Dynamic Software Evolution

An Evolutionary Approach to Artificial Intelligence

New Mexico

Supercomputing Challenge

Final Report

April 5, 2006

Team 126

Saint Pius X

Team Members:

Matthew Paiz

Ryan Loyd

Mark Wunsch

Teacher:

Kerrie Sena

Project Mentor:

James Carrie

Table of Contents

Introduction

In choosing a project, we wanted to select something that is unique and has never been done before. The thought of having an ordinary simulation occurred to us, and the positive effects of this type of simulation were considered. In the end, we decided to stay with our original idea of an evolving program. Its not every day that you can use a super-computer, and if were going to create a program worthy of being run on a supercomputer, we might as well take a risk and create something that is completely outside-of-the-box. Artificial Intelligence is still in its infancy and there is still so much to learn about it. If any thing could be discovered about artificial intelligence, it would be seen as a great benefit to humanity.

Executive Summary

What makes humans so smart? If one looks at any newborn child, it is obvious that vast knowledge and experience are not something given at birth. So then how can something very simple become a being capable of great thought and analysis? Any fool could tell you it is because humans have the capability to learn from their experiences. If a human fails at anything, he changes his approach and tries again. When he succeeds, he exploits whatever change brought him success. Over an extended period of time, a human builds up knowledge and experience and becomes a very advanced being. So then if a human can start off so simple and through trial and error become advanced, is it possible that a computer program can evolve in a similar fashion? Dynamic Software Evolution attempts to find out.

The idea of evolving a program has been around for a while. PhD's have attempted to create programs similar to Dynamic Software Evolution, but even they have received only a limited amount of results. Is it possible for three teenagers to yield any significant results? The first problem encountered during this experiment was uncertainty of where to begin. After spending weeks sharing ideas, a workable design was finally created. After another couple of weeks the design begin to evolve. We discovered that in order for anything to evolve, three things are necessary: an environment, some form of genetic material (DNA), and variation. After simulating these three things, it was possible to create Dynamic Software Evolutions.

After running our program, it became evident that we had succeeded. We had managed to create a program that could learn to go through a maze without being told how to do it.

Problem Definition

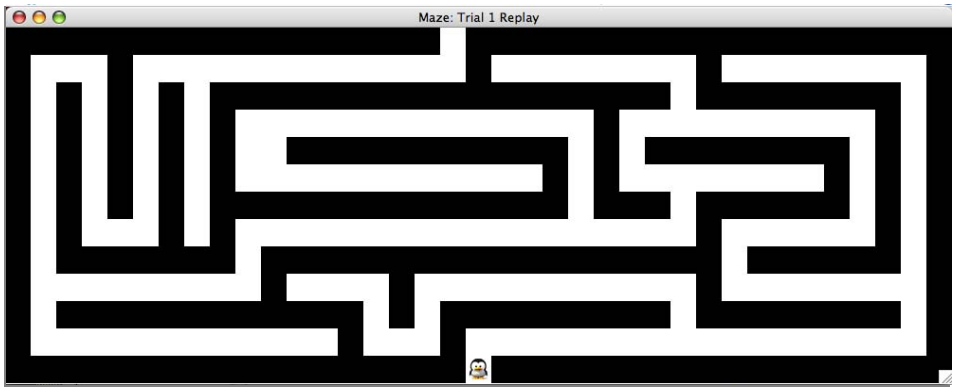
Arthur Samuel summed up the problem best with his quote on artificial intelligence, “How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?” Dynamic Software Evolution is a program that attempts to find the answer to the question above. We will attempt to create artificial intelligence using evolution.

Procedure

Previously stated, three things are necessary in order to simulate evolution. These three things are: an environment, some form of genetic material, and variation. Without these three things, evolution would not be possible. Dynamic Software Evolution extensively uses all three things in its design.

An environment can be defined as any condition that an organism is submitted to. The reason why an environment is essential to evolution is that it measures the fitness of an organism. The fitness of an organism is defined as how well an organism adapts to its environment. If the organism's fitness is poor, then the organism dies and has no offspring. If the organism's fitness is good, then the organism flourishes. When most people think of an environment, they imagine a forest or some other form of ecosystem. In reality an environment can be anything, it can range from forests to mathematical equations. All that is simply needed is a way to measure fitness. Dynamic Software Evolution's solution to multiple environments is a feature called the "*plug-in environment*". Support for plug-in environments means that Dynamic Software Evolution isn't only restricted to one environment. This allows a wide variety of programs to evolve using Dynamic Software Evolution. For example, you may want to evolve an answer to a mathematical equation. Using our program, you would only have to specify the characteristics of the environment and an organism could evolve in it. Some of the characteristics that are required in order to be an environment in our program include: some way to measure fitness, some form of GUI, and some way to communicate with an organism. At the heart of our program, is the Organism Class. The Organism Class is responsible for simulating an organism. The environment is represented by the

MiniApp Class. Any class that extends MiniApp can be classified as an environment in our simulation. The most widely used environment in this program is a maze. The reason why a maze was chosen to be our primary environment was because of its simple design and the complexity of its challenge. When an environment is executed it accepts an organism as an argument, and returns a success percentage. A success percentage ranks the organism on its fitness in the environment. The fitness score will be used later in the program to determine the number of offspring an organism can have. The higher the success percentage the more offspring an organism can have.



Screenshot of the Maze Environment

Genetic materials can be defined as any substance that genes can be stored in. The most common genetic material to humans is DNA. Evolution would not be possible without genetic materials because there would be no way to inherit the positive qualities of a previous generation. Without genetic material, an organism can never improve the design of another organism. If a program were to be classified as an organism then its genetic material would be its source code. So if you wanted to evolve a program you would modify and copy source code from other organisms. In Dynamic Software

Evolution, the programming language MiniScript represents genetic material. MiniScript is a language that we invented in order to simulate genetic material. The reason the language was invented was that Java source code would not be capable of evolution. There are a number of reasons why MiniScript is better suited for evolution than Java. First of all, Java is a very complex language and it would be hard for an organism to use such a complex language. MiniScript is a very simple language that can be easily interpreted by any environment. Second, Java has to be compiled then interpreted; this would slow down the program significantly. MiniScript is a scripting language so it does not have to be compiled. The last reason why MiniScript is better suited for evolution than Java is that MiniScript imports its commands from the environment. This means that the organism can talk to environment and nothing else. If source code were written in Java, it would have no real way to communicate with the environment. MiniScript is a class that reads MiniScript source code and executes it. The constructor is show below:

```
public MiniScript(String source, MiniScriptDictionary library) {
```

The MiniScript class accepts two arguments. The first argument is the source code of the MiniScript file. The second argument is the library that the MiniScript language uses to interpret the source code. MiniScriptDictionary is the interface that is used to represent a MiniScript Library. Any class that implements MiniScriptDictionary can communicate with a MiniScript file. For example, the MiniApp class implements MiniScriptDictionary. As a result MiniApp must specify an array of commands that makeup its library. In addition, any MiniApp must override the

```
public void execute(Command command)
public int getInt(Command command)
public boolean getBoolean(Command command)
public double getNumber(Command command)
```

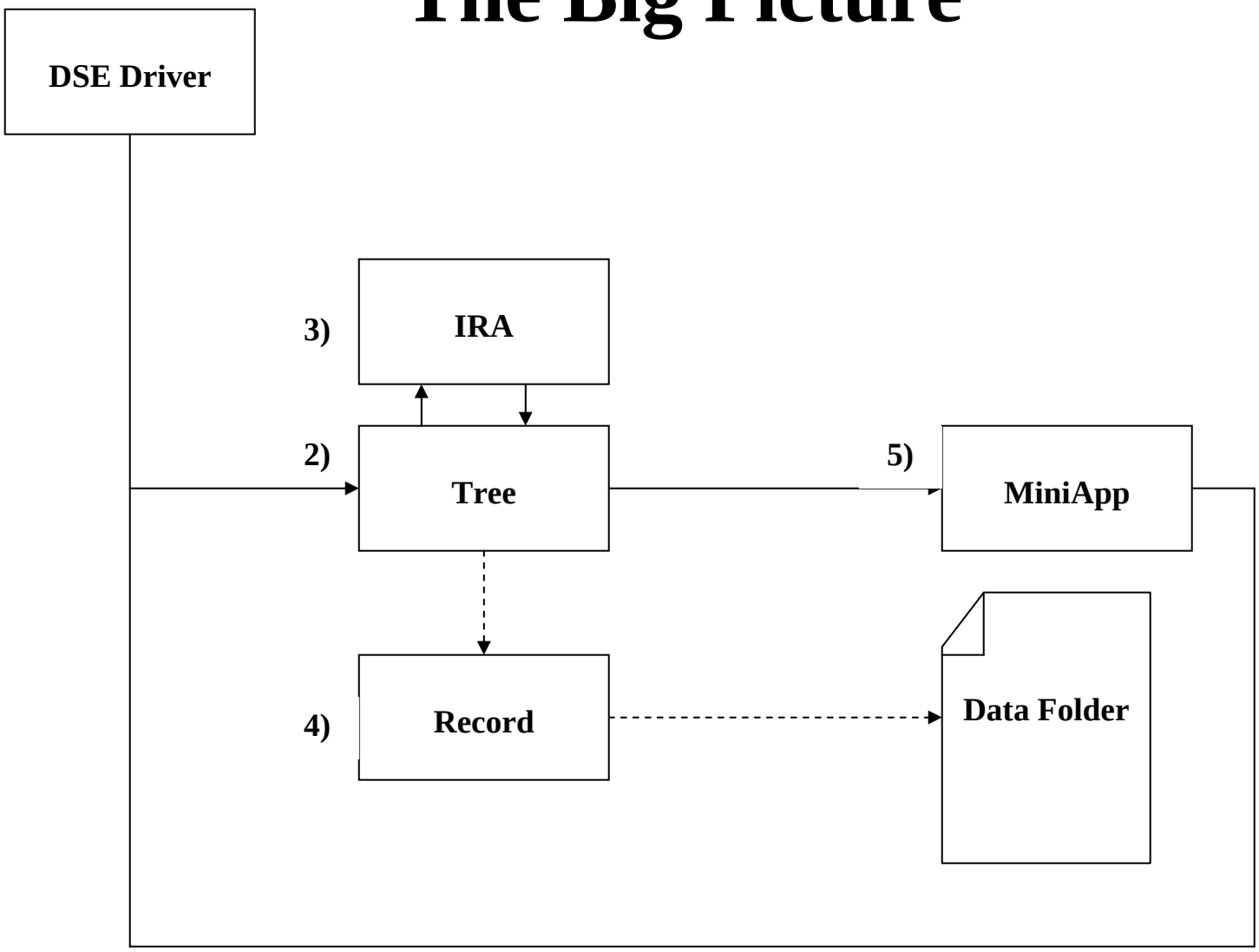
These methods are called when a command is executed. If the command has a return type of an integer that the getInt() method is called etc. This allows the environment to be notified by the MiniScript Class when a command is executed. The following is sample MiniScript code from the maze program:

```
if [wall_ahead]
turn_left
else
move
end
```

This code is a fairly primitive strategy to get through a maze. If an organism had code like this then I would likely receive a decent success percentage.

The last essential part of evolution is variation. Variation is what allows change in an environment. If all organisms were copied perfectly, then there would be no change throughout the execution of Dynamic Software Evolution. Change is essential therefore variation is essential. The Imperfect Replication Algorithm (IRA) represents variation in our program. The IRA class is the heart of variation. The IRA is in charge of copying one organisms code into a new organism. The replication of the code must be imperfect every so often in order to simulate variation. One of the greatest features of the IRA is that it will change code, but it will make sure that the new code follows MiniScript syntax.

The Big Picture



1.DSE Driver - execution of the program begins here. The DSE Driver is responsible for creating generation zero (the initial generation) and sending it to the Tree.

2.Tree - responsible for modeling the genealogy of each organism (see Figure 2). After receiving a success percentage from the MiniApp, its possible to determine how many offspring an organism will have. The more successful the organism, the more offspring it will have.

3.IRA (Imperfect Replication Algorithm) - after an organism has completed its life cycle it will reproduce. The IRA is responsible for replicating the genetic instructions contained

in each organism. In order to simulate variation, the copy of the organism's genetic instructions might not be identical to its predecessor.

4.Record - responsible for recording each individual organism that has completed the MiniApp (Maze) and storing it in an external file.

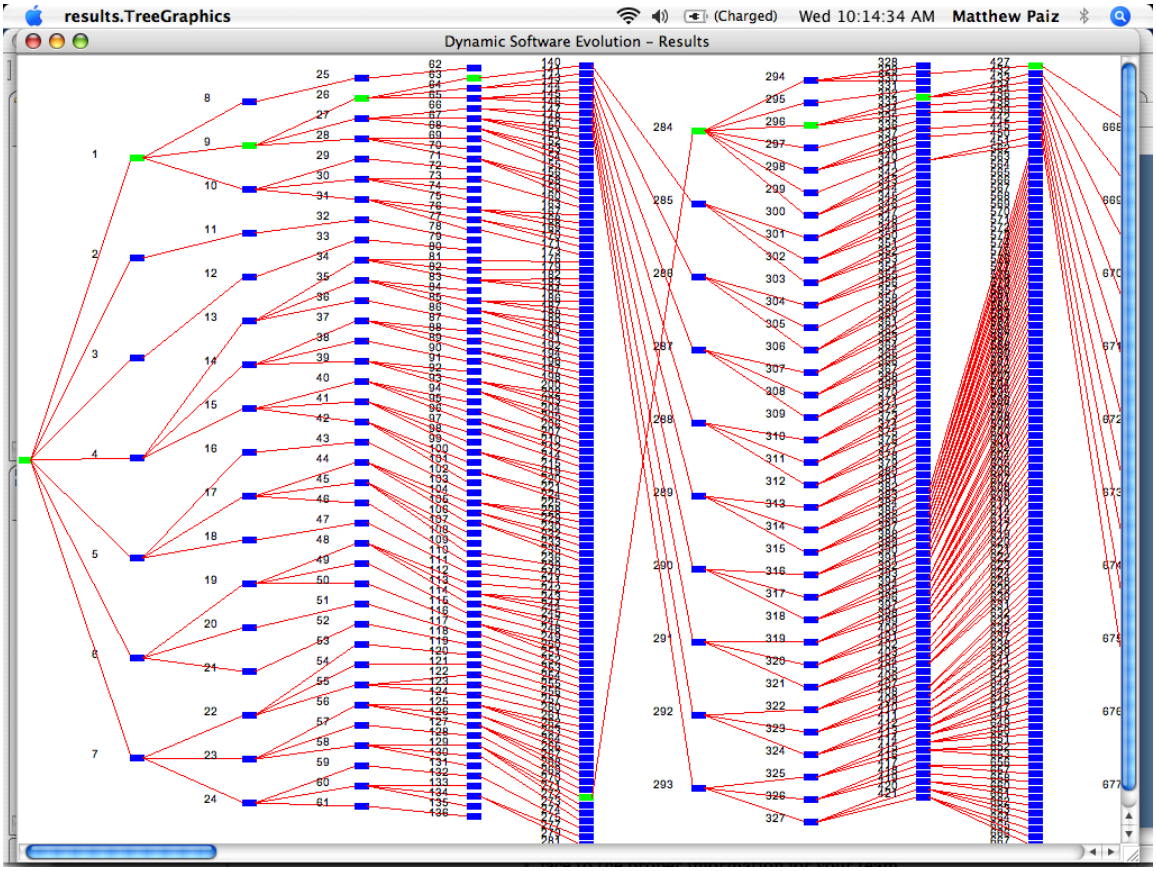
5.MiniApp (Maze) - the MiniApp is responsible for simulating the environment that the organisms live in. After an organism is sent to the MiniApp, it is executed and given a score based on its success in the environment. After this takes place the organism is sent back to the Tree where the process begins again.

Results

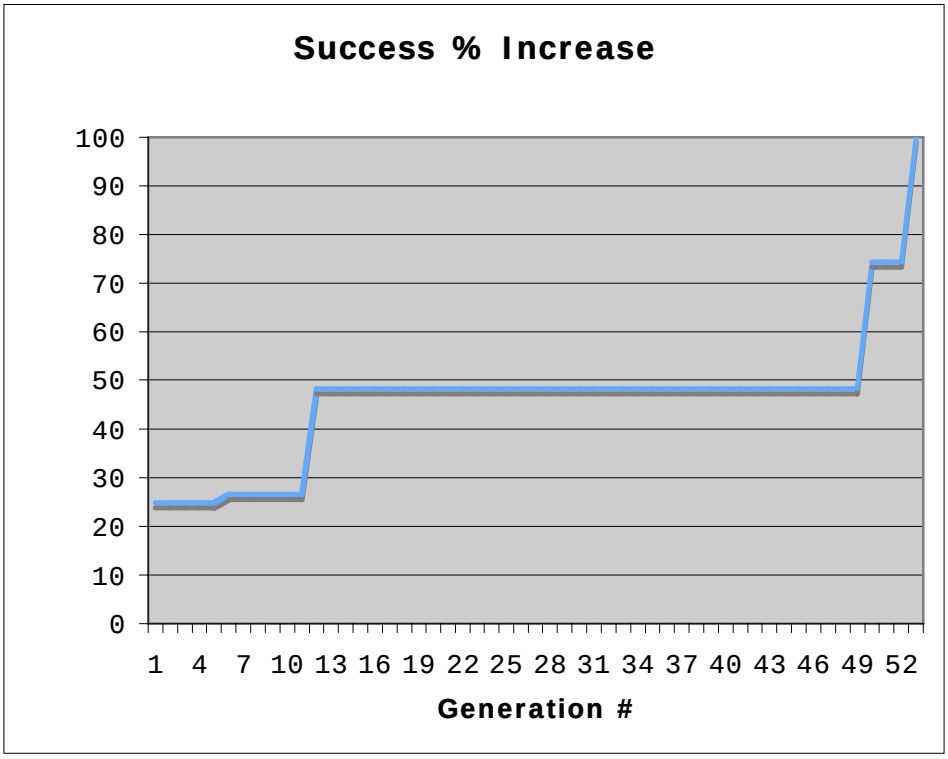
The results that Dynamic Software Evolution receives are displayed on an output that looks similar to a family tree. Each organism is represented by a rectangle. Each organism contains an ID# that is displayed to the left of the organism. Each rectangle can be clicked on with mouse. After clicking on a rectangle, a window will appear revealing the organism's properties. Some of these include: source code, ID#, parent ID#, and success percentage. The reason a family tree was used to view the results was that it made it possible to see the relationships of the organisms to one another. Initially the first organism was given a simple MiniScript file, with only one command: "move". This organism progressed 0 throughout the maze. Eventually as more generations progressed the success scores begin to progress. The first generation's highest score was 24.78%. This score was given specifically to the first generation in order to stimulate population growth because if the first organism dies then there are no others after it. The score begin to grow slowly jumping to a high of 26.52% within five generations. It stayed constant until generation eleven where it had a huge jump to 48.26%. It stayed constant until generation forty-nine where it jumped again to 74.34%. It stayed constant again for four more generations until it finally reached 100% at generation 52. The first organism to get a success score of 100% had the following source code:

```
if [wall_left]
turn_right
while [false]
end
else
turn_left
move
end
move
```

At first look, this source seemed incorrect. After plugging it back into the Maze the Organism successfully completed the Maze! What began as just “move” on the first organism, evolved into an algorithm that solved the maze by the 3501st organism. Dynamic Software Evolution was successful, it figured out how to solve a maze without being told anything but to move in a straight line.



Dynamic Software Evolution Results



Graph of Max Success % for every Generation #

Analysis

The results that we received indicated that it is possible to evolve a program. By looking at the relationships between the organisms it was possible to determine that the IRA correctly simulated variation between the organisms. Clicking on an organism, looking at its code and then comparing it to its parent could determine this. For example, a parent may have source code that looked like:

```
while [not [wall_ahead]]
move
turn_left
end
```

and its offspring had code that looked like:

```
while [not [wall_ahead]]
move
turn_right
end
```

The results also showed the advantages of having a high success score. Organism with low success scores may only have one offspring but organism with high success scores could have five or more offspring. Once an organism had a lot of offspring, its genes became more common in the simulation, causing other programs to inherit the organism's success. After improving the success scores over time, eventually the ideal program will be generated. Looking at the graph, this can be clearly illustrated.

Conclusion

Even though our program has successfully evolved, we have barely scratched the surface of artificial intelligence. We have proven that software evolution is possible on a small scale. All that's left is to prove it on a large scale. We hope to continue work on this project up until the supercomputing presentations. We are currently working on a wide range of MiniApps that can evolve the answers to equations and even play simple games like Hang Man or Tic-Tac-Toe. We also hope to make MiniScript object-oriented, in an attempt to give more power to the Dynamic Software Evolution. We were expecting our program to work, but we were not expecting to get the extraordinary results that we received. Hopefully some day it will work well enough to benefit technology.

Greatest Achievements

Clearly the greatest achievement of our project was the creation of a self-learning program. Besides that, probably the greatest achievement that was accomplished during this project was the creation of a reusable evolution utility. Because of plug-in environments and MiniScript it's possible for us to have a wide range of evolutionary experiments without changing the source code of Dynamic Software Evolution. Another great achievement is the creation of the programming language, MiniScript. Its very flexible and it can be reused in a number of programming projects not related to Dynamic Software Evolution. Its simplicity and integration with the IRA would benefit a number of programmers in the future.

Acknowledgements

- Kerrie Sena, who kept us on track and supported us throughout the entire project.
- James Carrie, who also kept us on track and gave us advice on a number of topics concerning our project.
- Los Alamos National Labs, who gave us permission to use their supercomputer.
- Celia Einhorn, who answered any questions we had during the project.