COMP 4411 – RESEARCH PROJECT
By Mark Calleija 3061691 and Sam Mason 1058231
50/50 Contribution

SnakeBot
SnakeBot
1. INTRODUCTION
Where it all started
Why SnakeBot?
Aims for SnakeBot
SnakeBot 2D vs. SnakeBot 3D

2. HARDWARE
Existing Work
SnakeBot 2D
SnakeBot 3D
GA Learning Platform
Further Work

3. SOFTWARE
Aims
Existing Work
Motivation
Architecture and Technologies
Problems
Other Resources
Further Work
References

4. LEARNING
Aims
Existing Work
Motivation
Basic GA Description
Phase-Based GA Learning Framework
Fitness
Problems
Further Work
References

5. RESULTS
Aims
Testing Snsnake
GA Experiments
Bare Snake vs. Scaled Snake
GA Results
GA vs. Snsnake
Conclusion

6. THE END
Suggestions for future work
Conclusion

APPENDIX
A – System Parameters
B – SnakeBot Explorer Model
C – Fitness Evolutions
Introduction

This chapter provides an insight to our snake – a study towards one of the world’s greatest life forms.

Where it all started

The amazing diversity of snakes is an evolutionary success story that began more than 130 million years ago. The SnakeBot attempts to replicate in many ways the diverse and astonishing nature of these highly evolved animals. Such, it is the aim of this research project to apply appropriate scientific experimentation to our robotic snake, and report on our findings.

In session 2 of 2004, David Cock and Alex North (two COMP4411 students) designed and built a simple snake robot. It consisted of five servos, and could move in a variety of pre-programmed 2D gaits in the vertical plane, as shown in image 1.1. This project intends to extend on their research by dramatically increasing the performance of the snake in as many ways as time permits. The name SnakeBot refers to our research project, and reflects any contributions that are referred to within this document.

Why SnakeBot?

Snakes are one of the most versatile animals in the world, capable of movement in almost every imaginable medium except ice or snow. Because of this, it was decided that this research project should be based on these marvellous animals. The benefits of such a robot that could be controlled by a person would be endless, and would carry sure success within the field of searching through disaster sites. This success would be credited to the way in which a snake moves, unlike any other animal in the world.
Aims for SnakeBot

The aim for SnakeBot is to outperform other methods of robotic movement, so as to create a niche in the robotics industry. This niche would be one where a robot is required to move in an unknown 3D terrain of various material structures. A well-suited use of the SnakeBot would be to search through disaster sites, where extreme mobility is imperative. Clearly, the freedom of movement of a snake is our main advantage here, since its highly advanced bodily structure allows movement in a class of its own. Ideally the SnakeBot would be able to mimic every action, and more, that are currently genetic to real snakes - under human control. Because we have set ourselves an unachievable goal, we have set out by making a list of milestones, so as to allow a consistently improving snake, rather than try to build our ideal SnakeBot straight up. In chronological order, the milestones for SnakeBot are set out as follows:

1) Get the previous snake working
   a) Compilation and verification of previous groups code
   b) Configure home systems with Linux

2) Get a longer snake working
   a) Add 6 servos to the snake
   b) Program servo information
   c) Modify software to handle larger snake

3) SnakeBot 2D
   a) Build and modify SnakeBot to successfully operate in 2D
   b) Develop SnakeBot Explorer to control SnakeBot 2D
   c) Test SnakeBot 2D and record the results
   d) Using a GA, develop optimal forms of movement
   e) Compare GA movement to manual movement

4) SnakeBot 3D
   a) Remake SnakeBot 2D to operate in 3 dimensions
   b) Add a further 6 servos
   c) Modify SnakeBot explorer to control SnakeBot 3D
SnakeBot 2D vs. SnakeBot 3D

SnakeBot 2D operates in two dimensions. It is made up using all 11 servos operating in 1 lateral plane. Because SnakeBot 2D operates in just 2 dimensions, and all 11 servos operate in this plane, the SnakeBot 2D is capable of achieving a movement wave which represents approximately two periods of a sinusoidal wave, however, it cannot turn or wrap around objects. Image 1.2 displays a CAD representation of the SnakeBot 2D.

![Image 1.2 - SnakeBot 2D](image)

SnakeBot 3D (Image 1.3) operates in three dimensions, and involves a far more advanced operating technique than SnakeBot 2D. Unlike SnakeBot 2D, it is made up using 17 servos, however, they are set at right angles to one another to allow 3D movement. This configuration allows movement in a wide range of directions, and greatly broadens the range of movement capable. Table 1.1 lists movement capabilities, and servo quantities within each plane.

![Image 1.3 - SnakeBot 3D](image)

<table>
<thead>
<tr>
<th>Feature</th>
<th>SnakeBot 2D</th>
<th>SnakeBot 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement forward/backward</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Servos operating forward/backward</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>Movement left/right</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Servos operating left/right</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1.1 Comparison of SnakeBot 2D and 3D

Overall the SnakeBot 2D has better mobility when operating in a straight line than SnakeBot 3D. This is because SnakeBot 3D has only half of its servos operating to propel it forward, with the other half performing movements in a vertical, and somewhat useless dimension when it comes to straight path movement. However, SnakeBot 3D is capable of turning, which SnakeBot 2D cannot do. In a disaster zone, SnakeBot 3D would be of far better use, since likeliness would have it that the snake would not simply be moving in a straight path.
Hardware

This chapter looks at the hardware side of SnakeBot

Existing Work

The following hardware was already built when this project commenced:

1. 5 Segment Snake – basic servo frame with no grippers etc
2. RS-232 controller with tether

SnakeBot 2D

What is SnakeBot 2D?

SnakeBot 2D is the configuration of SnakeBot in which all servos operate in the same lateral plane. SnakeBot 2D is simple, and easy to understand, however, it is limited to movement in a straight line only. The original snake was already set up in 2D mode when this research project commenced, however consisted of only 5 servos. The vast potential of the snake’s movement was only really appreciated after the next 6 servos were added, which produced higher speeds, elegant and consistent movement and the ability to perform tasks such as climbing stairs, up poles and inclines of 45°.
Why SnakeBot 2D?

It was decided to commence the research project towards the study of the 2D SnakeBot configuration. This is because it is simpler, easier, and faster to gain accurate and concise results than for the 3D configuration - which is significantly more detailed when it comes to carrying out experiments than 2D. The 2D was seen as a good platform, as the basic movement and learning principles could be carried out within our time constraint, and would still provide an insight to robotic snake movement before stepping up to the next level – the 3D snake.

Problems and Solutions

Problem: Stability of the snake was poor at high amplitudes.

Solution: Add polystyrene stabilizers to each segment in order to level the bot. (Image 2.3)

Problem: Uneven frictional levels at different areas on each servo

Solution: Wrap servos with balloons to supply a consistent frictional surface (Image 2.4-2.5)

Problem: 1.2A power supply could not adequately supply power

Solution: 3A supply used, running at approx 1.5-2A

Image 2.3  Image 2.4  Image 2.5
SnakeBot 3D

What is SnakeBot 3D?

SnakeBot 3D is the configuration of SnakeBot in which alternating segments of the snake are set at right angles to one another, so as to allow movement in 3 dimensions. The configuration of SnakeBot 3D was set up, however, due to time constraints could not be experimented with much further.

Why SnakeBot 3D?

It was decided to at least build the configuration of SnakeBot 3D since it holds far more mobility potential than the 2D configuration. It was built to supply a framework for future work and will achieve far better results in disaster site locomotion than the 2D snake could possibly accomplish.

Problems and Solutions

Problem: Every second segment was hovering approx 1cm off the ground

Solution: Add razor formed balls to each servo to level itself out by marking out carving lines for each of the 17 balls (Image 2.7). Then using a red-hot knife, each ball was individually carved (Image 2.8). Balls were then added to each servo (Image 2.9) and coated with balloons (Image 2.10). The entire 17-segment snake was then rebuilt (Image 2.6).
GA Learning Platform

What is the GA Learning Platform?

The GA Learning Platform allows the SnakeBot to try different movements without having the snake leave the platform. Information about the movements the snake makes are fed back through the mouse. The platform was originally a manual treadmill purchased from K-Mart, but soon became the snakes learning bed through a few custom modifications.

![Image 2.11 – GA Learning Platform](image)

Why use a Learning Platform?

It was decided to use the platform purely because of convenience. On an average walk, the snake will travel a distance of around 90 metres, so the platform basically lets it stand still, whilst feeding back information to the computer via mouse motion detection. The downside to using the platform is that the environment is not the same as it would be if it were to be travelling on the ground because:

- The lever is not perfectly counterweighted
- The track requires some force to move, even though only a small amount

These factors, along with a few other minor ones contribute to the inaccuracies involved in using the Platform as a method of learning. However, these contributing factors have been minimised by the use of a counter weight, a long aluminium lever arm and a lightweight Glad-Bake surface.
Problems and Solutions

Problem: Mounting the mouse and Snake

Solution: Create support to hold both mouse and snake, whilst limiting vertical snake movement:

- Cut existing hand-rails to a much smaller height using a cold-saw (Image 2.12)
- Re-weld rails to support mouse and snake using a Mig Welder (Image 2.13)
- Create support for mouse and snake (image 2.14)

Problem: Snake would not move belt because of large steel disc on roller (Image 2.15)

Solution: Remove metal disk on belt roller using an oxy-acetylene cutter (Images 2.16-2.17)

Problem: Bearing housing on roller deformed due to heat from oxy-cutting (Image 2.18)

Solution: Re-Shape bearing housing using metal cutting lathe

Problem: Rubber belt too heavy for snake to move (image 2.19)

Solution: Use Glad-Bake as Platform mat - cut to size, sand back then super-glue together
Further Work

The final SnakeBot configuration has been set up in 3D mode. Only a small number of tests have been performed on this set up, such as:

✓ Check all servos work under 3D software system

✓ Use the 3D software to walk using 2 sinusoidal waves to control movement – not very successful

This means that the 3D configuration of the snake has a vast potential, and anyone performing future work should consider the following hardware additions:

• Wireless communication

• Battery operated with small batteries inside each segment

• Bi-directional scales on each segment to optimise movement

• Video Feedback to enable remote viewing
Software

This chapter describes the research platform software.

Aims

We designed and built the SnakeBot Software Suite for this project specifically with the following aims:

1) To allow easy manipulation of the Snake via an intuitive interface
2) To allow real-time output to be captured and recorded for later playback
3) To enable the robot to be manually controlled via the keyboard
4) To facilitate the GA learning process providing feedback via the mouse
5) To allow low-level control of segments and simulate damage in a search & rescue scenario
6) To provide a framework for a number of support tasks required while the SnakeBot is operating (background threads, file management, configuration, snake state management etc)
Existing Work

At the start of this project, we inherited the work done by the previous group last year. This was provided to us in the form of a number of C++ executables and Python scripts. The scripts were used by the previous group to do Reinforcement Learning using C-Trace.

This code base included the following use cases:

1) Sinsnake – a sinusoidal wave executable where wavelength and frequency are command line parameters
2) FileSnake – a simple executable that read a series of angles from a provided file and played it back (optionally) in a loop
3) Console – a text-based interface supporting a number of utility functions such as segment id specification, position read testing etc
4) ReadSnake – an executable that periodically polls the Snake for its current state and prints the angle set to std out
5) Relax – an executable that when executed causes the entire Snake to relax

To realise the above use cases, the previous group implemented a low-level serial communications package and abstracted the following Snake actions (commands) into a class called “Snake” (not exhaustive):

1) Initialise to listen on a specific serial port
2) Set a specific joint to a specific angle
3) Set the snake joints to a configuration specified by a supplied vector of angles
4) Query the Snake for the current angle of a specific segment
5) Query the Snake for the current angles between each segment (returned as an STL vector)
6) Query the Snake for the current workload (current in mA) of a specific segment
7) Query the Snake for the current workloads of each segment (returned as an STL vector)
8) Relax the entire Snake

Most useful for our purposes was the core SinSnake algorithm and the Snake interface (supported by the comms package).
Motivation

While the existing work was of good quality, we encountered a number of problems adapting most of it to the framework we had designed and planned to build. Some of these were due to the supplied code base, and others were specific to our skill set.

1) We have no knowledge or experience of Python (or the desire to gain either)

2) We do not believe that software should be constrained to run on a specific version of a specific operating system unless there is some justifiable reason why it should

3) Executing a series of scripts/executables from the command line is not a particularly intuitive way of exploring a learning problem of this sort of complexity

4) There were a number of hard-coded values in the code base (device name, number of segments in the snake etc) as well as large segments of duplicated code across the executables creating problems in maintenance and scalability

5) The mouse feedback mechanism was specific to the Debian flavour of Linux using Python scripts

Further, in order to implement learning via a genetic algorithm, we would need a number of more advanced features not possible from the command line, hence the need to build a GUI.

We also wanted to support manual control of the snake in real-time using the keyboard effectively allowing the user to “drive” the snake - somewhat like a game.

Finally, we wanted to leverage the part of the existing code-base most useful to us – the Snake interface in C++.
**Architecture & Technologies**

In order to re-use as much existing code as possible while achieving our aims, we decided on a two-component architecture: the SnakeBotServer and the SnakeBotExplorer, using C++ and Java technologies respectively. The server would act as a proxy to the Snake interface designed by the previous group, while all new of our new functionality would be implemented in the Java AWT based explorer interface. This allowed us to abstract high-level user actions to a sequence of low-level commands to the robot with a minimum of programming work.

Communications between the two modules is implemented using a simple text-based protocol and binding the respective STD in/out streams of the two applications. The server simply runs in a standard message loop waiting for the next command, which is then executed synchronously on the Snake instance. The server instance is a child process of the explorer and is therefore not intended to be started directly by the user (although the Explorer allows the user to restart the process via menu commands should that be necessary).

The sequence diagram for the high-level instantiations is shown below. Also shown is a typical command sequence.

The whole system is completely configurable based on a simple name/value pair format configuration file supplied on the command line at application start-up. Should such a file not be present, a set of “reasonable” defaults are used. Details of the available system parameters can be found in appendix A.

In addition, both components of the architecture support “test” and “operational” modes. When in test mode, the explorer will write every command sent to the Snake to the output console (whether from a macro file, GA learning or from real-time navigation). When in test mode the server does not communicate with the Snake robot – it simulates responses in software. This is useful for running the system without actually having the SnakeBot present and attached to the host computer.

To start the system, it is best to use a shell script (sample included on CDROM), with a command in the format:
java SnakeBotExplorer <path to configfile> <path to SnakeBotServer Executable> [-e] [-t]

From the syntax, it can be seen that the command line switches “-e” and “-t” are optional. If “-e” is present, the explorer will be started in test mode. If “-t” is present, the server will be started in test mode. Otherwise both components are started in their normal operating modes.

**SnakeBot Server**

As explained above, the SnakeBotServer runs in a simple message loop. We have cheated slightly here – the message loop simply blocks waiting for input from the standard stream, processes any commands, writes the result to STD out and blocks again. Therefore the term “server” is a slight misnomer – in a real system, this would listen on a port and spawn worker threads to handle incoming requests. However, since time was limited and a fully scalable server was not actually necessary, this simpler approach has been used.

The server processes the simplified command set explained above in the “Existing Work” section – ie :

1) Set a specific joint to a specific angle
2) Set the snake joints to a configuration specified by a supplied vector of angles
3) Query the Snake for the current angle of a specific segment
4) Query the Snake for the current angles between each segment (returned as an STL vector)
5) Query the Snake for the current workload (current in mA) of a specific segment
6) Query the Snake for the current workloads of each segment (returned as an STL vector)
7) Relax the entire Snake
8) Shutdown

Further explanation of the server is unnecessary. It was created simply to allow us to pipe commands to the existing Snake interface from another process (the SnakeBot Explorer) without having to bind directly to C++ from Java.

**SnakeBot Explorer**

The SnakeBot Explorer supports all the functionality we required for this project. The explorer is written entirely in Java using AWT. It should be run with at least version 1.5 of the JRE.

The Explorer contains three main sections – the Menu/Status bars, the Output console and the Control Panel. The Menu commands are tabulated below:

<table>
<thead>
<tr>
<th>Menu</th>
<th>Command</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explorer</td>
<td>Start Server</td>
<td>Starts the SnakeBot Server</td>
</tr>
<tr>
<td></td>
<td>Stop Server</td>
<td>Stops the SnakeBot Server</td>
</tr>
<tr>
<td></td>
<td>Quit</td>
<td>Shuts down explorer and server</td>
</tr>
</tbody>
</table>
### SNAKEBOT

#### Navigation

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Starts snake in basic real-time sine wave navigation mode</td>
</tr>
<tr>
<td>Pause</td>
<td>Pauses navigation</td>
</tr>
<tr>
<td>Resume</td>
<td>Resumes navigation</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops navigation</td>
</tr>
<tr>
<td>Record Macro</td>
<td>Starts recording a new macro</td>
</tr>
<tr>
<td>Stop Recording</td>
<td>Stops the recording of the current macro</td>
</tr>
<tr>
<td>Playback</td>
<td>Choose a macro file for playback</td>
</tr>
<tr>
<td>Stop Playback</td>
<td>Stop the current playback</td>
</tr>
<tr>
<td>Kill Random</td>
<td>Randomly select a segment of the Snake and “kill” it</td>
</tr>
<tr>
<td>Relax</td>
<td>Relax the entire Snake</td>
</tr>
</tbody>
</table>

#### Learning

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start GA</td>
<td>Start the GA learning process</td>
</tr>
<tr>
<td>Stop GA</td>
<td>Stop the GA learning process</td>
</tr>
</tbody>
</table>

The output console is a non-editable region of the application where output is displayed. Output may be the current angles being sent to the snake, information about existing processes or simply confirmation of user actions (e.g., the path to a macro file created as a result of a recording request).

The status bar displays the current relative “speed” when the SnakeBot is being driven under manual control. This is only available when navigating the Snake in real-time. The keystrokes to support this control are tabulated below:

<table>
<thead>
<tr>
<th>Key Stroke</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Increase amplitude</td>
</tr>
<tr>
<td>Z</td>
<td>Decrease amplitude</td>
</tr>
<tr>
<td>↑</td>
<td>Accelerate forwards (increase frequency)</td>
</tr>
<tr>
<td>↓</td>
<td>Accelerate backwards (decrease frequency)</td>
</tr>
</tbody>
</table>
In real-time navigation mode, the wavelength of the sine wave is held fixed, therefore the “speed” displayed in the status bar is always dependent on frequency only.

Early in the design process, it was decided that it would be extremely useful to be able to record a series of robot movements and play them back at some later time. We implemented this feature and call the resulting files SnakeBot “macros” (denoted by text files with the “.sbm” suffix).

The ability to record a macro is only available when the SnakeBot is actually in real-time navigation mode. Likewise the ability to playback a macro is only available when the Snake is not navigating. A simple macro file contains a series of pipe-delimited angle sets, which the playback mechanism will loop through at a configurable playback rate (set in the config file).

The macro file framework also supports comments, which is useful for including Meta data in the playback sequence. This is used particularly by the genetic algorithm (GA) code to record the settings that were used in the experiment that produced the recorded sequence. A sample of one such macro file is shown here:

```
# Population Size  8
# Number Of Phases 3
# Generations Per Phase 100
# Crossover Prob  0.9
# Crossover Type Two Point
# Mutation Prob  0.2
# Allele Set Range [-75.0..75.0]
# Allele Set Increment 15.0
#
# Phase 0 : elapsed Time 0.577 minutes
45|45|-60|-60|-75|45|45|15|45|-60|-15
# Phase 1 : elapsed Time 0.570 minutes
30|15|-45|45|0|-60|60|-75|15|60|30
# Phase 2 : elapsed Time 0.566 minutes
-60|30|-30|0|-30|0|45|-30|-30|15|30
# Elapsed Time 1.714 minutes
```

The control panel allows for granular control of each segment of the SnakeBot while in real-time navigation mode. Each SnakeBot segment has a numbered control column for it (see diagram above). In each column is a toggle switch between “Manual” and “Auto” (the default). When switched to “Manual”, the real-time navigation code will use the angle supplied in the textbox for that segment, as opposed to calculating it based on a propagating sine wave. This angle can be set using the slider or by direct entry.

At the top of each control column is a tri-state button displaying the segment number. Repeatedly clicking this button cycles the segment state through “Alive” (green), “Damaged” (orange) and “Dead” (red).

In this implementation however, we did not complete the “Damaged” state – thus segments are either alive or not. Therefore there are four possible states for a given segment affecting the angle sent to it whilst in real-time mode:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Status</th>
<th>Angle Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>Alive</td>
<td>Calculated by propagating sine wave</td>
</tr>
<tr>
<td>Auto</td>
<td>Not Alive</td>
<td>Whatever the last angle was before the segment died</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>Manual</td>
<td>Alive</td>
<td>The angle in the segments’ control column text box</td>
</tr>
<tr>
<td>Manual</td>
<td>Not Alive</td>
<td>Whatever the last angle was before the segment died</td>
</tr>
</tbody>
</table>

Down the right hand side of the control panel are command buttons that affects the state of every segment. The top three set the states (“Alive”, “Damaged”, “Dead”), and the next two set the modes (“Auto” or “Manual”) for all segments. The “Rand” button produces a random configuration of angles and at any time, the “Set” button will apply the current angle set to all segments in “Manual” mode.

A screenshot of the Explorer in test mode is shown below.

In addition to the features described in this section, the explorer also supports the capturing of mouse movement events when running in the learning mode. This will be discussed further in the following chapter on genetic algorithm learning.
Problems

Not surprisingly, there were a number of problems developing this system. Most of these were more or less standard software development problems, however, a few things stood out in particular:

1. Linux/Debian dependencies – we tried to remove all of these and develop a more robust system. Our optimal solution here would have been a complete Java port so that the entire SnakeBot server component would become redundant, but time did not allow completion of this sub-task. However, we have ported over the entire low-level communication package that supports the Snake interface. This code can be found in the “snakebotcontroller.comm.*” package in the provided code base (although this has not been tested, all that remains is to write the Snake interface and leverage the Java Communications API extension available from Sun).

2. A lack of a robust low-level interface to the mouse device as we could not use the Debian/Linux/C++ approach in our architecture. We investigated various USB and Serial libraries but in the end decided that it would be acceptable to “screen scrape” the mouse movement events by attaching an event listener to the main explorer window. This actually works extremely well, although we found it necessary to slow down the responsiveness of the mouse using the host operating system.

3. Finding a genetic algorithm library that we could use as is, or failing that, some source code we could use as a starting point. We did manage to find source code for two libraries (one C++ and the other Java) but neither directly met our needs. This particular issue will be covered in detail in the following chapter.

4. As we covered in our mid-session report, we also considered building a simulator using the Open Dynamic Engine system, but decided that we could not afford to risk taking this path. We had no valid reason for claiming that a virtual simulation would apply to the real-world snake learning process, so this idea was abandoned.

5. Simply getting an operating JRE installed under Linux was a difficult problem for us, but this was largely due to our inexperience with this particular OS.

Other Resources

With the exception of the Snake interface from the previous group and the Java GA library that we based our learning framework on, this work is entirely our own.
Further Work

There were a number of features that were not implemented due to time constraints and the change of project focus from manual navigation to GA learning. These include:

1. The development of online help available within the explorer

2. Direct mouse input using the Java Communications API found at http://java.sun.com/products/javacomm/javadocs/API_users_guide_3.html

3. A graphic display of the current configuration of the snake (using OpenGL). This would be easy to do and has applicability in a search and rescue environment, as the operators/observers would not be able to see the SnakeBot.

4. Completion of the difference between a “Dead” segment and a “Damaged” one. To do this however, requires the addition to the Snake interface of the ability to relax a single segment, which is not currently supported.

5. Completion of the port to Java. This would reduce maintenance greatly and remove the overhead involved with the custom text-based protocol between the server and the explorer.

References

The GA library we eventually used can be found at http://sourceforge.net/projects/java-galib
Learning

This chapter describes the genetic algorithm learning framework used in this project.

Aims

We had one very simple aim in this part of the project – to get the SnakeBot to learn to walk forward as fast as possible using an approach based on genetic algorithms.

Existing Work

Genetic algorithms have been around since the 1960s and as such there are many implementations available that could be used. The most popular one is called GALib and is an open source C++ library with the broadest feature set we could find. It may be found at http://lancer.mit.edu/ga/. While popular, it does not easily integrate into other systems and suffers from a lack of support (there is a mailing list but no-one responds).

In addition, GALib has compilation problems specifically with real number hypothesis representations (which is precisely what we wanted to use), so we abandoned this approach and found a simpler Java based library at http://sourceforge.net/projects/java-galib/.

This library lacked some core features we required, but being Java based it could be integrated directly into our code base. This library forms the core of our GA, although it has been largely re-written as will be explained later.
**Motivation**

In both our mid-session and final demonstrations, a number of people asked why we chose a genetic algorithm approach over reinforcement learning. We did so for the following reasons:

1. We had read in Kevin Dowling’s paper that achieving good results was not exceedingly difficult in the sort of time frames we had in mind.

2. The physical structure of the SnakeBot immediately suggests an easy representation of hypotheses as an array of bounded and discrete real numbers representing segment angles above and below the horizontal.

3. The previous group used reinforcement learning and we wanted to try a different approach.

4. This would be an exciting challenge – particularly given that there was some hesitancy as to whether or not the outcome would produce significant results.

**Basic GA Description**

Genetic algorithms were introduced by John Holland in the 1960s. There is essentially an optimisation learning technique motivated by a loose analogy to biological evolution.

GAs search a space of candidate hypotheses to find the best one – which is defined as the one that optimises a predefined (problem domain specific) numerical measure called the hypothesis fitness.

At a high level, a GA iteratively updates a pool of hypotheses called the population. On each iteration all members of the population are evaluated according to the fitness function. The next generation will then be created by probabilistically selecting the most fit individuals from the current population. Some of those selected are carried forward intact while others are used as the basis for creating new individuals using genetic operations such as mutation and crossover.

GAs also often use what is called an allele set (biologically inspired from chromosomes). An allele set essentially defines what genes (in our case angles) are allowed to be used when defining or mutating a hypothesis. It effectively provides a built-in set of constraints that can dramatically reduce the size of the search space the GA must operate in. It can be defined simply by a maximum value, a minimum value and an increment to use between the two boundaries.

For this project, the use of an allele set was critical. Each segment in the SnakeBot can move through 180° (+/-90 from horizontal). With 11 segments and no constraints the search space is so large (~$2^{626}$) that the algorithm would have little chance of finding any optimums.

As a simple example of the affect of an allele set on search space size, an allele set of +/-50° and an increment of 10° yields a search space of ~$2^{38}$ whereas simply changing the increment to 25° reduces this search space to ~$2^{17}$.

In our implementation, there is a configurable probability that the worst member of the current population will undergo a random mutation. This involves randomly changing a particular angle from its current value to another (as defined by the allele set). As a general rule, higher mutation rates will tend to help the GA explore more of the search space and increase the population diversity.
The crossover operation takes two members of the population and takes randomly selected portions of each to create offspring for the next generation. The chance of this happening is also a configurable probability (set to 90% in all of our experiments). Our implementation supports one-point, two-point, uniform and roulette crossover operators, but we note that in all of our experiments, we used two-point crossover for generating offspring.

Further details of crossover operations can be found elsewhere in standard literature.

We also implemented a feature we designed and called “generation convergence”. This can be enabled and configured or disabled in the configuration file. The aim of this is to allow the GA to be able to stop evolution early in the case where it has quickly converged on a local optimum position. This is mainly to prevent long learning periods where the best result (or very close to it) has already been found.

When enabled, this works in two stages:

1. At each generation the GA compares the current best hypothesis to the best from the previous generation and calculates their “similarity”. Similarity between hypotheses is calculated by comparing the angles from each in sequence. If each angle is within a certain number of increments (defined by the GENERATION_SIMILARITY_FACTOR parameter in the config file) of the other for the entire length of the Snake, then they are deemed to be similar (i.e. – basically the same shape).

2. If they are deemed to be similar enough, the GA will then calculate the fitness gradient over a number of generations defined by the FITNESS_GRADIENT_WIDTH parameter (see appendix A). If this gradient is effectively zero, then the GA’s best hypothesis over successive generations is not improving in fitness.

If both the above conditions are met, the generation are said to have converged and the phase completes early. An example of this in action can be seen in our final experiment where phases 1 and 4 converged earlier than the maximum of 300 generations possible (reproduced below).
Phase-Based GA Learning Framework

In order to get the SnakeBot to learn to walk, we had to design a framework where it could learn a series of positions to move to in sequence. This was a fundamental problem as there is no inherent way in a GA to include a priori knowledge.

Our solution is to have the learning system for SnakeBot run through a number of “phases”, where the result of each phase is a step in the final walk. At the start of a phase the SnakeBot is assigned an initial position, and the GA starts to evolve over the set number of generations.

In each generation, the SnakeBot moves to each hypothesis in the current population, one at a time, always moving back to the initial phase position before trying the next one.

When a phase completes, the result is stored and the next phase starts. The GA is then run again with all the same parameters to learn the next step.

What this means is that every time the GA runs (once per phase) it learns a new optimal position given its initial position.

Therefore we could address the problem of learning an optimal sequence of steps by making the following observation:

| The resulting walk should be a sequence of optimal positions provided that the initial position for phase $N$ is the optimal position found at the end of phase $N – 1$. |

In our experiments we kept the initial position for the first phase to all zero angles and the number of phases to run over to 4. The initial position and the subsequent 4 positions that are learned are written to a macro file as the learning framework runs.

We then playback the five steps from the macro file in a loop resulting in a walk (albeit with a discontinuity between the final learned step and the first all-zero one).

Based on this approach, we also tried to introduce the idea of “phase convergence” – so that instead of the framework running over a specified number phases only, it could run until it found a configuration that was “similar” enough to one already encountered in the learned sequence. This would use the same definition of similarity as generation convergence.

We implemented this, but as time was short and the first pass did not work very well it was removed from the framework. It would be interesting to follow up on as we feel that intuitively a Snake like robot would find some sort of periodic movement sequence, although there is no evidence to support this conjecture.
**Fitness**

The fitness function we developed works in two parts. Its task is to calculate a numerical number representing a relative fitness for a given hypothesis, which can then be used by the GA to favour positions that will drive the SnakeBot forwards.

1. The first part is based on the amount of forwards/backwards movement detected by the mouse. If the hypothesis results in a forward movement, the fitness value is a positive number equal in magnitude to that amount of movement. If the movement is backward (or no movement occurs), the fitness value is a negative number equal to twice the magnitude of the movement.

2. The second part of the function is only applied when a forward movement has occurred (i.e. the result of part one is a positive value) and represents an adjustment to the fitness value. We call this the “shape factor” which is described further below, and is shown in image 4.1.

As we developed this system we realised that we needed to “discourage” (assign low fitnesses) certain types of shapes that are semantically valid as defined by the allele set.

For example, if the allele set was +/-75°, the SnakeBot could curl up in a ball with all angles at +75°. While this might cause a forward movement, it would not be particularly stable and would unlikely to be a good starting point for a subsequent movement. Therefore part 2 of the fitness function penalises hypotheses with large amounts of angles of the same sign.

Based on the observation that hypotheses that are wave like are more likely to have alternating angles, we run through the length of the hypothesis and sum up all the angles. The more balanced it is, the closer to zero the angle sum should be. This sum is then normalised using the worst possible scenario (the curl described above) to a factor between 0.0 and 1.0, which we call the shape factor. More balanced hypotheses receive shape factors closest to 1.0.

*Image 4.1 - Examples of good, average and bad shapes*
Once the shape factor is known, the fitness value from part 1 is multiplied by this amount giving the final fitness value.

It should be noted that this fitness function does not in any way prevent bad positions (they are semantically legal as per the allele set); it merely strongly discourages their inclusion in the final sequence by giving them very low fitness values.

Problems

There were a number of problems we had to overcome to get this learning platform operational - most of which we have already discussed above:

1. The primary problem has already been explored in depth – how we could learn a sequence of steps to produce a walk. The solution is the phase-based learning framework.

2. It was also imperative that we not only find a usable GA library, but that it supported the use of allele sets. The C++ library did, but could not be integrated. The Java library could be easily integrated, but did not have this support. Therefore we took the source for the Java library, trimmed it completely down and added allele set support. In addition, we had to completely re-architect its structure to support phase-based learning.

3. Our initial tests operated correctly, but the learned walks were poor. We eventually discovered that we were playing back the learned sequence too quickly. So the SnakeBot would be in the process of moving to one position when another command would start it moving to a different one. As soon as we started playing back the sequence at the same learning rate, the quality of the resulting walk dramatically improved.

4. As explained above, we had to eventually develop a two-part fitness function. Initially we had no shape factor adjustment, and poor configurations would not be penalised.

5. The mouse feedback mechanism we have used works, but it would have been better to read directly from the USB device for more fine grained input. The problem was simply deciding which approach to take. We opted for the simpler event handler on the explorer frame approach for simplicity and due to time constraints.

6. Another problem to be dealt with was how to have the SnakeBot learn for long periods in a confined area. This was solved by the introduction of the treadmill as explained in the hardware section.
Further Work

Some ideas we had for further work include:

1. A detailed comparison to reinforcement learning. We thought about this early on, but the addition of a number of new servos makes any comparison between our work and that of the previous groups’ invalid (the more servos the SnakeBot has the faster it can go and ours is twice the previous length).

2. It has also been suggested that one could use reinforcement learning to minimise or prepare a search space for a GA to then operate in. This would drastically reduce the search space and should yield better results in a shorter time frame.

3. We have only implemented this single fitness function – whereas others could be developed that could change the learning process.

4. It may be possible to somehow use our shape factor concept to control the semantics of the allele set so that it could only produce “good” shapes (or ones most likely to be good). The system would then be imposing a stronger constraint on the search space at a more fundamental level rather than simply discouraging poor hypotheses.

5. We did not explore the affect of varying the different crossover types the system supports. This has an immediate affect on population diversity and would help prevent the system from converging on local optima.

6. Completion of the phase convergence concept outline earlier.

7. In order to perform a wide variety of experiments, we kept the number of generations very small (with the exception of our final experiment which used a maximum value of 300). Further work should explore the use of longer generations with and without generation convergence enabled.

8. Changing the physical configuration of the SnakeBot to 3D mode would have immense ramifications on the GA learning process. This would present challenges in both hardware and software for a future group.

References

Results

This chapter outlines our experimentation and the results that followed

Aims

There are a few aims of our experimentation, these are:

1. Test the speed of Sinsnake at a variety of settings in order to find a maximum speed.
2. Run the GA on a variety of settings in order to find a maximum speed.
3. Compare the use of manual control to the walks that the GA produced.

Note: All experiments were performed in the SnakeBot 2D configuration.

Testing Sinsnake

The first step towards our experimentation was to see how fast Sinsnake could go, so that when or if our GA produced walks, it could be compared to Sinsnake. Sinsnake was tested on varying frequencies and amplitudes, in order to find the fastest settings possible. Figure 5.1 details the use of amplitudes ranging from 2.3 – 2.7, which was a good window to be testing for high speeds, and frequencies ranging from 0.3 – 1.5hz. The following conclusions are drawn from this data:

- High frequency does not mean high speeds, nor does low frequency
- Higher amplitudes are faster, but become increasingly unstable as amplitude increases
- Sinsnake has a top speed of 71mm/s
**GA Experiments**

There were a total of 17 experiments performed using the GA. The details of all experiments are shown in Appendix C.

The first 4 experiments (set 1) were conducted with a maximum allowed angle of 75º. It was decided after these tests to reduce the maximum angle to 50º for all future tests since the bot would often try to curl itself up in a ball. The next 12 experiments (set 2) were then performed using the new max angle of 50º. This proved to produce good movements, which remained stable and produces good walks.

**Bare Snake vs. Scaled Snake**

There were two configuration of the snake throughout the GA testing; these included the bare snake and the scaled snake. The bare snake consisted of the following:

- 11 segment 2D snake
- Rubber balloons attached to each servo for added grip
- Stabilizers inside the balloons to level the snake

Note: The bare snake was the configuration used in ALL of the GA Learning Platform Tests.
The scaled snake was exactly the same set up from the bare snake, apart from having “scales” on the tail of the snake, which allowed maximum forward movement. These “scales” were devised using a leather belt with needles offset at a 45° angle to allow the tail to prevent any backward movement.

Image 5.1 shows the orientation of the scales on all segments (note that only tail scales were used in tests). The direction in which the scales point allows the snake to stop from moving in one direction, but allows movement in the other. From this, it was identified that the use of Bi-Directional scales would be a useful component for any future development.

**GA Results**

The following experimental data represents the tests of experiments 1-12 from set 2 in Appendix C. From this data it can be easily determined that the scaled snake produces far better speeds, consistently.
Experimental Data - Bare Snake

Experimental Data - Scaled Snake
The above graph clearly displays the difference between the average speeds with and without the use of scales on the snake. The scaled snake outperforms the bare snake in each experiment.

The following graph provides a clear understanding of how the different rate in which the snake performs its walk affects the speed of its walk. It can be concluded from this data that the rate that produces the highest walking speed lies between 150-450 ms, probably around 300ms.
The above graphs represent the average speeds based on either rate or mutation probability. The data here draws no clear conclusions, but vaguely represents the following:

- An upwards trend in speed for scaled snake with higher population sizes and mutation probabilities
- A downwards trend in speed for bare snake with higher population sizes and mutation probabilities

This suggests that as either population size, or mutation probability increases, the snake’s movement relies move on having the tail at a fixes position. However, no conclusions can be drawn from this data without further experimentation.

**GA vs. Sinsnake**

The GA had a maximum walking speed of 51 mm/s, compared to Sinsnake’s 71 mm/s. This shows that the GA performs comparably to Sinsnake, however is not faster. Future speed tests could prove that sinsnake is beaten by a GA walk by:

- Increasing the generations of each GA test to further optimise walks
- Further optimising the rate of commands to produce the highest possible speed
Conclusions

It was through around 20 hours of experimental work on the GA learning platform that the following conclusions have been made:

- Scaled snake consistently outperforms bare snake
- The speed of the snake does not increase as the rate of commands increases
- Scaled snakes performance tends to increase as both Population Size or Mutation Probability increases.
- As shown in figure 5.2, the Snake tended to form a shape which drew the tail in towards the head of the snake, which explains why the scales had such a dramatic effect on the performance of the snake

Image 5.2 – GA outcome tendency
The End

This chapter wraps up our research on SnakeBot

Suggestions For Future Work

Our thoughts of any future development have been described consistently throughout each corresponding chapter. A lot of work remains to be carried out before this bot would be of any viable value within the field of rescue.

Conclusion

The research conducted throughout this project has achieved a high level of success due to the logical approach taken in all aspects of its design, development, testing and analysis. The results drawn from the experiments have provided us with a clear understanding towards the approach that should be taken for any future work, whether it is hardware, software or learning based.

The abundant range of problems faced throughout our research were addressed swiftly yet accordingly, and the overall outcome of the project has laid the foundation of what could one day save the lives of disaster victims. The snake has proved to be able to learn movements at a basic level, and in relatively short periods of time, which leaves a world of potential for future development.

We have shown that the use of a GA has produced movements that may one day be extended to higher levels of the SnakeBot’s mobility. However, we believe that better results will be achieved with more extensive testing (i.e. allowing the SnakeBot to learn over longer periods). From this, the snake has been configured in its 3D state to offer anyone willing the opportunity to develop the manual and autonomous controls of this complex configuration.

Although we have successfully reached our milestones, the path towards a successful real life implementation of the bot remains a long and arduous one, however, we feel that we have contributed strongly towards this fascinating adventure through many hours of design, development and testing to what may one day be the first snake to ever save a person rather than kill one.
System Parameters

The parameter set can be divided into two main sections – parameters that drive the SnakeBot Explorer (general UI layout, SnakeBot characteristics, real-time navigation etc) and those that support the GA learning framework.

General Parameters

The general parameters are tabulated below.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNAKE_SIZE</td>
<td>11</td>
<td>Number of segment in the SnakeBot</td>
</tr>
<tr>
<td>PORT</td>
<td>/dev/ttyS0</td>
<td>The device name for the serial port the SnakeBot is connected to</td>
</tr>
<tr>
<td>NUM_SEGMENTS_IN_HEAD</td>
<td>4</td>
<td>Unused</td>
</tr>
<tr>
<td>HEAD_SEGMENT_ID</td>
<td>10</td>
<td>Unused</td>
</tr>
<tr>
<td>NUM_SEGMENTS_IN_TAIL</td>
<td>4</td>
<td>Unused</td>
</tr>
<tr>
<td>TAIL_SEGMENT_ID</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>COMMAND_PUMP_RATE</td>
<td>500</td>
<td>Time delay (ms) between successive commands to the SnakeBot when operating in real-time mode.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PLAYBACK_PUMP_RATE</td>
<td>1500</td>
<td>Time delay (ms) between successive commands to the SnakeBot when playing back a macro.</td>
</tr>
<tr>
<td>WINDOW_WIDTH</td>
<td>800</td>
<td>Explorer frame width</td>
</tr>
<tr>
<td>WINDOW_HEIGHT</td>
<td>600</td>
<td>Explorer frame height</td>
</tr>
<tr>
<td>CONTROL_PANEL_HEIGHT_PERCENTAGE</td>
<td>35</td>
<td>Percentage of the main explorer height to be used by the control panel</td>
</tr>
</tbody>
</table>

Parameters specific to calculations in real-time navigation (propagating sine wave) mode

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_ANGLE</td>
<td>45.0</td>
<td>Max angle in degrees</td>
</tr>
<tr>
<td>MAX_AMPLITUDE</td>
<td>1.5</td>
<td>Maximum amplitude factor</td>
</tr>
<tr>
<td>MAX_FREQUENCY</td>
<td>1.5</td>
<td>Maximum frequency</td>
</tr>
<tr>
<td>MAX_WAVELENGTH</td>
<td>2.5</td>
<td>Unused</td>
</tr>
<tr>
<td>OPTIMAL_AMPLITUDE</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>OPTIMAL_FREQUENCY</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>OPTIMAL_WAVELENGTH</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>AMPLITUDE_INCREMENT</td>
<td>0.1</td>
<td>Amount to change the current amplitude by every time ‘A’ or ‘Z’ keystrokes occur</td>
</tr>
<tr>
<td>FREQUENCY_INCREMENT</td>
<td>0.1</td>
<td>Amount to change the current frequency by every time ‘↑’ or ‘↓’ keystrokes occur</td>
</tr>
<tr>
<td>WAVELENGTH_INCREMENT</td>
<td>0.0</td>
<td>Unused</td>
</tr>
</tbody>
</table>
## GA Parameters

The parameters supporting the genetic algorithm learning framework are tabulated below:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA_COMMAND_PUMP_RATE</td>
<td>1</td>
<td>Time delay (ms) between successive training sequences</td>
</tr>
<tr>
<td>POPULATION_SIZE</td>
<td>30</td>
<td>Population size to use</td>
</tr>
<tr>
<td>NUMBER_OF_PHASES</td>
<td>10</td>
<td>Number of phases to execute (or steps to learn)</td>
</tr>
<tr>
<td>MAX_GENERATIONS_PER_PHASE</td>
<td>750</td>
<td>Number of generations a phase will evolve over unless convergence occurs (if enabled)</td>
</tr>
<tr>
<td>CROSS_OVER_TYPE</td>
<td>1</td>
<td>Possible types are 1 – one point, 2 – two point, 3- uniform and 4- roulette</td>
</tr>
<tr>
<td>CROSS_OVER_PROBABILITY</td>
<td>0.9</td>
<td>Probability that a cross-over will occur given two population members</td>
</tr>
<tr>
<td>MUTATION_PROBABILITY</td>
<td>0.05</td>
<td>Probability that the least fit member of the population will undergo a random mutation in each generation</td>
</tr>
<tr>
<td>ALLELE_SET_MIN</td>
<td>-30</td>
<td>Minimum angle below the horizontal that the GA may select when constructing a hypothesis</td>
</tr>
<tr>
<td>ALLELE_SET_MAX</td>
<td>30</td>
<td>Maximum angle above the horizontal that the GA may select when constructing a hypothesis</td>
</tr>
<tr>
<td>ALLELE_SET_INCREMENT</td>
<td>2</td>
<td>The increment to use between the min and max angle values to derive new hypotheses.</td>
</tr>
<tr>
<td>USE_GENERATION_CONVERGENCE</td>
<td>True</td>
<td>If true, the GA will attempt to stop evolution early if it can. If false, evolution will always continue to the value of MAX_GENERATIONS_PER_PHASE</td>
</tr>
<tr>
<td>GENERATION_SIMILARITY_FACTOR</td>
<td>1</td>
<td>Two segments whose angles are within this number of increments of each other are said to be similar for the purposes of convergence</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FITNESS_GRADIENT_WIDTH</td>
<td>75</td>
<td>This is the number of generations over which the GA will calculate the fitness gradient when using generation convergence</td>
</tr>
<tr>
<td>USE_PHASE_CONVERGENCE</td>
<td>False</td>
<td>Unused</td>
</tr>
<tr>
<td>PHASE_SIMILARITY_FACTOR</td>
<td>2</td>
<td>Unused</td>
</tr>
</tbody>
</table>
SnakeBot Explorer Model

A high-level class diagram of the Explorer is shown below. For clarity, only the high-level dependencies have been shown between the main system classes. For further details regarding properties and methods, please consult the source code supplied on the CDROM.
Fitness Evolutions

This appendix contains fitness graphs for all of the experiments that we performed throughout this project. It is included for completeness only as the result has already been discussed at length.

All experiments used a two-point crossover operator and a 90% crossover probability. The number of phases to learn was fixed at 4 and with exception of the final experiment, a maximum of 50 generations per phase was used with generation convergence disabled.

Set 1
These sets of experiments were all performed with an allele set of +/-75º with an increment of 15º. After only 4 experiments it was clear that the allele set allowed unstable configurations (too high amplitude) so further investigation was halted and a 50º allele set was used (see set 2 hereafter)

Experiment 1
Population Size – 10, Mutation Probability 20%, Elapsed Time 34.731 minutes
Experiment 2
Population Size – 10, Mutation Probability 50%, Elapsed Time 34.789 minutes

Experiment 3
Population Size – 10, Mutation Probability 100%, Elapsed Time 34.588 minutes
Experiment 4

Population Size – 20, Mutation Probability 20%, Elapsed Time 69.375 minutes

![Graph showing fitness over generations for different phases of Experiment 4. The graph includes lines representing different phases: Phase 1, Phase 2, Phase 3, and Phase 4. Each phase has a distinct color, and the y-axis represents fitness values ranging from -200 to 300.]
**Set 2**
This set of experiments was all performed with an allele set of +/-50°. We varied three different independent variables –

1) A population size of 10 and 20
2) Allele set increments of 10 and 25
3) Mutation probability values of 20%, 50% and 100%

This resulted in twelve core experiments the results of which have already been summarised. What follows is simply the record of how fitness evolved over generations for each of these experiments.

**Experiment 1**
Pop. Size 10, Increment 10°, Mutation Prob. 20%, Elapsed Time 34.825 min

![Experiment 1 Graph]

**Experiment 2**
Pop. Size – 10, Increment 10°, Mutation Prob. 50%, Elapsed Time 34.807 min

![Experiment 2 Graph]
Experiment 3

Pop. Size – 10, Increment 10°, Mutation Prob. 100%, Elapsed Time 34.847 min

Experiment 4

Pop. Size – 20, Increment 10°, Mutation Prob. 20%, Elapsed Time 69.133 min
Experiment 5
Pop. Size – 20, Increment 10°, Mutation Prob. 50%, Elapsed Time 69.366 min

Experiment 6
Pop. Size – 20, Increment 10°, Mutation Prob. 100%, Elapsed Time 69.221 min
Experiment 7
Pop. Size – 10, Increment 25°, Mutation Prob. 20%, Elapsed Time 34.708 min

Experiment 8
Pop. Size – 10, Increment 25°, Mutation Prob. 50%, Elapsed Time 34.520 min
Experiment 9

Pop. Size – 10, Increment 25°, Mutation Prob. 100%, Elapsed Time 34.847 min

Experiment 10

Pop. Size – 20, Increment 25°, Mutation Prob. 20%, Elapsed Time 69.800 min
Experiment 11

Pop. Size – 20, Increment 25°, Mutation Prob. 50%, Elapsed Time 69.330 min

Experiment 12

Pop. Size – 20, Increment 25°, Mutation Prob. 100%, Elapsed Time 69.323 min
Final Experiment
Pop. Size – 20, Increment 10°, Mutation Prob. 20%

Max Generations Per Phase- 300, Elapsed Time 319.266 min (5.3 hours)