ratios appeared on faster computers. Additionally, using modules proved to be much slower, though in some cases their use gave a considerable increase in average fitness (see chapter 7).

6.2.3. Crowd-Sourcing Genetic Programming

One unrelated but very interesting observation, both about the performance of the framework in Python and about the power of the Internet, was how the experiments ended up actually being run. Originally the project was provided with CPU time on a PowerPC cluster, as well as several desktop computers. None of these machines, even with Psyco on the x86 machines, performed adequately to obtain any results at all – they crashed, suffered hardware failures, and were unable to complete any runs. The PowerPC clusters were unable to run Psyco and had poorly-optimized Python binaries, and would complete a run in about twenty to thirty days. A few machines provided by us, were able to make some headway and complete a few runs, but it was questionable at that point whether even the combinations in section 6.1.7. would begin before time ran out. The problem was alleviated by resorting to crowdsourcing: a post asking for help on the website f3.to [13] resulted in new computers becoming available to the project, often for periods of several weeks at a time, and many additional runs could be completed in a much shorter period of time. Unlike larger-scale projects such as Seti@Home [59], the experimental programs were distributed as stand-alone packages with no online capabilities, and a web page with an uploader was set up for users to manually submit the resulting log files.This simple "honor system" crowdsourcing method proved to be effective enough to generate usable results.

Not all of the specifications of every computer being used were made available by the users during the test. The minimal specifications of a computer successfully finishing a run were as follows:

CPU: Intel Pentium 4, 1.4 GHz

RAM: 1 GB

Disk Space: 40 GB

Additionally, all of Windows, Linux, and Macintosh OSX were used by at least one user at one point. At any given time the number of computers doing GP runs ranged from 1 to a maximum of 8, though the number of concurrent runs only has bearing on memory usage, not the experiment itself. The program was distributed in the form of self-contained executables generated with py2exe to run on a x86 architecture; volunteers would allow the program to run to completion and manually send the log file back for analysis.

6.2.4. Combinations of Techniques

Upon observing the performance of the individually-run GP techniques, we chose pair-wise combinations of the techniques to test next. The following tests were then scheduled, prioritized by potential utility (based on the best individuals seen in the single-technique experiments):

- Hi-Lo Fit with Elitism
- Elitism with Multiple Distributed Populations
- Hi-Lo Fit with Multiple Distributed Populations
- Modules with Hi-Lo Fit
- Modules with Multiple Distributed Populations
- Modules with Elitism

All of these techniques were again run a minimum of five times; some arbitrarily received more, due to the lack of exact regulation described in 6.2.3.

6.2.5. Classical Parameters

As shown in testing, an unusual parameter set had been chosen for the experiments, with a very high mutation rate and little crossover. Because of this, a "classical" parameter set for mutation and crossover was selected and some of the experiments were repeated. The experiments chosen were the baseline and distributed populations as the two most affected by the selection pressure, and the combinations of hi-lo fit and module acquisition, and hi-lo fit and elitism, which were high-performing combinations (see chapter 7) that acted as a control. The results of these experiments show whether the low selection pressure and high mutation was a significant negative effect on previous results, both in cases with low average fitness, and high average fitness.

The final parameter set was:

Mutation Rate	0.05
Crossover Rate	0.9

Table 6: Classical GP Parameters [28]

Otherwise, parameters were as defined in tables 1 through 5.

6.2.6. Expectations and Hypothesis

Upon seeing the individual in 6.2.1. showing a potential high fitness of around 750 thousand, it is apparent that a good measure of the effectiveness of a technique is how many "strong" individuals such as this it can produce. The threshold of such individuals was approximately 700 thousand, because the individual shown in 6.2.1., when reversed, is a weak corridor-follower, and there is no likely point observing anything less effective. It is expected that with the parameters found during the testing of the framework, that each technique has at least sufficient probability to yield a strong individual that at least one should be observed within five runs. Given the high mutation rate, it is expected that techniques with additional selection pressure, such as elitism, will yield the best results. It was also expected that the parameter set from testing will outperform the traditional parameter set added to the experiment plan in 6.2.5. These expectations, as shown in chapter 7, were met by the test results.

7. Results

In this section, the results of the various experimental runs described in chapter 6 are explained and analysed. For each run in each experiment, the best fitness value obtained, along with the number of distinct individuals obtained at the end of the experiment will be given, along with average values for each experiment. Additionally, the best individual from each run will be shown, and its program explained in detail. Afterward, the most functionally interesting individuals will be shown, and their programs explained in further detail, and simplified by removing non-operating code so that they may be run on a functional Antbot. Finally, these individuals will be tested on Antbots in a physical environment, to determine how effective the simulated code is in the real world.

Note that in these runs, none of the upper-case variables are initialized to any value. Only g, h, i, and j, the sonar values, are used for anything other than scratch variables. These values, as explained previously, contain real-time sonar data from the rangefinders. Since the other variables are uninitialized, their values are nondeterministic at startup, and may contain junk memory data. In the simulations they nearly always evaluate to true, or if used as an integer, to a random value until modified. There is a chance, however, that they will initialize to exactly 0.0, causing them to be evaluated as false.

7.1. Raw Results

This section contains the results of each GP experiment, beginning with traditional genetic programming as a baseline, and progressing through the various techniques and combinations in the order presented in 6.2.4. The parameters used for each run, including traditional GP, are given in 6.1.1. For each run, the best fitness achieved, the generation at which this best fitness was achieved, as well as the final number of distinct individuals, will be presented. The average best individual, average number of distinct individuals, and a graph of the best individual at each generation for each run will then be plotted, each colour representing a run. Finally, some observations on the best individual reached will be given, as well as the results of a test of that individual on a physical Antbot.

7.1.1. Traditional Genetic Programming

This experiment was run to obtain a baseline for comparing the other results. No additional genetic programming techniques were added onto the basic algorithm shown in section 3.1.

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	688733*	252	1519
1	628213*	455	1496
2	270848*	450	1536
3	300026*	260	1524
4	542727*	268	1506
5	568615*	212	1542
6	1022742*	417	1559
7	629906	154	1532
8	1381344*	341	1540
9	769451*	145	1514

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 680260

Average Distinct Individuals at Generation 500: 1526.8

Fitness Graph:

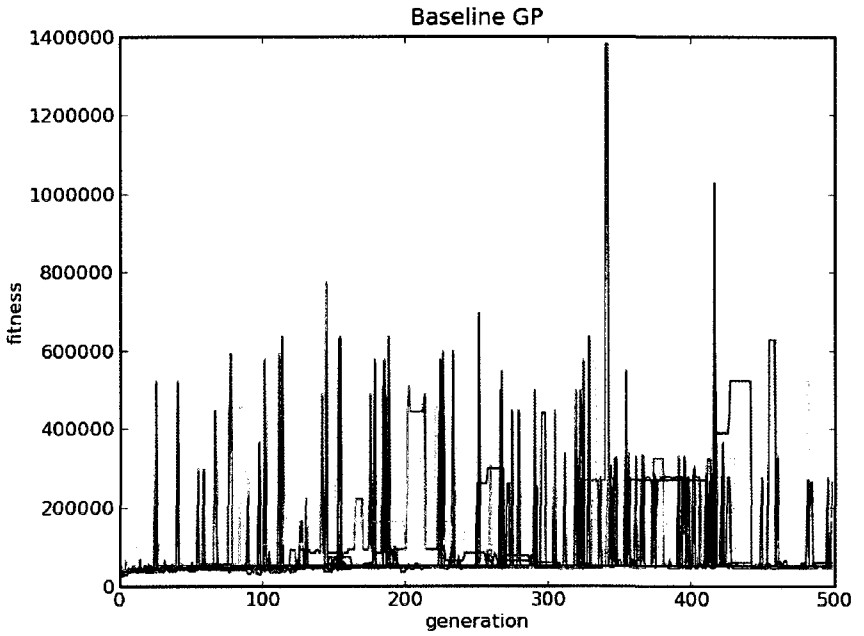

Figure 22: Results of Traditional Genetic Programming

Best Individual:

```
STA 0
@?0.0 ; @N1,@FC 43.0,@J4,@?C
@?156.0 ] A } X [ * C 61.0 ~ } ) ~ ; @FB g,@N4,@?59.0 ) |,@FC j -143.0 *,@FY Z
@?170.0 ; @J2,@J4,@?j -6.0 ) },@J8,@?150.0 X ` /
@?172.0 g + ; @FY Z g /,@?-162.0 |,@N2,@?j [
@?Y ; @J2,@J6,@J6,@FC 142.0 A | -130.0 - *
```

Fitness of 1381344, obtained in Run 8, Generation 340.

The first line is non-executing, the rest seem to be fixed to always execute or not execute regardless of variables. The last line is dependent on the initial value of Z – if Z is zero, the last line will execute and cause the robot to turn clockwise. It does not appear to use the rangefinders in any effective manner, and must therefore be overfit to one or both courses (likely the straight course).

Observations: In most cases, the high-fitness individual discovered was lost by the next generation. Selection pressure was likely not high enough despite significant testing showing otherwise early on. The best individuals seemed to be more likely anomalies caused by random mutation than any steady process (see figure 22). A new technique would need to be introduced, but additionally, increasing the crossover rate, and increasing the selection pressure from tournament selection, would benefit the effectiveness of this experiment. It did produce three strong individuals out of ten, but one of these was very close to the threshold of fitness.

Physical Antbot Tests: Based on repeated simulations of this individual, it was determined that the initial value of Z was some value between 38 and 40. In a straight corridor, the Antbot will turn, then progress forward down the corridor, making slight course corrections. It appears that the individual is overfit to this course, as when changing the initial heading from 60 degrees to the corridor (as in the simulation), the Antbot will crash into the first wall it encounters.

7.1.2. Elitism

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	121867*	492	1306
1	736838*	354	1370
2	1079230*	493	1262
3	793319	249	1239
4	740696	286	1324
5	686089	418	1328
6	737271	319	1297
7	99592	243	1363
8	706570	499	1272
9	810724	496	1209

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 633496

Average distinct individuals at generation 500: 1297

Fitness Graph:

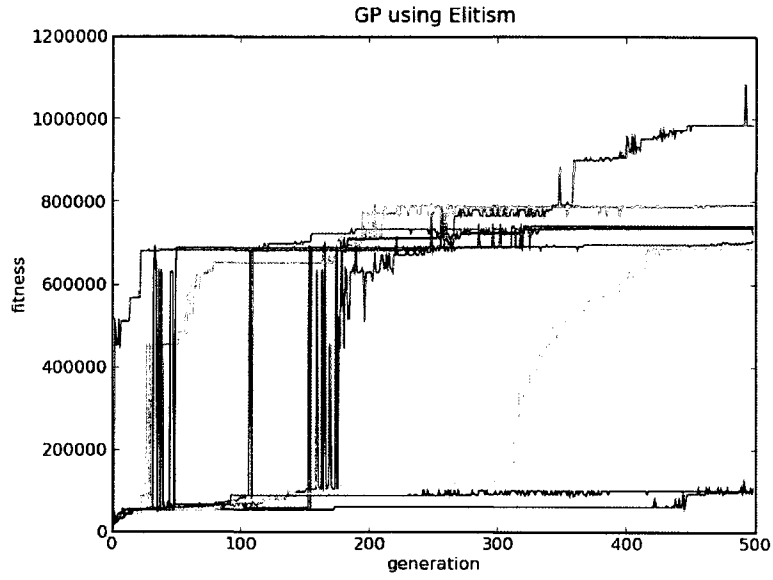

Figure 23: Results of Genetic Programming using Elitism

Best Individual:

```
STA 0
@?Y ; @?-162.0,@J2,@?85.0 C ~ -31.0 i - % `,@FY Y h } -169.0 ~,@J2
@?j 0.0 g ~ ~ ~ * < ; @FA 120.0 41.0 \
@?0.0 ~ ~ ; @J7,@?-74.0,@?143.0 ! X },@N0
@?131.0 ~ 9.0 + ; @J4,@N2,@FY i
@?0.0 ; @FA g,@J3,@J2,@N0,@J4
@?B ; @J3,@N2,@J8,@FY Y 122.0 110.0 / -82.0 ] h } Y 122.0 110.0 / -82.0 ] h } C | _
      ~ 38.0 125.0 ~ ] - ! ) ) \ | ! | _ ~ 38.0 125.0 ~ ] - ! ) ) \ | !
@?h ; @?-65.0 [,@N0
@?X 0.0 > ~ ; @FZ 74.0,@?C Z ] | - 122.0 [ B * +,@FA 126.0,@N2,@J4
@?Y ~ ~ ; @FC 46.0
@?83.0 ~ ~ ; @?54.0
@?X Y ~ i ~ ~ == 101.0 < ~ > ; @N1,@FZ -179.0
@?g ; @?B,@FZ C ) ( 97.0 {,@N1,@?17.0 `,@N2
@?X ! A 140.0 % 0.0 { g \ ) % 144.0 ~ ] ; @N2
@?164.0 5.0 159.0 * + ; @J8,@N2,@?100.0 ] ! -109.0 A | ( { ] h ! -164.0 + + [ *
      [ 122.0 +,@N2,@?- 25.0
@?Y ~ ; @?h (,@FX i,@J7,@N1
```

```
@?33.0 ) | ; @J6,@?112.0 ],@?36.0,@FZ -92.0 | (,@FY 127.0 ] ! ]
@?0.0 0.0 119.0 + + 0.0 119.0 + + 0.0 * ; @N0,@N1

STA 1
@?B ; @?i [,@FZ Z [ ),@?h g ( ] ~ B },@?i
```

Fitness of 1079230, found at generation 493 of Run 2

This program is another example of an overfit individual, whose sonar inputs, and scratch variables where expressions containing sonar inputs are evaluated, never correspond to joystick commands – they simply execute on their own. State 1 is never reached, and state 0 simply loops to follow a cycle of joystick events that navigate both the linear and circular tracks somewhat effectively, likely earning a high fitness on one track, and low fitness on another, by means of concentric circles fit to the width of the track and starting heading. These commands are executed on the first, fifth, and eighth lines of state 0 – none of the rest of the code is relevant.

Observations: Elitism clearly produced sufficient selection pressure to achieve high fitness, with an average fitness considerably higher than the baseline. Fitness was a steady progression from the initial population towards generation 500 (see figure 23) with little to no loss of good individuals, and the best individual was always achieved relatively late. Four individuals were generated over the fitness threshold, though the highest fit of them was a clearly overfit individual. Some of the others behaved similarly, while others demonstrated true navigation behaviour like in section 6.2.1.

Physical Antbot Test: Starting pointed at 60 degrees, as in the simulation, the Antbot will navigate the corridor for some time before encountering the left wall and emergency stopping. The motion of the Antbot is interesting – it drives in reverse, turns 180 degrees, then drives forward, turning around again to drive briefly in reverse, and repeating this pattern. Since it does not turn perfectly precisely, the controller eventually fails. In all other cases, the controller simply collides with the first wall it encounters – it has no obstacle avoidance capability.

7.1.3. Multiple Distributed Populations

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	1172629*	91	1509
1	420850*	372	1494
2	65317*	357	1510
3	683015*	156	1543
4	680501*	53	1518
5	61612*	81	1500
6	1212049*	467	1543
7	300026*	121	1525
8	682554*	449	1511
9	69202*	250	1509

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 524775

Average distinct individuals at generation 500: 1516.2

Fitness Graph:

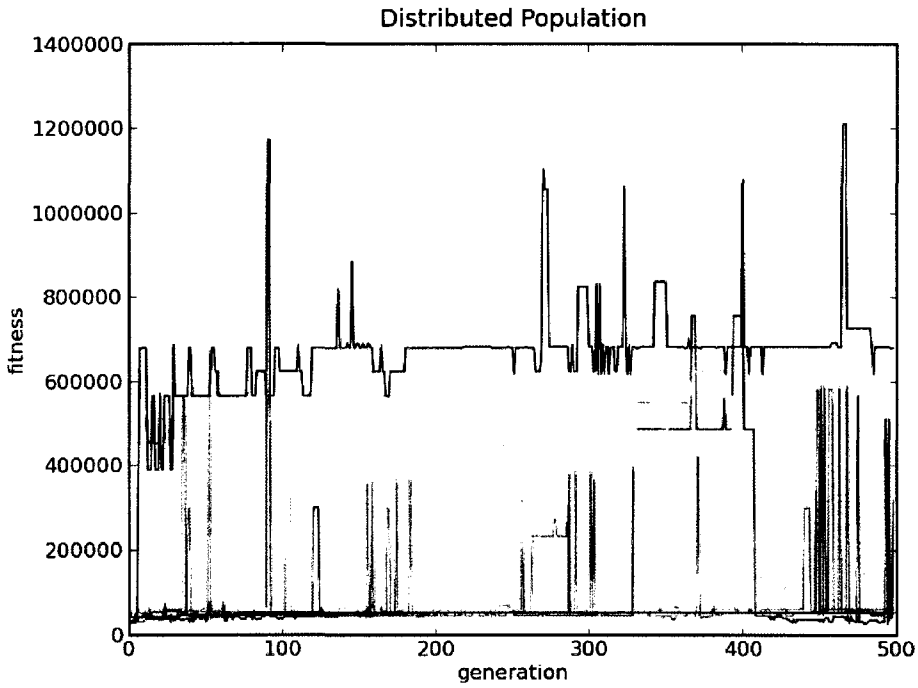

Figure 24: Results of GP using Distributed Populations

Best Individual:

```
STA 0
@?58.0 0.0 0.0 * < ; @FB Y,@J6,@J2,@FX i,@FA -43.0
@?C ; @J6,@J6
@?Y h > ; @?i,@FA g ` `,@?A -110.0 - Y / [ !,@J4
@?j 58.0 ~ 127.0 + * ~ ; @FA 20.0,@FC h 4.0 ! ` | Y * *
@?126.0 168.0 + j + ; @J5,@FB i,@J2,@J8
@?0.0 0.0 * 0.0 + ; @FA -78.0,@FX A
@?B g * ~ ; @N0,@FC Y Z ] ( C i C | { * + ( + 113.0 },@N1,@FZ A
```

Fitness of 1212049, found at generation 467, Run 6

This individual will do nothing but drive forward, and on the second and third lines, turn based on its sonar values of g, i, and h. Its operation is quite confusing, using a large amount of self-recursive math on the variable C, and is in fact quite an interesting individual to earn such a high fitness.

Observations: Selection pressure appears slightly weak, with some information being lost to mutation or under-selection, as shown by the graph in figure 24. Though only twice was the threshold of a "good" individual broken, many of the individuals were very close to that point, with a fitness of around 680 thousand being common. Given the effectiveness of the individuals being generated, this is clear evidence that the use of genetic programming to program simple robotic navigation is promising using SEBeLS.

Physical Antbot Test: With a starting angle of 60 degrees, as in the simulation, the Antbot does not avoid the right wall before emergency stopping. When facing the left wall instead, it will avoid it, then collide with the right wall – the Antbot clearly has no ability to recognize or avoid obstacles to its right, being overfit to the circular track.

7.1.4. Hi-Lo Fit

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	909145	482	1324
1	218710	495	1134
2	80027*	429	1313
3	656717*	320	1271
4	776504*	325	1190

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 921020

Average distinct individuals at generation 500: 1246.4

Fitness Graph:

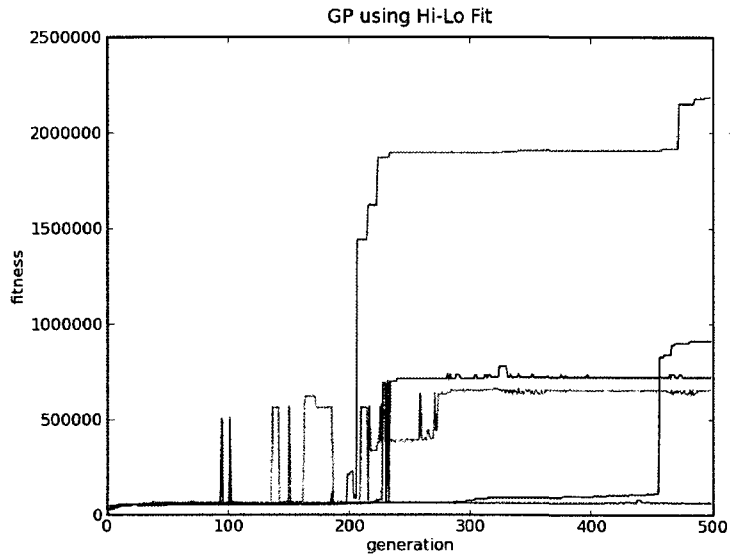

Figure 25: Genetic programming using Hi-Lo Fit

Best Individual:

```
STA 0
@?j 0.0 176.0 ~ h ~ + 107.0 * * + ; @J8,@?-148.0,@FB -29.0,@N2,@?C


STA 1
@?0.0 ! ; @?1
@?100.0 35.0 g < * ; @N2,@N1,@J4,@N0,@?-58.0 Y _
@?C ~ 10.0 < ~ h ~ ~ ~ B 5.0 * ~ < ~ ~ ~ B ~ X > > ~ ~ Z ~ ~ ~ < < ~ ; @?85.0
      |,@N2,@N0,@N0
@?0.0 ; @N0,@N0,@?A h _ 78.0 _
@?0.0 g * ; @FX 151.0 |,@FA -29.0,@J2


STA 2
@?0.0 ~ A == ; @J3,@N2,@?Y 135.0 21.0 { *,@?-17.0
@?100.0 h g < * ; @N1,@?Z B ) -167.0 _ -133.0 ! \ X j * { 143.0 ) \
@?46.0 ; @FB 105.0,@FY A !
@?C 0.0 * ) ; @N0,@?Y 63.0 -144.0 -25.0 } -47.0 - ` +,@FC -88.0 ),@FX -94.0 161.0 }
      ],@J6


STA 3
@?0.0 ; @N2,@FX h
@?j 0.0 93.0 * 176.0 ~ h ~ + 107.0 * * + ; @?153.0 173.0 ` - !
```

Fitness of 2182710, found at generation 495, run 1.

This individual demonstrates the use of multiple states and good decision making. In state 0, the robot will drive forward and go to state 2. In state 2, depending on the difference between g and h, the robot will (or will not) go to state 1 – if not, it will return to state 0. In state 1, the robot can be made to turn left by the second line, and/or return to state 0 on the third line, which always executes. State 3 is never called. It is a simple method of navigation based on sonar positions and going forward while turning left, but it appears at least somewhat effective.

Observations: Overall, hi-lo fit produced a good set of individuals, with three over the threshold, and very little loss of good individuals due to selection pressure. Out of the three cases where the peak fitness at generation 500 was lower than the peak overall (see figure 25), the differences were always less than 20 thousand, unlike with traditional GP. The best individual found is simplistic but demonstrates a clear ability to navigate based on a relation between the values of some sonars, like in the example wall-follower in chapter 2.

Physical Antbot Test: The Antbot spends most of its time driving towards the left and right walls, overcompensating when it reaches them, and making slow forward progress. It never collides with the walls, however, and if it meets a wall head-on, it will reverse, then turn, rather than turn immediately. It clearly shines.

7.1.5. Module Acquisition

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	673224*	426	1441
1	561311	495	1456
2	327898*	232	1380
3	1304358*	27	1458
4	453786*	17	1457

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 664115

Average distinct individuals at gen 500: 1438.4

Fitness Graph:

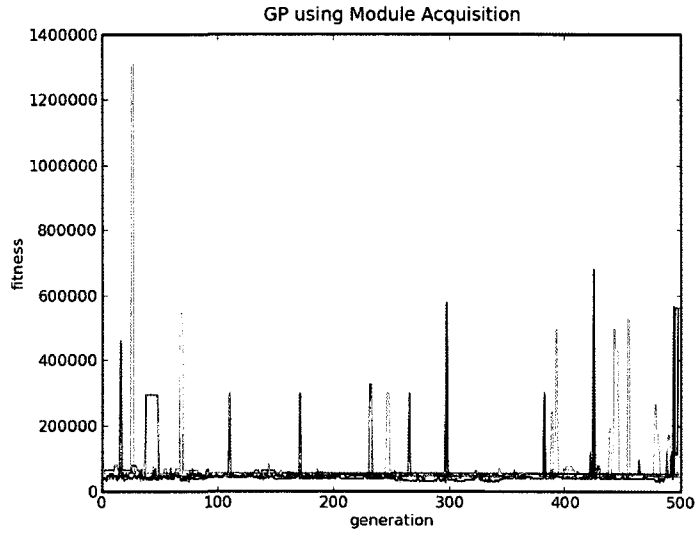

Figure 26: Genetic programming using module acquisition

Best Individual:

```
STA 0
@?122.0 ~ ~ 129.0 + ; @J2,@N6
@?108.0 ; @J5,@FX -29.0,@FC C ),@N1,@J4
@?j 0.0 + ; @J8,@N1,@FY Z,@J2,@?178.0 !
@?0.0 ; @FX i Y / g \ Z _,@N2,@J1


STA 1
@?0.0 X 0.0 ~ 155.0 * ~ < A < ~ 0.0 71.0 + < > ~ ~ ; @N0,@FC h )
@?X ; @FA h (,@J8
@?67.0 ; @FB A,@?X,@N1,@N2,@N0


STA 2
@?A 0.0 ` - A - [ ; @N2,@?j,@N1,@N0
@?78.0 ! ; @?-125.0 |
@?j i > ~ ; @N0,@J8
@?119.0 ~ ; @?Y ) ( 118.0 +
```

Fitness of 1304358, found at generation 27, run 3.

This individual progresses in a mostly backward direction, moving between states 0 and 1 while making alternating left and right turns at arbitrary points. It is a clearly overfit individual to the straight track, though it obviously achieves a fitness of at least 325 on the circular track. This is likely caused by the first line of state 1, after the first loop through – with A initialized to the value of h several observations ago, it will be able to make limited navigation decisions. In state 2, it shows the ability to make a simple navigation decision on the third line, that results in not only a state change, but a change in direction.

Observations: A poor set of runs with only one strong individual that appeared early in a run, clearly through random mutation. There is no strong selection pressure to retain good individuals (see figure 27) so no good further evolution occurs. Module Acquisition in combination with the baseline technique does not appear to produce any useful patterns.

Physical Antbot Test: The Antbot emergency stops on the first obstacle. There is no generality in this solution at all.

7.1.6. Hi-Lo Fit combined with Elitism

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	788118	498	879
1	2237330	494	953
2	1615302	89	1060
3	688733	188	989
4	313922	40	888
5	709121	59	929
6	697559	430	1028
7	2050524	215	892

Average Best Fitness: 1137576

Average distinct individuals at gen 500: 952.25

Fitness Graph:

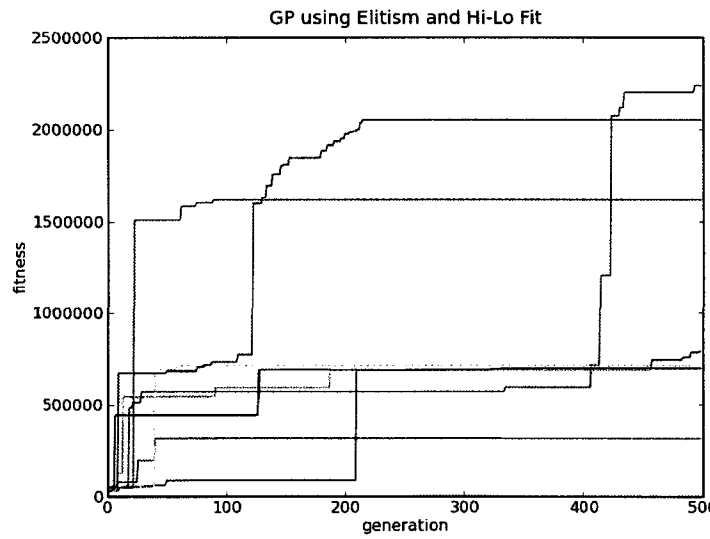

Figure 27: GP using Elitism combined with Hi-Lo Fit

Best Individual:

```
STA 0
@?65.0 ~ Z j + 0.0 ~ < == 61.0 B ~ 0.0 > ~ + 0.0 * * ~ 37.0 g < > ~ ;
      @J6,@J3,@J1,@J4
@?j h < ; @J8,@FY 163.0,@?-33.0,@N1
@?81.0 ; @N0,@N2,@?-2.0 j ~,@N2,@FC 58.0 ] ` -106.0 -33.0 ~ 141.0 - _ A ) +
@?h 47.0 { ; @J7,@N2,@N0,@J6
@?i ; @?X [,@J8,@N0,@FC 92.0
@?85.0 ; @FA -21.0,@N2,@?Z
@?12.0 j < ~ ~ ; @J1,@?B ] C - A j ! -150.0 [ { } \ `,@J4,@J8,@FB -162.0
@?g ~ 0.0 * h 0.0 < > ~ ~ ~ ~ ; @N0


STA 1
@?0.0 ~ ; @N2

STA 2
@?0.0 j + g == 0.0 3.0 ~ > * 64.0 * ; @N2,@?X 128.0 C ~ { B } 84.0 ~ ],@FY
      68.0,@J6,@J3
@?Z ( 34.0 126.0 ( ) ] ] - ` 79.0 ] + ) ) ( ( [ ` ( % 28.0 _ ; @?1.0 58.0 -72.0
      104.0 74.0 \ ~ % } Y \,@FZ 114.0,@N1,@J2
@?B ; @?102.0,@?A ),@N2,@J7,@?-145.0 C Z [ [ 86.0 / ) -148.0 [ \ ( * _ [
@?j ; @FX B ( ` !,@J2,@?Z | 111.0 - ],@J6,@?-105.0
@?11.0 ; @J4,@?C

STA 3
@?116.0 ~ ; @J1,@N1


STA 4
@?0.0 0.0 0.0 < ~ ~ ~ ~ ~ == ~ j ~ ~ ~ ~ * ~ ~ ; @N0,@?i
@?65.0 ~ Z j + 0.0 ~ < == 61.0 B ~ 0.0 > ~ + 0.0 * * ~ 37.0 g < > ~ ; @?h,
      @J8,@J8,@FZ 95.0,@J4
@?B ; @FC -95.0 ],@FZ 164.0 `,@FC 120.0 j - 101.0 ` { 96.0 { (
```

Fitness of 2237330, found at generation 494, Run 1

At a glance, an interesting multi-state individual has emerged with the highest fitness so far. The most interesting comparisons are the direct check if j (right rear sonar) is greater than 12 on the second last line of state 0, and the comparison between j and h (rear right and front right, respectively) on the second line of state 0. Also, note the use of useless evolution code – state 3 has clearly not evolved into anything yet, as it is currently an endless loop of non-executing code (effectively an end state). The state is never called. Note also that State 1 does nothing but transition into State 2; this state is called several times. While State 4 contains some interesting comparisons on sonar data, it is also never called.

Observations: This method has produced outstanding results – enough selection pressure to retain all of the best individuals throughout the process (see figure 27), and multiple individuals with fitness over

one million (in fact two above two million!). Though only half of the individuals evolved exceed the threshold of 750 thousand, many others come close. There is significant merit in applying parameter-optimization techniques to this experiment for further testing of this algorithm.

Physical Antbot Test: The Antbot navigates close to the right wall, following it through the straight course. It is somewhat able to navigate a corner, though the sides and rear of the Antbot occasionally collide with or scrape against the walls.

7.1.7. Elitism combined with Multiple Distributed Populations

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	654596	482	1176
1	61488	424	1045
2	688733*	130	1304
3	776382*	364	1138
4	620705*	378	1291

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 560380

Average distinct individuals at generation 500: 1190.8

Fitness Graph:

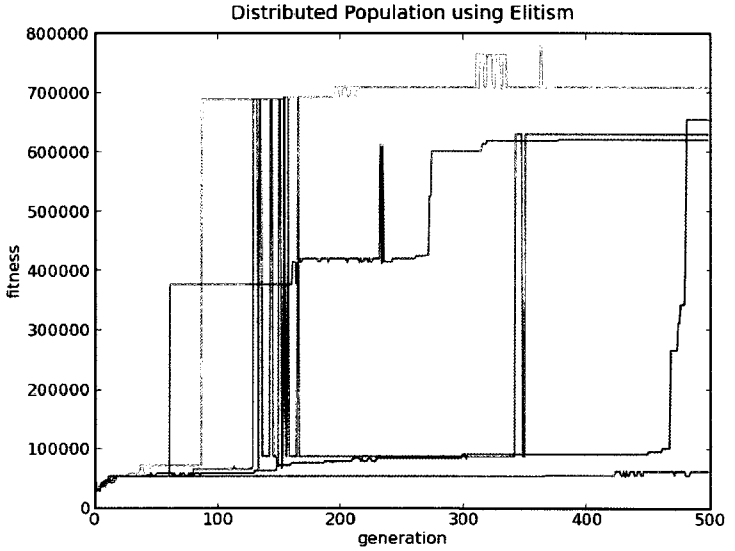

Figure 28: Distributed Population using Elitism

Best Individual:

```
STA 0
@?Y ; @?-31.0 `,@J4
@?C ; @?-174.0 g ) j } ~,@J2,@J8
@?0.0 0.0 > ~ ; @FY A
@?g ; @?-46.0 ( ` h | ] -
@?0.0 ; @?0,@FX -155.0,@N2,@J7
@?h ; @FC g ),@?-79.0 -50.0 \,@?-134.0 64.0 21.0 _ -50.0 -52.0 \ * / ] (,@?h,@FA B
@?Y 150.0 + ; @FB -137.0 -73.0 + Z { 36.0 [ ` / (
@?Z h ~ * 162.0 > 0.0 Y ~ 0.0 86.0 > ~ < == i 0.0 C ~ == Z < 0.0 ~ > * < C Y * >
      < ; @?-163.0 (,@FA i,@J1

STA 1
@?A ; @?139.0 38.0 ~ ] ` -69.0 -118.0 ] ( + [ _,@FC -161.0 `,@FB B h [ ~,@N0
@?0.0 | ; @N1,@FY i Y _ 27.0 /,@J3,@J8
@?0.0 C X < 0.0 + ~ i > * 107.0 * ~ ; @?103.0 -112.0 } B ) 128.0 { X * %,@?h g [
      -74.0 + ] 31.0 / %,@FA Z,@N0,@J1
@?0.0 14.0 97.0 / | ( \ 0.0 \ A { ; @FY C ),@?119.0 120.0 {,@N1,@N2,@FZ -154.0 | !
@?i ; @N1,@?j Y { [ 114.0 * 101.0 + 133.0 -,@?g
```

Fitness of 776382, found at generation 364 on Run 3.

The individual seems to start poorly, with a few useless movements before performing an assignment to variable C, which is later used to determine the forward motion of the Antbot on line 2 (after one iteration). Otherwise, the Antbot will turn to the right from line 1. It is a simple single-sonar operation, but it will use this single left-front sonar to maintain a set distance from one of the walls, allowing it to hopefully avoid the other based on the width of the track. It is not a truly general solution, and will likely fail in real-world testing, however. State 1, like in many cases, is never reached, hence the rather gibberish operations within it.

Observations: Average fitness is relatively low, and randomness has been increased due to the distributed populations (see figure 28). As has become expected by this point, selection pressure is too low to maintain the use of multiple populations. Only one interesting individual was produced, though most of the rest were close to the threshold.

Physical Antbot Test: The Antbot avoids obstacles to its left, and drives in a somewhat circular pattern to the right – if it encounters an obstacle on its right, it will collide.

7.1.8. Hi-Lo Fit combined with Multiple Distributed Populations

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	688733*	39	1333
1	109352*	428	1258
2	133249	424	1380
3	775919*	214	1358
4	778351	433	1358
5	1017229*	421	1281
6	1436470*	339	1193

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 705614

Average distinct individuals at generation 500: 1308.4

Fitness Graph:

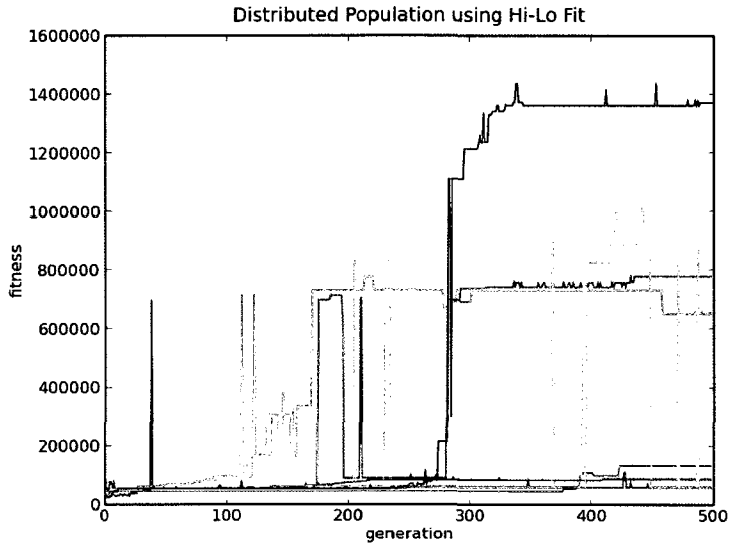

Figure 29: Multiple distributed populations, combined with Hi-Lo Fit

Best Individual:

```
STA 0
@?0.0 ; @J8,@?A -76.0 },@FX -88.0 X h % - 101.0 \,@N1
@?Y ; @J115.0 ! !
@?0.0 ) 0.0 ( _ X | | ( | % ` 155.0 } [ ; @N1,@FY 87.0,@FZ -111.0 j h ( ~ ` \
      -144.0 } -5.0 C % {,@?i |
@?Y ; @?Z,@?-118.0 154.0 ` + ` C /,@FX -143.0 ) ! Z 95.0 -78.0 * ] ~ 20.0 ~ ( `
      %,@N2,@FB -33.0 -13.0 _
@?j | ; @FY B ` -117.0 ! | / !,@?-153.0 125.0 { `
@?0.0 ; @NX -49.0 `,@N2,@N2,@N1
@?g ; @FC -103.0 X ] +,@N2,@J4
@?X ; @J1,@N5.0,@?-56.0,@N2,@N2
@?B 0.0 ~ ~ ~ g * > ; @FZ Y,@J6,@?Z
@?X ; @FB B j j ~ -,@N1,@N2,@N0
@?33.0 ; @N2,@FB g B ) ~ ] -83.0 _ Z Y g h * - ] } ] Y 104.0 ~ ` ` ` X % C [ ~ + h
      Z % { ] [ ] *,@J8,@?144.0 |

STA 1
@?Z ; @FY g -45.0 /
@?Y ; @N0,@FB X ],@?113.0 Z [ 151.0 \ [ C { ! ~,@FA -64.0
@?0.0 h * ; @J3,@J7
```

Fitness of 1436470, found at generation 339 of run 6.

From a navigation standpoint, it is difficult to analyse where the Antbot's turn decisions are made due to recursive variable math, but it appears to primarily be in the third-last line of State 0, where it decides whether or not to turn to the left for two ticks. It will always turn right, and always go forward, but being able to variably straighten its course makes it able to actively navigate the straight course in concentric circles, while being probably somewhat overfit to the clockwise circle.

Observation: Multiple populations give much better results when combined with Hi-Lo Fit than alone. The increase in selection pressure is sufficient to achieve a much higher average best fitness, as well as individuals that show some code patterns that resemble human-engineered navigation systems. Most individuals break the threshold of 750 thousand, and in many cases (especially run 6, black, in figure 29) there is a steadily increase in fitness over time.

Physical Antbot Test: As predicted, the Antbot can navigate the straight course in concentric clockwise circles, and does so without an emergency stop. Its forward progress is quite slow, but measurable, and it never appears to collide with walls.

7.1.9. Module Acquisition combined with Hi-Lo Fit

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	717000	488	1170
1	1662895	395	1239
2	1822956*	287	1270
3	1719452	192	1241
4	183122*	170	1247

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 1221085

Average distinct individuals at generation 500: 1233.4

Fitness Graph:

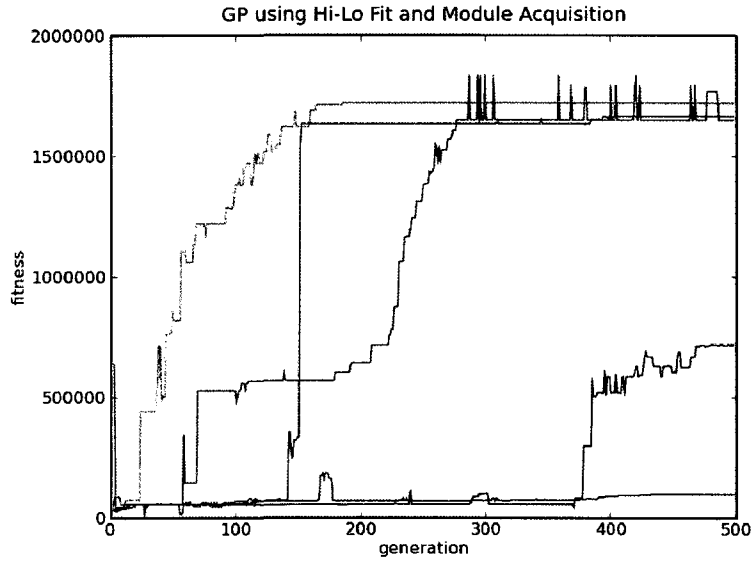

Figure 30: Module Acquisition combined with Hi-Lo Fit.

Best Individual:

```
STA 0
@?B ~ 0.0 11.0 < ~ < ; @J4,@J4,@J5
@?A ; @?Z `,@N2,@J4,@N0
@?1.0 ; @?-129.0 [,@J4
@?h g < ~ ; @N1,@FY B,@FY g g | | -,@J8,@FX 97.0
@?j g < ~ ; @J8,@FZ -162.0 [ [ )
@?g 14.0 > ~ ; @FX -85.0 17.0 ) / |
@?Y ; @J8,@N0
@?j ~ 124.0 + ; @FX 146.0,@?14.0 -167.0 ` j [ ~ _,@FA -74.0,@FB C (
@?85.0 ; @N1,@?C i -32.0 - -,@FY X i ` ` ] _ (,@N2
@?g ; @N0,@?116.0 | -93.0 ~,@FY Y ( -51.0 \
@?0.0 66.0 * ; @N1
@?39.0 ~ ; @FY 152.0,@FY i,@?145.0 85.0 | -168.0 ~ -136.0 * \,@N2
@?B ~ 46.0 > ~ X + ; @N0,@J5,@J3,@J8
@?B ~ j ~ g 0.0 + 0.0 B == == ~ == + ; @FA 28.0,@J7,@?72.0,@?X h _ )
@?0.0 ( C ) 68.0 / 0.0 ) / { ` ; @J5,@J7
@?B X 160.0 > < ; @?B,@FB -123.0,@FB -154.0,@?-28.0 68.0 } -104.0 ] ~,@?A
@?Y ~ ; @FY A,@J1,@N0
@?Z ; @FB g [,@FY -33.0 -19.0 /,@FZ j,@FC 103.0,@?i
@?0.0 33.0 < ; @N6,@FZ C Y 12.0 ] B } } j + \ [ | [,@N1
@?A 171.0 h C == ~ 169.0 ~ ~ * + * 0.0 == ~ ~ ; @J7
```

Fitness is 1822956, found on generation 287 of Run 2.

This state machine is deceptively simple with its single state. Its repeated use of comparisons between g (forward left) and h and j (the two sonars on the right) suggest a proper steering algorithm has evolved in this individual, superior even to the ones that appeared in testing.

Observations: A very high average fitness, and a single amazing individual make this experiment stand out from the others. Its loss of several good individuals to mutation, including the one above, suggests that it would perform more suitably with an increase in selection pressure, but the combination is definitely a sound one. Figure 30 indicates that many "spikes" in fitness could have been preserved through the addition of elitism here.

Physical Antbot Test: The Antbot moves in concentric clockwise circles forward, making counter-clockwise adjustments when it encounters an obstacle. When it reaches a wall, it will stabilize very close to or against the wall, and travel along it until it reaches the end of the wall, then return to concentric circles.

7.1.10. Module Acquisition combined with Multiple Distributed Populations

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	598295*	349	1517
1	190925*	444	1475
2	960050*	376	1521
3	600325*	327	1445
4	121003*	475	1503

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 494119

Average distinct individuals at generation 500: 1492.2

Fitness Graph:

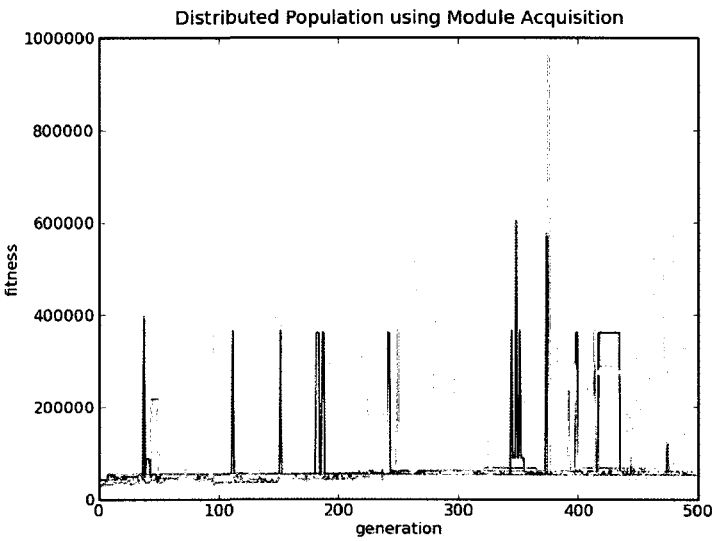

Figure 31: Distributed population combined with module acquisition

Best Individual:

```
STA 0
@?i ; @J7,@J2
@?A 0.0 ~ == ; @FX 88.0,@?i,@?Y 170.0 ` | ) ! ` {,@N1,@J2
@?Y ~ ; @N2,@FC 99.0,@FC h C Y -81.0 1.0 124.0 / ] | + \ -20.0 j X ] / - ( ( ( )
-69.0 { g / ! + +,@N1
@?0.0 ; @N0,@N1,@?Y -120.0 (,@FZ j [ Z _ ) ],@N0
@?146.0 ; @FC C,@?X,@J4,@N0,@?h ]
@?i 18.0 > ; @FX X ] h 93.0 * | +,@J6,@FZ 77.0 (
@?0.0 150.0 ~ 32.0 141.0 + < ~ == ; @FY 120.0 54.0 {,@N1
@?i 0.0 * 27.0 0.0 + * h 60.0 * * ~ ] ; @N1,@N0,@FZ Z,@J5


STA 1
@?i 0.0 < ; @?1
@?94.0 C 0.0 * + ; @J8,@J3
@?g ; @J2,@J7,@N2,@J5
@?A ; @N0
@?j ` 114.0 g 0.0 ) 0.0 ) ( % ) % % - ( [ ; @J2,@J6,@J4
@?g ; @N1,@?Z
```

Fitness of 960050, found on generation 376 of Run 2

This individual has a navigation scheme based on the value of the left-rear sonar sensor, making its major decision on the third-last line of state 0. If it is less than 18, the Antbot will perform a right turn, and hold the turn for at least three ticks. Its default motion appears to be reverse, with an automatic turn to the left for one tick, just before the sonar is checked.

Observations: Figure 31 shows considerable similarity with figure 22, traditional GP. There is only one strong individual out of the five, and good innovations are lost to a lack of selection pressure almost immediately, as the graph shows.

Physical Antbot Test: If placed at a 60 degree angle to the right wall as in the simulation, the Antbot will begin reversing, turning to navigate the corridor going backward, and drive down the corridor. The Antbot is able to avoid obstacles to its right, and the constant turning to the right allows it to avoid obstacles to its left. This individual is relatively general, and performs well in physical testing.

7.1.11. Module Acquisition combined with Elitism

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	719149	237	1174
1	6669892	487	1095
2	2063883	478	1193
3	688733	23	1309
4	887168	489	1079

Average Best Fitness: 2205765

Average distinct individuals at generation 500: 1170

Fitness Graph:

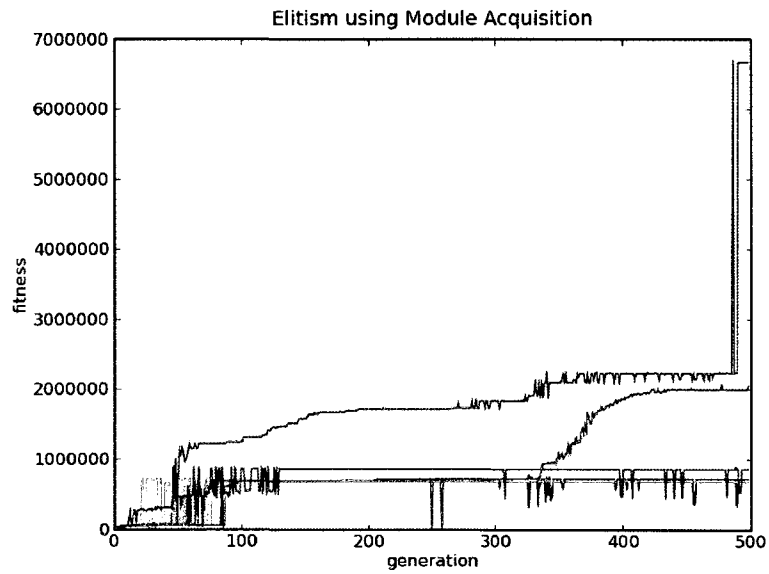

Figure 32: GP using Elitism combined with Module Acquisition

Best Individual:

```
STA 0
@?0.0 ; @N0,@FA 71.0 176.0 A } Z ( ` _ -28.0 * 74.0 B + ] Y 130.0 * / ] + -48.0 _ (
      -95.0 ~ \,@J2
@?100.0 ~ ~ ; @N0
@?Z ; @FZ 15.0,@?169.0,@N2
@?0.0 ~ ~ 35.0 + j > ; @J6,@N1,@?-47.0
@?h ( ; @FC j,@FZ -70.0,@J3,@N0,@J4
@?57.0 161.0 Y > < 17.0 h * * ~ ~ ; @N0,@?Z,@N2,@J3
@?Z j 0.0 == ~ == ; @N2,@J5,@J8,@FA g g ( g _ *,@N1

@?53.0 ; @J2,@?2
@?180.0 ~ ; @?h Y ] h | } ~,@J6,@?h,@FA A
@?0.0 ~ ~ 13.0 + j > ; @F1,@FX C ) |,@N1,@J6,@?Z


STA 1
@?h Y > g 11.0 0.0 + ~ * == ; @FZ A,@J2
@?172.0 ; @FY C,@?6.0,@N0
@?g ~ ; @?131.0,@FY Z 102.0 {,@?B 108.0 * A ( B - A _ B -55.0 / Z C * } % \ |,@N1
@?0.0 19.0 ~ ~ ~ ~ * ~ 98.0 ~ 111.0 < ~ + ; @J5,@N2,@N0
```

Fitness of 6669892, found on generation 487, run 1.

This individual shows some clear similarity to the hand-coded corridor-follower, such as turning and making a state change on line 3 of state 0. After that state change, it will either reverse, or continue turning, then return to state 0. The most interesting thing is the fact that it moves along using @J2, rather than @J8 – this indicates that the robot is progressing in a clockwise direction, in reverse, meaning in the circular simulation it first turned around, then progressed down the track. This is a minor case of overfitting to a similarity between the two problems (neither care which way the Antbot is facing, or if it turns around first) but there is a clear emergence of corridor-following behaviour here.

Observations: This parameter set produced three "good" individuals out of five, and the two remaining were close to the fitness threshold despite being likely local maxima (especially run 3). The combination of module acquisition to provide good reusable elements and elitism to maintain a higher retention of that good code provided for a very healthy environment for effective evolution. Figure 32 shows a relatively steady climb in fitness, with few instances of innovative material being lost.

Physical Antbot test: The individual shows clear wall-following behaviour, generally staying near the right wall (relative to the front of the Antbot). It makes its progress in reverse, and its fitness in the clockwise circle simulation is unusually good for a left-turning program – the reason is in fact its very strong ability to navigate – it will follow the corridor in concentric circles at a high enough rate of speed to earn a high fitness on both tracks. Additionally, when left to run in the corridors of the building, it was able to navigate around doorframes and corners with no difficulty – this is a very generally effective solution.

7.2. NEW PARAMETERS

As mentioned several times in 7.1., many of the runs faced insufficient selection pressure. Because of this, a "classical" parameter set for mutation and crossover was selected and some of the experiments were repeated. The experiments chosen were the baseline and distributed populations as the two most affected by the selection pressure, and the combinations of hi-lo fit and module acquisition, and hi-lo fit and elitism, which were high-performing combinations that will act as a control. The results of these experiments should show whether the low selection pressure and high mutation was a significant negative effect on previous results, both in cases with low average fitness, and high average fitness.

The final parameter set was:

Mutation Rate	0.05
Crossover Rate	0.9

Otherwise, parameters were as defined at the beginning of Chapter 7, in table 6.

7.2.1. Traditional Genetic Programming

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	593986*	355	840
1	592200*	346	822
2	515822*	159	777
3	543470*	318	793
4	60577*	247	785

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 461223

Average distinct individuals at generation 500: 803.4

Fitness Graph:

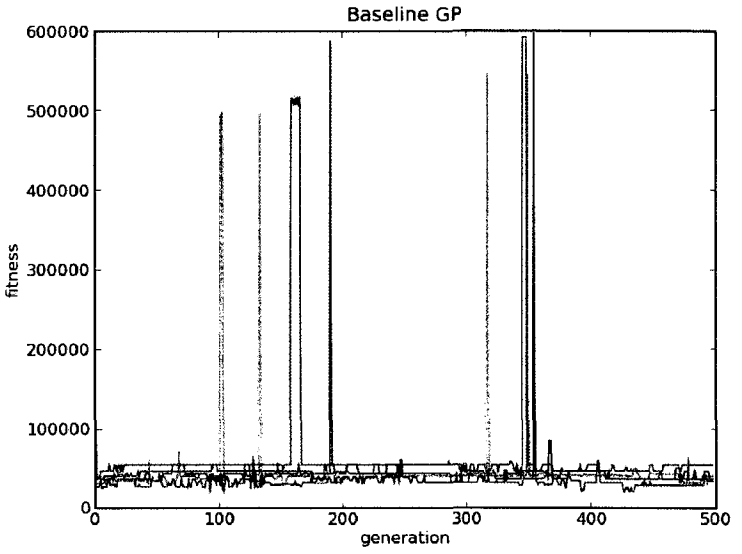

Figure 33: Traditional genetic programming with "classical" parameters

Best Individual:

```
STA 0
@?B ; @N2,@FC 47.0 156.0 j -26.0 * A ` 98.0 i - } + ` ! ~ ) / -122.0 ~,@J2,@N0,@?-
      141.0 B h % -168.0 } | +
@?3.0 j ~ ~ > ; @N0,@N1,@?C,@?76.0 g | -165.0 ) % ( 39.0 ` ` + ! } B [ | + ) ` 33.0
      ~ -47.0 /,@FB j
@?h ~ ; @J5
@?X ; @N2,@FY Z 90.0 +
@?0.0 ; @J5,@J4
@?0.0 ; @J8,@FB -55.0 [,@N0

STA 1
@?9.0 g > ~ 0.0 == ; @J1,@N2
@?B ; @J7,@FB -52.0 -116.0 ` { -91.0 _
@?j B == ; @FB -61.0,@?Z ],@N0,@FX C
@?g ; @N0,@?86.0 g Z B / | h % _ -110.0 ) * _ ),@J4,@?i |
@?Y B == ~ ; @N2
@?0.0 ; @FX Y


STA 2
@?139.0 B * 117.0 0.0 + > ; @J5,@?65.0,@FY 19.0 -58.0 \ [,@J2,@FX -177.0
```

Fitness of 593986, found on generation 355, run 0.

This is another run where it appears the Anbot turns around before attempting to navigate in reverse, or earns a poor fitness on the clockwise circular course (quite likely the latter in this case due to its lower fitness than the individual in 7.1.11). It does not demonstrate nearly the navigational sophistication, only using the sonar rangefinders in equality comparisons on lines three and five of state 1. State 2 is a "die" state, that only appears to be reached if the Antbot's left front rangefinder detects an obstacle within 9 units (within one "move" of an emergency stop). This state causes the Antbot to drive in reverse until it crashes, possibly giving it slightly longer survival.

Observations: There is no obvious sign of improvement with this parameter set – the best fitness values at generation 500 were still very low, showing that most of the "good" information was lost, and the average best fitness was slightly lower (though not in a statistically significant way) than the baseline with the higher mutation rate. Figure 33 shows that all of the good innovation "spikes" were still lost despite the lower mutation rate. Of the five individuals, none even came close to the threshold of 750 thousand.

Physical Antbot Test: The individual makes no forward progress, but instead drives in clockwise circles. It is a clearly overfit individual to that particular track.

7.2.2. Multiple Distributed Populations

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	457122*	88	712
1	457122*	104	790
2	383335*	413	748
3	53357*	179	719
4	53357*	13	759

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 280858

Average distinct individuals at generation 500: 745.6

Fitness Graph:

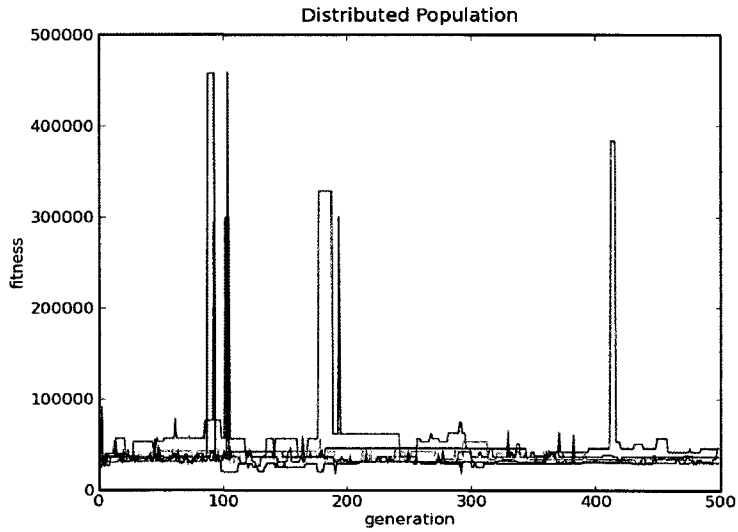

Figure 34: Genetic programming with distributed populations and "classical" parameters

Best Individual:

```
STA 0
@?0.0 ~ A > 167.0 < ; @N2,@FA 4.0 X * ( [ (,@?Y ! `,@N0,@FZ Y
@?C ~ ~ 178.0 + A * 153.0 < ; @J6,@FX g,@J4
@?153.0 h _ ; @FY -75.0,@J3,@?Z B \ 83.0 _ ` (
@?115.0 ; @?Y,@J1,@FB -124.0,@J8,@FB 115.0


STA 1
@?A g 0.0 + - ; @J3,@J6,@N1,@N1,@J5
```

Fitness of 457122, found on generation 88, run 1.

This individual shows no ability to navigate by its rangefinders at all – it is a clear case of overfitting to one of the courses, and using a very high fitness on that course to create a "good"individual. The individual did not persist in the population, and no other better individuals emerged to take its place. This robot simply alternates the actions of turning counter-clockwise and stopping, in a set pattern based on a set of relations on scratch variables X, Y, and A, which create a sequence of boolean activations and therefore a pattern of movement.

Observations: Again, no strong individuals were obtained at all, and the best individuals obtained were quickly lost (see figure 34). There is very little merit in this experiment configuration. There is no improvement compared to 7.1.3.

Physical Antbot Test: The Antbot emergency-stops on left wall. This is likely due to the corridor width not matching that of the simulation, and this individual being extremely overfit.

7.2.3. Hi-Lo Fit combined with Module Acquisition

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	10477	183	467
1	46074	16	497
2	2217039*	135	350
3	42391*	148	500
4	688733*	249	278
5	46074	16	520

* In this run, the best individual's fitness was lower than the highest fitness reached at the end of 500 generations

Average Best Fitness: 524064

Average distinct individuals at generation 500: 435.3

Fitness Graph:

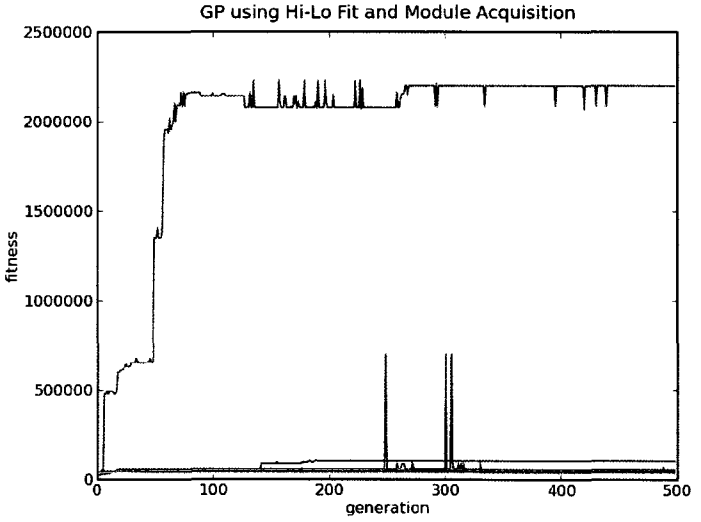

Figure 35: Genetic programming using hi-lo fit combined with module acquisition, and "classical" parameters

Best individual:

```
STA 0
@?j Z 41.0 { * ; @J4,@N1,@?-94.0 |,@?-161.0 C 92.0 { | +,@N2
@?g ~ 53.0 + Z > ; @?174.0 X ( i g \ _ } 82.0 + ( (,@J8,@N2,@FZ j,@FX B ` | -100.0
      A ) -33.0 * -144.0 ! \ -147.0 + -66.0 ] -147.0 + ` -61.0 \ _ ~ 48.0 C + 119.0
      % % ) /
@?77.0 C ~ X ~ 32.0 > + A ~ + 154.0 ~ + + ~ X ~ * ~ ~ ; @FA B,@N0
@?C ~ ; @FC C
@?h 157.0 < ; @J8
@?j Z 41.0 { * ; @N0,@J6
@?j Z 41.0 { * ; @J5,@J8,@?i
@?143.0 102.0 ( h + { ; @N0,@FZ g i [ +,@N1,@?-56.0
```

Fitness of 2217039, found on generation 135, run 2.

This individual primarily uses the scratch variables Z, X, A, and C to make navigation decisions with some delay based on the rangefinders. It seems to be a fairly general solution, with no obvious overfitting to either problem.

Observations: The fitness results from this test were extremely variable, and little else can be said about its performance. One individual did achieve strong fitness, and some real ability to navigate, but two others did not even reach fifty thousand. What innovative results were achieved were mostly preserved from mutation (see figure 35) and allowed to evolve further. One point of note is that the number of distinct individuals was very low, which could have lead to early stagnation in the population, hence never getting an improvement in population fitness after generation 249 out of all six runs. When compared to the experiment in 7.1.9, only the best run shows any similarity to the strong showing from nearly all of those runs.

Physical Antbot Test: This program is yet again somewhat overfit to the circular track, only being able to avoid obstacles to the left of the Antbot and colliding with the right wall of the straight corridor.

7.2.4. Hi-Lo Fit combined with Elitism

Individual Best Fitnesses:

Run	Best Fitness	Birth of Best Fitness (Generation)	Distinct Individuals at Generation 500
0	90139	61	213
1	46074	4	253
2	712053	254	188
3	673224	29	228
5	67067	388	382

Average Best Fitness: 317711

Average distinct individuals at generation 500: 252.8

Fitness Graph:

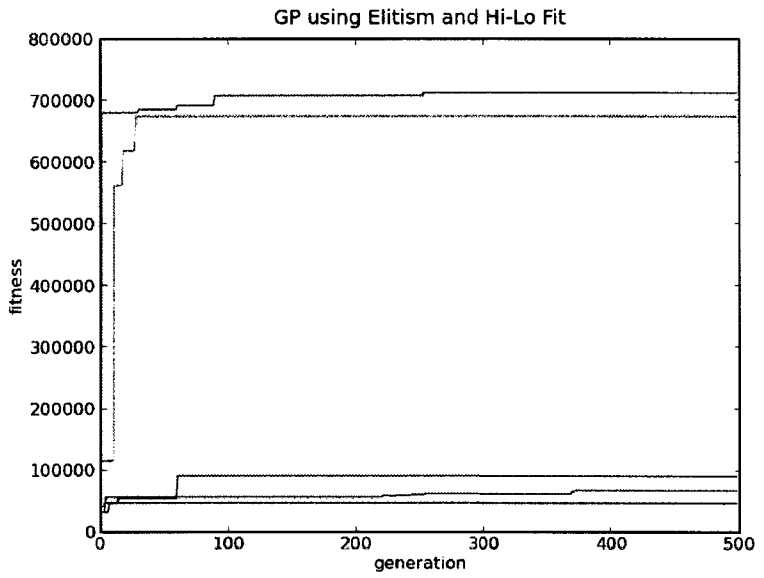

Figure 36: Genetic programming using elitism and hi-lo fit, and "classical" parameters

Best Individual:

```
STA 0
@?h X X ` ! } \ [ ; @J2
@?X ; @J4,@?-169.0 g C -9.0 % i } ! [ | | ) ! [ ` | _ } | ` j Y [ % \,@?C `,@?
64.0 ) B _,@FZ -79.0
@?A ; @?i,@N2,@J4,@N0
@?B ~ ; @N0,@J1,@J5
@?164.0 ; @FC X,@J4,@N1
@?40.0 g [ \ | Y + [ 0.0 { Z Z ( * / ; @?-169.0,@FA 23.0,@J5,@N1,@?-121.0
@?0.0 ; @N0,@J5


STA 1
@?128.0 ; @J8,@N1,@N1,@FC h -87.0 %
@?1.0 ; @N2,@FB A,@?164.0 [,@N1
@?0.0 ; @N1
@?Z ; @J4,@FZ h Z \
```

Fitness of 712053, found on generation 254, run 2.

This individual appears to attempt to navigate the straight track in reverse, without probing the sonars for information. Turns are made at fixed points on lines 3 and 5 of state 0. The second-last line of state 0 does not seem to ever execute, as it would sent the Antbot into an endless loop of driving forward and turning clockwise. State 1 is never reached. It is a clearly overfit individual that has evolved to meet the simulation parameters of the straight track, rather than find a general solution.

Observations: The variability of the populations was extremely low using these techniques and parameters, which is likely part of the cause of low fitness compared to the use of the same techniques in the originally-chosen parameter set (see 7.1.6). The average fitness was very poor, and no significant navigation techniques or strong individuals evolved at all. Figure 36 does show that elitism worked as desired, however, and no innovative mutations were lost over time.

Physical Antbot test: When arranged at 60 degrees to the corridor, it will navigate until encountering the right wall. Otherwise, it will collide with the wall closest to the rear of the Antbot, regardless of angle.

7.3. Notable Patterns

Throughout the various runs, some similarities were clearly visible between the different individuals. Many of these similarities did not execute (such as @?0.0 conditional statements), and acted as non-coding elements until they evolved usefulness. Other patterns, however, created definite run-time similarities between many of the individuals within, and outside of specific experiments. This section explores these common traits and attempts to explain them in some detail.

7.3.1. Overfit Individuals

In many cases throughout the various experimental runs, good individuals evolved that did not probe the rangefinders in any meaningful way, be it to manipulate their outputs into variables, or to use them directly in conditional statements. Instead, these individuals created patterns, generally of @J8s and @J4s or @J6s in a certain way to navigate either the circular or straight track, without completely failing on the other track. These individuals are not general solutions, and when physical testing was applied, they consistently failed in any but the most simulation-specific conditions. To reduce the appearance of these patterns in future experiments, additional simulations should be added to the fitness function. An example of an overfit pattern follows:

```
STA 0
@?122.0 ~ ~ 129.0 + ; @J2,@N6
@?108.0 ; @J5,@FX -29.0,@FC C ),@N1,@J4
@?j 0.0 + ; @J8,@N1,@FY Z,@J2,@?178.0 !
@?0.0 ; @FX i Y / g \ Z _,@N2,@J1


STA 1
@?0.0 X 0.0 ~ 155.0 * ~ < A < ~ 0.0 71.0 + < > ~ ~ ; @N0,@?C h )
@?X ; @FA h (,@J8
@?67.0 ; @FB A,@?X,@N1,@N2,@N0
```

There are no interesting conditions in this code, and every line will either always or never execute, regardless of variables – the program can in fact be simplified considerably to the following (with comments in italics).

```
STA 0
@?1; @J2                    drive in reverse
@?1; @FX -29.0, @N1, @J4    go to state 1 and start turning clockwise


STA 1
@?0              do nothing
@?1; @J8         drive forward
@?1; @N0         go back to state 0
```

This individual will simply alternate between making net forward progress (driving forward for two turns, reversing for one) and turning clockwise, so it can navigate a track. Similar patterns of @J8s and @J4s appear in many individuals.

7.3.2. Single State Steering

The majority of individuals evolved in the experiments showed a very similar behavioural pattern, using a single relevant state (sometimes divided into multiple states with a constant transition set, effectively acting as one state) with zero or more "junk" states containing non-executing or irrelevant instructions. These states contained some conditions, based on the rangefinders, in which the Antbot would drive forward, and other conditions in which it would turn in one or more directions. In some cases the forward motion was constant, in other cases the rotational motion was constant, but in all cases there was some sonar-based stimulus causing the Antbot to turn to avoid obstacles. An example of this simple steering method follows, using italics to indicate comments:

```
STA 0
@?B ~ 0.0 11.0 < ~ < ; @J4,@J4,@J5                    stop
@?A ; @?Z `,@N2,@J4,@N0                               turn clockwise
@?1.0 ; @?-129.0 [,@J4                                turn clockwise
@?h g < ~ ; @N1,@FY B,@FY g g | | -,@J8,@NX 97.0     if g < h, drive forward
@?j g < ~ ; @J8,@FZ -162.0 [ [ )                      if g < j, drive forward
@?g 14.0 > ~ ; @FX -85.0 17.0 ) / |                   irrelevant
@?Y ; @J8,@N0                                         never executes as Y = 0
@?j ~ 124.0 + ; @FX 146.0,@?14.0 -167.0 ` j [ ~ _,@FA -74.0,@FB C (
@?85.0 ; @N1,@?C i -32.0 - -,@FY X i ` ` ] _ (,@N2
@?g ; @N0,@?116.0 | -93.0 ~,@FY Y ( -51.0 \
@?0.0 66.0 * ; @N1
@?39.0 ~ ; @FY 152.0,@FY i,@?145.0 85.0 | -168.0 ~ -136.0 * \,@N2
@?B ~ 46.0 > ~ X + ; @N0,@J5,@J3,@J8
@?B ~ j ~ g 0.0 + 0.0 B == == ~ == + ; @FA 28.0,@J7,@?72.0,@?X h _ )
@?0.0 ( C ) 68.0 / 0.0 ) / { ` ; @J5,@J7
@?B X 160.0 > < ; @?B,@FB -123.0,@FB -154.0,@?-28.0 68.0 } -104.0 ] ~,@?A
@?Y ~ ; @FY A,@J1,@N0
@?Z ; @FB g [,@FY -33.0 -19.0 /,@FZ j,@FC 103.0,@?i
@?0.0 33.0 < ; @J6,@FZ C Y 12.0 ] B } } j + \ [ | [,@N1     turn counter-clockwise
@?A 171.0 h C == ~ 169.0 ~ ~ * + * 0.0 == ~ ~ ; @J7         stop
```

This individual, and those like it, use a sequence of commands in a single state to navigate – in this case, rotational motion is constant, and forward motion is determined by differences between the left (front and rear) and right front rangefinders. Good implementations of this pattern can earn very high fitnesses, and this sort of behaviour is responsible for the highest fitness achieved in these experiments.

7.3.3. Aiming and Steering States

Another set of characteristics that is common between many runs is a separation of aiming (rotational course correction) and steering (combination of forward motion and rotational course correction) into multiple states. The example follower in 2.5.4. is a relative of such an aim/steer design, though it continues forward progress within its aiming states. The NAVCOM AI uses a more explicit two-state machine for aiming and steering, only making forward or rearward progress in the steering state. An example of such an aim/steer design produced by GP follows:

```
STA 0
@?j 0.0 176.0 ~ h ~ + 107.0 * * + ; @J8,@FB -29.0,@N2    Drive Forward, go to state
                                                          2


STA 1
@?100.0 35.0 g < * ; @N2,@N1,@J4,@N0,@?-58.0 Y _      if g < 35, turn clockwise, go
                                                          to state 0
@?C ~ 10.0 < ~ h ~ ~ ~ B 5.0 * ~ < ~ ~ ~ B ~ X > > ~ ~ Z ~ ~ ~ < < ~ ; @N0   go to
                                                              state 0
@?0.0 ; @N0,@N0,@?A h _ 78.0 _                        never executes
@?0.0 g * ; @FX 151.0 ],@FA -29.0,@J2                 never executes


STA 2
@?0.0 ~ A == ; @J3,@N2,@?Y 135.0 21.0 { *,@?-17.0
@?100.0 h g < * ; @N1,@?Z B ) -167.0 _ -133.0 ! \ X j * { 143.0 ) \    if h < g, go
                                                          to state 1
@?46.0 ; @FB 105.0,@FY A !
@?C 0.0 * ) ; @N0,@FC -88.0 ),@FX -94.0 161.0 } ],@J6      turn counter-clockwise
                                                           and go to state 0
```

In this individual, states 1 and 2 combine to form an aim state, while in state 0, the system drives forward, then decides how it will aim. Due to the fixed transitions from states 1 and 2 to state 0, it does not have a perfect aim/steer algorithm (if it did, it would stay in the aim state until on course, rather than alternating between them) but the use of separate states for rotation and forward motion shows clear evolution toward that end.

7.3.4. Use of Rangefinders

In both single-state and aim/steer navigation patterns, the rangefinders are probed to determine at what distance the Antbot is from the walls, relative to its four corners. These values, g, h, i, and j, for left-front, right-front, left-rear, and right-rear corners respectively, are used in different ways, and through genetic programming, two clear comparison strategies emerged: rangefinder to rangefinder comparison, and rangefinder to static value comparison. In the human-designed corridor follower in 2.5.4. both of these strategies are used. Rangefinder to rangefinder comparison appears much more rarely in the results than rangefinder to value comparison. While approximately 10% of aim/steer and

single-state steer programs use rangefinder to rangefinder comparison, while all of them use rangefinder to static value comparison to some degree. These values, both for the rangefinders and the numerical values, may be stored in scratch variables (for rangefinder values, this results in operational latency), or used outright. Though there is no obvious correlation between the use of rangefinder to rangefinder comparison and very high fitness, many of the individuals that used it did achieve fitness values above two million.

7.3.5. Distribution of Patterns

These three high-level patterns can be used to describe nearly all of the results shown in this chapter. Those individuals that do not conform to these patterns generally have very low fitness, and can be considered "garbage" individuals with no functional merit. The table and pie chart below show a comparison of the number of each type of individual achieved as the best result of each run.

Experiment	Runs Performed	Garbage	Overfit	Single State Steer	Aim/Steer
Traditional	10	1	1	6	2
Elitism	10	1	1	8	0
Distributed Pop.	10	2	0	7	1
Hi-Lo Fit	5	1	1	2	1
Module Acquisition	5	0	1	2	2
Elitism + Hi-Lo	8	0	2	4	2
Distributed Pop. + Elitism	5	0	2	3	0
Distributed Pop. + Hi-Lo	5	1	0	4	1
Modules + Hi-Lo	5	0	1	4	0
Distributed Pop. + Modules	5	0	1	4	0
Modules + Elitism	5	0	3	2	0
Traditional (Classic Parameters)	5	0	1	2	2
Distributed Pop. (Classic Parameters)	5	3	1	1	0
Hi-Lo + Modules (Classic Parameters)	6	2	1	2	1
Elitism + Hi-Lo (Classic Parameters)	5	2	3	0	0

Table 7: Distribution of patterns in the best individuals of each run, by experiment type

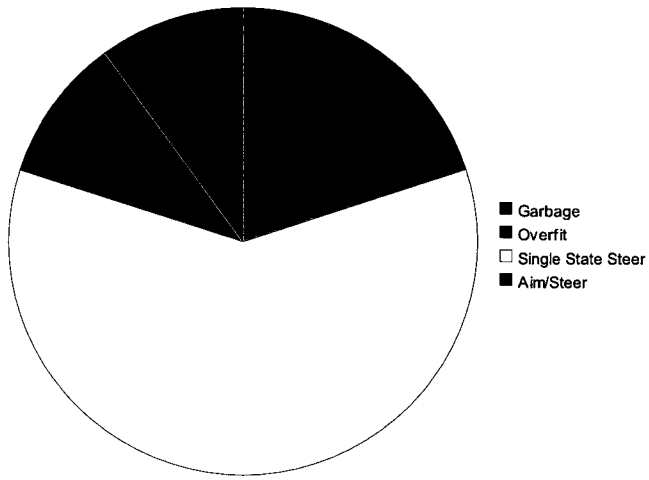

Figure 37: Distribution of patterns in the best individuals of each run

7.4. ANTBOT PERFORMANCE

Of all physical Antbot tests performed, the programs in 7.1.8, 7.1.9. and 7.1.11. performed best, navigating indefinitely without emergency stopping, and without requiring any external modifications at all. Of these, none of them used Aim/Steer designs, sticking to single states for steering, or loops between a group of states with no defined aiming state. This is likely because of the difficulty of designing a good aim/steer system, which is significantly more complex at the algorithmic level. It is likely with more experiments using larger populations, better results using more aim/steer patterns will emerge. Of the others, most of them were able to avoid obstacles on at least one side of the Antbot (usually the left, showing some overfitness to the clockwise circle), but demonstrate some flaw (usually then inability to avoid obstacles on the other side) that prevents them from being fully autonomous.

The individual in 7.1.11. stood out the most, showing its high fitness as a general solution in the real world. Able not only to navigate the straight course intended for the physical test, it is capable of navigating around door frames, floor debris and furniture, human beings, and other obstacles without collision, in any environment it has been placed in so far.

8. Discussion

With the simulated experiments and the field testing complete, there is strong support for the use of genetic programming to create robotic controllers in SEBeLS. Even during initial testing, evidence that good individuals could be generated with some good navigational patterns appeared. Once the various genetic programming techniques were added, results of varying effectiveness were generated, providing greater insight into the effectiveness of SEBeLS as a language for GP. This section will begin by discussing the effectiveness of the various genetic programming techniques explored throughout the thesis. Next, the influence of selection pressure on the usefulness of the results, and the poor final results of some experiments caused by insufficient selection pressure will be explained. Third, the accuracy of simulation versus the field tests on the Antbots will be discussed, along with any observations that may lead to improvements in simulation accuracy. Finally, conclusions about how effectively SEBeLS can be used alongside genetic programming to generate programs for robot controllers will be drawn and explained.

8.1. Effectiveness of Techniques

After all of the techniques and combinations chosen were tested for at least five generations, there were three techniques that showed the greatest influence when combined correctly: Elitism, Module Acquisition, and Hi-Lo Fit. Alone, these techniques produced little to no improvement to the average best fitness when compared to traditional GP. What was more useful, however, was the measure of elitism that all of these techniques added to the population, forcing the retention of many good individuals from the previous generation. With these techniques, new generations could be more often built on the best individuals in the previous one, rather than significantly mutated versions thereof. The definite downside to elitism, however, was population variance – in many cases, the logs showed less than 1300 unique individuals, meaning there were many, many clones in a population of 2000. In other selection methods, uniqueness ranged from about 1400 to 1600 unique individuals – not a massive change, but a significant one that reduces population variance. The advantage is that with elitism, one can be reasonably certain these clones are clones of the highest fitness individuals in the population, preserving them against harmful mutations.

8.1.1. Multiple Distributed Populations

The use of multiple distributed populations produced no measurable increase in average best fitness, even when combined with elitism, Hi-Lo Fit, and module acquisition. Some good individuals, notably the result of 7.1.8. were created using distributed populations, but these individuals are not likely to be connected to the technique used, and could have as easily emerged under traditional GP.

8.1.2. Elitism

Elitism provided no measurable increase in the average best fitness over multiple runs until combined with other techniques, notably module acquisition and Hi-Lo Fit. Like Hi-Lo Fit, below, it did provide a measure of retention of good individuals, increasing the likelihood that a good individual would survive and improve over 500 generations, instead of becoming lost to mutation.

8.1.3. Hi-Lo Fit

Hi-Lo Fit also worked to preserve high-fitness individuals, because every primary selection occurs in the more fit half of the population (note: when reproducing using hi-lo fit, remember that if no crossover occurs, only the first of the two selected individuals will carry forward), again ensuring that an elite portion of the population is retained – in some ways, using Hi-Lo Fit equates to a measure of elitism, though it is more difficult to measure its true value than if an explicit elite subsection is used. There is little in the related work to support Hi-Lo Fit being effective for robotics problems – until now, it has been a largely untested selection measure that has shown considerable merit. Though the results here lack sufficient statistical significance to declare it an improvement over tournament selection for solving these problems, there is definite correlation between higher average fitness and the use of Hi-Lo Fit, even (to some degree) without the addition of other techniques.

8.1.4. Module Acquisition

The use of module acquisition to provide a better solution mirrors the results of Pilat [42] in his thesis on hierarchical structures for genetic programming of mobile robots. In both this thesis and Pilat's, the use of module acquisition does not provide any increase in behavioural performance over traditional tree-based genetic programming. There is no clear benefit to the use of module acquisition in combination with any other technique, either.

8.1.5. Combinations of Techniques

Alone, neither elitism nor module acquisition had any noticeable effect on average fitness when compared to traditional GP. In combination with each other, and with Hi-Lo Fit, however, there was a considerable improvement. The average fitness of these three combinations was in the worst cases (Hi-Lo with Modules and Hi-Lo with Elitism) almost twice that of traditional GP, and in the case of elitism combined with module acquisition, the average best fitness was almost four times higher. Additionally, the best individual produced over five runs had a much higher fitness, and in all three cases provided a good solution in physical testing.

8.2. Influence of Parameters

As described in the beginning of chapter 6, the mutation and crossover rates chosen for the experiments are unusual when compared to those chosen by Koza, Nordin, and other experimenters described in chapter 4. The mutation rate chosen is very high, and the crossover rate much lower than these other experiments use. The high mutation rate is likely the cause of the "disappearing" high-fitness individuals: with a full 50% chance for any individual created by crossover to mutate randomly, and a 5% chance that any such individual will gain or lose an entire state to mutation, it is obvious that mutation has serious power over individuals changing fitness greatly. Interestingly, however, as shown in 7.1. and 7.2., this loss of good individuals occurred even when the mutation rate was reduced to 5%, and the crossover rate increased to match previous experiments. As explained by Pilat[42], this loss will always occur where elitism is not present, and it appears that some measure of elitism (or an elitism-like effect such as Hi-Lo Fit, see 8.1.3) should be used to create the best results.

As section 7.2 shows fairly clearly, there is a decrease in average best fitness when using the classical parameters described in 6.2.5. Even when previously good techniques, such as the combination of elitism and module acquisition are used, average best fitness barely approaches even that of traditional GP using the parameters obtained through testing in 5.6. Given the common nature of these parameters, it is likely that the reason for the higher mutation rate providing better fitness is a product of the parameters of the mutation and/or crossover operators, or a property of SEBeLS itself. More exhaustive exploration of the parameters is required for a conclusion based on sound statistical hypothesis testing to be drawn on this topic.

An additional possible cause for the variance where tournament selection is used is the value of p (the probability to select the best individual in a given tournament) and k (the size of said tournament). The value of p, 0.5, may in fact be too low for accurate results to be generated at the later stages of evolution, though it does promote more variation in genetic material. Possibly a lower k, combined with a higher p, will allow for some randomness while still retaining a strong measure of selection. This, along with other possibilities, should be tested on a case-by-case basis for each combination of techniques being considered for use. The counter to this argument, however, is that in experiments such as 7.1.4. and 7.1.8. similar loss of genetic material is shown without the use of tournament selection. Again, more exhaustive testing on the experimental parameters is required for a conclusion to be drawn.

8.3. Effectiveness of Mutation and Crossover

The new mutation and crossover algorithms described in section 5.4 have shown considerable promise after reviewing the results obtained. Given that the "classical" parameters with a higher crossover rate resulted in poorer results, the crossover operator's parameters may be significantly below optimal. The mutation operation, however, has shown a high degree of effectiveness, and its internal parameters may be very close to the best possible for the given problem. We consider the likelihood that the design or of the crossover operator itself is the cause of the poor results using classical parameters very low, due to its similarity with more common subtree-based crossover. The level parameters chosen, however, are one potential source of the discrepancy.

8.4. Accuracy of Simulations

As described in section 5.5., the simulation used for determining the fitness of individuals in all of the GP runs is relatively simplistic, using a turn-based design which does not precisely model a physical environment. As such, some discrepancy between Antbot performance in simulation and in practice is expected, though the results in chapter 7 show that it is not crippling by any means, and individuals that perform well in the real world have emerged.

The largest inaccuracy found between the simulation and the real world is the way the simulation reacts to the size of the Antbot. Because it is treated as a single point in a Cartesian plane in simulation, it is able to manoeuvre in ways that a physical Antbot cannot due to its rectangular shape. In many cases, even in the best programs, the trailing edge of the Antbot will bump or scrape a wall during a turn, though it does not emergency stop, nor does it impact at any speed to damage the Antbot. In certain environments this inaccuracy may cause behaviour hazardous to the Antbot, though it is unlikely in all but the most dangerous conditions, due to the low impact velocity and small area of contact. Given the physical testing results, this appears a rather minor inaccuracy in this case, and in the best result of all GP runs, this behaviour is eliminated, showing that as fitness improves, the slight inaccuracy of the simulation becomes less of an issue. This observation, along with the behaviour of the individuals in 7.1.9. and 7.1.11. demonstrate generality in the results of these experiments.

8.5. Summary of Results

In the end it appears that SEBeLS does have the capacity to be a good language for genetic programming. The results of the simulation-based experiments produced several excellent corridor-followers that operated in the real world, including in more noisy environments with more difficult obstacles than in the simulations. Even when a run's best individual does not show good performance in the real world, there is a very high probability that it will demonstrate one of the good patterns shown in 7.3 – more than 75% of the best individuals generated by each run demonstrated either a single steering state or some sort of aim/steer state machine that resembles the NAVCOM's two-state method. The lack of defined roles for states (such as aim/steer) is likely due to a lack of time to evolve – in

about 20% of cases, crude aim/steer systems with low fitness do appear, but their fitness is too low to overtake the single-state steering systems at this point. The peak fitness of these aim/steer systems is likely to be higher given more time to evolve, but at the moment, single state steering is better. This is the case for the evolution of many complex structures, both in nature and man-made – a new complex machine is not likely to be superior to its simpler predecessor, but it has greater potential, and will outperform it in later iterations. Even in the NAVCOM's case, optimizing the aim/steer system for a given platform takes weeks of human-driven testing, whereas the simple steering algorithm is much faster and easier to optimize.

The results are comparable to various experiments on the Khepera robot, by Nordin and Banzhaf [39,40,41], Pilat [42], and others, such as Koza, who evolved working wall followers, though only in one such case [40] were the results of such an experiment tested in the field. The individual generated in 7.1.11. surpassed any of these best individuals in terms of generality, and its navigational capability approached the aerial navigation results generated by Barlow [6,7] in terms of robustness, a trait he was directly selecting for in his experiments. Additionally, the selection for zero obstacle collisions in simulation translating to the real world is something not done in any but Barlow's experiments (due to the nature of UAVs and the problems associated with collisions).

In the end, we have created a language for genetic programming, and a framework around it, that is capable of evolving individuals with excellent generality in obstacle avoidance behaviour. We have also developed very flexible new mutation and crossover operators for the flatter program trees used in this language. Finally, we have provided, in section 2.5 and 2.6 testing and benchmark results of this language on the Robots Everywhere Antbot, showing its improvement in performance over the current default language on the platform.

8.6. Future Work and Improvements

While the quality of results obtained from the initial experiments given in chapters 6 and 7 is quite high, especially in terms of the real-world performance of some individuals, the parameters used were likely very far from optimal. The most significant additional work that can be done from these experiments is therefore a more exhaustive search of the level parameters for mutation and crossover, in addition to the other parameters used in the experiments, for optimal values. Likely, these values will be somewhat problem-specific, and a general set of values able to evolve good programs for a number of problems would make a clear baseline. One potential way to accomplish this task is to use another genetic algorithm to co-evolve parameters [15] based on the fitness distribution of individuals in each run produced. This would likely give a better understanding of the crossover operator's performance with various parameters.

Because this above suggestion would require large amounts of computing power, it is likely the Python-based GP framework has outlived its usefulness, and should be refactored into a more speed-oriented, compiled language such as C++. The ability to optimize the C++ code produced would likely give a significant speed increase over the automated code generation of Psyco [44]. Additionally, by

migrating the system to C++, or a similar lower-level language, the option of using distributed computing becomes available. Due to the crowd-sourced nature of the experiments in this thesis, it is likely a distributed approach over networked desktops would gain significant interest in the Internet robotics community, and a system resembling SETI@Home [59] could be developed. Through this system, new parameters, and new controllers could be evolved to solve more complex problems.

Finally, for more complex problems it is possible that the simulation environment is insufficient. Further experiments on the accuracy of various simulation environments using SEBeLS, either on the Antbot or on another autonomous platform, would benefit further work considerably. Through the use of more efficient languages and potentially distributed computing, a better, more resource-intensive simulator could be used, making results more accurate and helping to eliminate garbage results, and increase generality overall in the individuals evolved.

9. References

1. Ali, Elgasim Elamin Elnima. *A Proposed Genetic Algorithm Selection Method*, King Saud University, 2006.'
2. Alpaydin, Ethem. *Introduction to Machine Learning*, The MIT Press, October 2004, ISBN 0-262-01211-1
3. Anderson, Chris. *ArduPilot Main Page*, DIY Drones, 2009. http://diydrones.com/profiles/blogs/ardupilot-main-page last accessed September 2009.
4. Angeline, Peter and Pollack, Jordan. "Evolutionary Module Acquisition". *The Second Annual Conference on Evolutionary Programming*, February 1993, La Jolla, California
5. Bajracharya, M et al. "Autonomy for Mars Rovers: Past, Present, and Future", *Computer*, Volume 41, Issue 12, IEEE Dec. 2008. pp. 44-50.
6. Barlow, Gregory J and Choong, K. Oh. "Robustness Analysis of Genetic Programming Controllers for Unmanned Aerial Vehicles".
7. Barlow, Gregory J et al. "Evolving cooperative control on sparsely distributed tasks for UAV teams without global communication", *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pp. 177-184, ACM, 2008.
8. Ciesielski, Vic et al. "Genetic Programming for Robot Soccer", *RoboCup 2001: Robot Soccer World Cup V*, pp. 319-324, Springer 2001.
9. Dain, Robert A. "Genetic Programming for Mobile Robot Wall-Following Algorithms". *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pp.13-16, Morgan Kaufmann, 1997.
10. Darwin, Charles. *On the Origin of Species by Means of Natural Selection*, London: John Murray, 1859
11. Doncieux, Stéphane and Mouret, Jean-Baptiste. "Single-Step Evolution of Robot Controllers for Sequential Tasks", *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 1771-1772, ACM, New York, 2009.
12. Ebner, Marc. "Evolution of a Control Architecture for a Mobile Robot", *Proceedings of the Second International Conference on Evolvable Systems*, pp. 303-310, Springer, 1998.
13. F3 Boards. (http://www.f3.to), last accessed June 29, 2009
14. D'haeseleer, Patrik and Bluming, Jason. "Effects of Locality in Individual and Population Evolution", *Advances in Genetic Programming*, vol. 1, pp.177-198, MIT Press, 1994.
15. Fatemeh Vafaee, Weimin Xiao, Peter C. Nelson, Chi Zhou, "Adaptively Evolving Probabilities of Genetic Operators, pp.292-299, *Seventh International Conference on Machine Learning and Applications*, 2008
16. Gruau, Frederick. "Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm" (Ph.D thesis, L'universite Claude Bernard-lyon I, 1994).
17. Gruau, Frederick et al. "Cellular Encoding for Interactive Evolutionary Robotics", University of Sussex, Falmer. MIT Press, 1996.
18. Gruau, Frederick, and Whitley, Darrell. "A Programming Language for Artificial Development". *Evolutionary Programming*, vol. 4, pp.415-434, MIT Press, 1995. ISBN 0-262-13317-2.
19. Gwiazda, Tomasz D. *Genetic Algorithms Reference*, Vol. 1: *Crossover for Single-Objective Numerical Optimization Problems*, Thomas Gwiazda, 2006.
20. Hein, Karl and Meystel, Alex. "A genetic technique for robotic trajectory planning", *Telematics and Informatics*, Volume 11, Issue 4, Elsevier Science, 1994.

21. Holladay, Kenneth et al. "FIFTH™: A Stack Based GP Language for Vector Processing". *EuroGP 2007*, Volume 4445, pp. 102-113. Springer, 2007. ISBN 978-3-540-71602-0
22. Holland, John. *Adaptation in Natural and Artificial Systems*. MIT Press, 1975, 1992. ISBN 0-262-58111-6
23. Gagné, Christian, and Beaulieu, Julie. *Open BEAGLE W3 Page*. 2002. http://beagle.gel.ulaval.ca/. Last Accessed September 2009.
24. Khepera Team. *Khepera Robot II*. http://www.k-team.com/kteam/index.php?page=17&rub=&site=1. Last accessed August 2009.
25. Knaggs, Peter. *FORTH: An Underview*. *http://dec.bournemouth.ac.uk/forth/forth.html*. 2006. Last accessed August 2009.
26. Kovacic, Miha et al. "Genetic Programming Approach for Autonomous Vehicles", *Mechatronics 2004 9th Mechatronics Forum International Conference*, Atilim University, 2004
27. Koza, John R. *Hierarchical Automatic Function Definition in Genetic Programming*, *Proceedings of Workshop on the Foundation of Genetic Algorithms and Classifier Systems*, Morgan Kaufmann, pp.297-318, 1993.
28. Koza, John R. "Automatic Programming of Robots Using Genetic Programming", *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp.194-201, MIT Press, 1992.
29. Koza, John R. "Evolution of a Subsumption Architecture that Performs a Wall-Following Task for an Autonomous Mobile Robot Using Genetic Programming", *Computational Learning Theory and Natural Learning Systems*, pp. 321-346, MIT Press, 1994.
30. Koza, John R. "Introduction to Genetic Programming", *Advances in Genetic Programming*, vol.1, pp.21-45, MIT Press, 1994.
31. Luke, Sean. "Evolving SoccerBots: A Retrospective". *Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence*, 1998.
32. Matsumoto, Makoto. *Mersenne Twister: A Random Number Generator*. Hiroshima University, 1997. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html last accessed October 2009.
33. Michel, Olivier. *Khepera Simulator Homepage*, http://diwww.epfl.ch/lami/team/michel/khep-sim/, 1995. Last Accessed October 2009.
34. Miller, Brad and Goldberg, David E. *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. University of Illinois, 1995.
35. Miller, Julian F and Harding, Simon. "Evolution of Robot Controller Using Cartesian Genetic Programming", *Proceedings of the 8th European Conference on Genetic Programming*, pp. 62-73, Springer 2005.
36. Mondada, F., et al. "SWARM-BOT: A Swarm of Autonomous Mobile Robots with Self-Assembling Capabilities". *Proceedings of the International Workshop on Self-Organisation and Evolution of Social Behaviour*, pp 11-22, Monte Verità, Ascona, Switzerland, 2002.
37. Montana, David J. "Strongly Typed Genetic Programming". *Evolutionary Compuatation*, Volume 3, Number 2, 1995.
38. Ng, Andrew Y et al. "STAIR: The STanford Artificial Intelligence Robot project" *Snowbird*, 2008
39. Nordin, Peter and Banzhaf, Wolfgang. "Genetic Programming Controlling a Miniature Robot". *Working Notes for the AAAI Symposium on Genetic Programming*, pp. 61-67, AAAI, 1995.
40. Nordin, Peter and Banzhaf, Wolfgang. "Real Time Control of a Khepera Robot Using Genetic Programming". *Cybernetics and Control*, Vol. 26, Number 3, 1997.
41. Olmer, M. et al. "Evolving Real-Time Behavioural Modules for a Robot with GP", *Proceedings of the 6th International Symposium on Robotics and Manufacturing*, pp. 675-680, New York, Asme Press, 1996.
42. Pilat, Martin. "Hierarchical Learning Systems: Robotic Control Using Hierarchical Genetic

Programming" (M.CS thesis, Ottawa-Carleton Institute for Computer Science, 2003)
43. Parallax, Inc. *Propeller General Information.* (http://www.parallax.com/tabid/407/Default.aspx) last accessed 29 June, 2009.
44. Psyco Team. *Psyco Home Page.* (http://psyco.sourceforge.net) last accessed June 29, 2009
45. Python Software Foundation, *Random – Generate Pseudo-Random Numbers.* Python Software Foundation, 2009. http://docs.python.org/library/random.html last accessed October 2009.
46. RoboCup Soccer. *RoboCup Official Site.* http://www.robocup.org/ last accessed August 2009.
47. RML Technologies Inc. *Discipulus: Genetic Programming Software.* 2004. Last Accessed October 2009.
48. Robots Everywhere, LLC. *Robots Everywhere.* May 2008 (http://www.robots-everywhere.com/) Last Accessed November 2008
49. Robot Store (HK). *SRF05 – Ultra-Sonic Rangefinder: Technical Specification.* Robot Store (HK), http://www.robotstorehk.com/sensors/doc/srf05tech.pdf, last accessed October 2009.
50. Rosca, J. P. and Ballard, D. H. "Hierarchical Self-Organization in Genetic Programming", *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, 1994.
51. Russel, Stewart J and Norvig, Peter. *Artificial Intelligence: A Modern Approach.* Second Edition, Prentice Hall, 2003. ISBN 0-13-790395-2.
52. Scanlan, David A and Hebel, Martin A. "Programming the Eight-Core Propeller Chip". *Journal of Computer Sciences in Colleges*, vol. 23, issue 1. October 2007
53. Silva, Sara. *GPLAB: A Genetic Programming Toolbox for MATLAB.* Sourceforge, 2003. http://gplab.sourceforge.net/ Last Accessed September 2009.
54. Sparrow Autopilot. *Sparrow Autopilot – Queensland University of Technology.* Queensland Univeristy of Technology, 2009. http://www.sparrowautopilot.com/ last accessed August 2009.
55. Spector, Lee, et al. *README: a README file for the Pushgp programming system.* School of Cognitive Science, Hampshire College 2001. http://hampshire.edu/lspector/pushgp/README. Last accessed September 2009.
56. Spector, Lee, et al. *Push 3.0 Programming Language Description.* School of Cognitive Science, Hampshire College 2004. http://hampshire.edu/lspector/push3-description.html. Last accessed September 2009.
57. Swarmanoid Project. *Swarmanoid Project.* http://www.swarmanoid.org. Last accessed August 2009.
58. Togelius, Julien et al. "Evolving controllers for simulated car racing using object oriented genetic programming", *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp.1543-1550, ACM, 2007.
59. University of California, *Seti@home.* http://setiathome.berkeley.edu/ last accessed November 2009.
60. Yamamoto, Lidia. "PlasmidPL: A Plasmid-Inspired Language for Genetic Programming". *Lecture Notes in Computer Science: Genetic Programming.* Vol. 4971/2008, pp. 337-349, Springer Berlin/Heidelberg 2008. ISBN 978-3-540-78670-2.