The Genome Generator

Jónas Friðrik Steinsson

Supervisor: Dr. Andrew Brooks

Faculty of Information Technology,

University of Akureyri

Submitted in April 2004, in partial fulfillment of the conditions of the award of the degree B.Sc.

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature________________________________

Date 16. Apr. 2004

Abstract

The aim of the project is to create an extended Java based version of the Genome Generator software, originally created in the year 2000 by Robert W. McGrail, Rebecca Thomas and Michael Tibbets. The Genome Generator is used to simulate and display information on the development of junk DNA. In order to achieve that aim the Genome Generator displays various information on genomic build up derived from given user-input requirements.
This document addresses the motivation for the work done, describes the work and addresses related work which has been reviewed during the process.
This document describes the ideas behind the software, the underlying algorithms and the process of development. This document will describe the software developed, it's intended usage and capabilities.
The implementation of the software is discussed and various design issues that had to be taken into account during the development process are discussed.
Evaluation of the software is discussed and future testing and development ideas are described.

The Java version of the Genome Generator has been in development since early October 2003. The first months of the project were spent analyzing and evaluating the existing software, which is written in Scheme, and the research surrounding the creation of that generator. The design of the algorithm and user interface started in October 2003 and in January 2004 the first version of the user interface was ready. The first version of a functioning generator was ready in late February 2004. Upon reviewing the algorithm used some glitches were discovered and a complete reconstruction of the software proved necessary. These issues will be further discussed in the chapters on design and implementation.

Due to various factors the implementation of a graphical display of the statistical output was somewhat hindered. Several graphical packages available over the Internet were

tested and evaluated, but none of them proved adequate. The idea was to use as much of freely available software as possible, to be able to keep all aspects of this software free and therefor keep the software itself freely available to those that might benefit from it. However some of the software originally considered, proved to be not as freely available as seemed at first. These issues will be further discussed under the section on implementation.

Table of Contents

Abstract.......... 2
Motivation for the work.......... 5
Description of the work.......... 6
Related work.......... 6
Design.......... 10
    The Genetic Algorithm.......... 10
    The GUI (graphical user interface).......... 13
    The Graphical Information Display.......... 16
Implementation.......... 19
Evaluation.......... 20
References.......... 21

Illustration Index

Illustration 1: The original Genome Generator User Interface 7
Illustration 2: The original Genome Generator graph window 8
Illustration 3: Genome Generator Class Diagram 11
Illustration 4: The Genome Generator GUI 14
Illustration 5: The Genome Generator Save to file dialogue 15
Illustration 6: The Genome Generators Statistical information box 15
Illustration 7: the comparison methods choice menu 17
Illustration 8: Graph comparing Genes per Genome with generations 17
Illustration 9: Graph comparing Genome length and generations 18
Illustration 10: Graph comparing Coding Region Percentage with generations 18

Motivation for the work

One of the great mysteries of biology is the structure and development of genes. We have known of the existence of genes and their role in evolution for some time, but how do genes evolve, and how much of the genome string is actually being used at a given period in time?
Monitoring changes in DNA structure can be difficult, especially when we keep in mind that genetic changes usually take several generations, if not hundreds of generations. Thus it is necessary to model genetic evolution, to be able to monitor the behavior of genes over time.

The Genome Generator project is meant to simulate the development of DNA, using genetic algorithms. The software is aimed at students of basic biology and biochemistry, as well as more advanced researchers. Therefore the appearance and usage of the software is kept simple and straight forward, while allowing for changes in the variables used for the simulation.

The Genome Generator project is a structural remake of software originally published in the year 2000. The original software is somewhat limited in the sense that once the simulation has run for a number of generations the software tends to freeze up and halt due to insufficient memory. The original software also limits the users control over simulation variables such as the length of the edited genome string, the original genome string used and the increase rate of genome viability, which controls the growth of the genome string by increasing the required number of legal genes in the edited string.
This project is intended to reconstruct and extend the earlier software, giving the user more control over the variables used during the simulation, and thus the simulation itself, as well as extending the statistical data derived from the simulations.

Description of the work

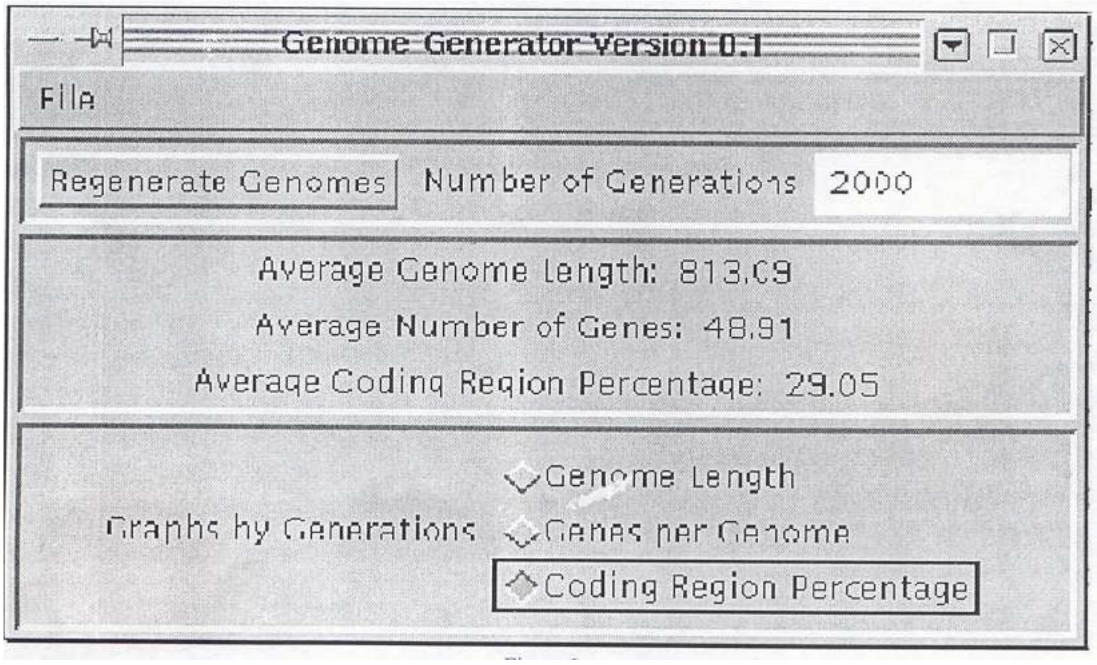
The motivation for this project is to extend the original program, which was written in the low level programming language Scheme. This re-implementation is done using Java as the base language. A secondary aim of the project is increasing the users control over the variables mentioned above, while still maintaining the simplicity and functionality of the original software.

The original idea behind the Genome Generator is to create software to simulate the growth of a genome string through several generations. However this reconstruction implements several changes in comparison to the original software, mainly in terms of the user environment, which has been extended to allow the user to access and manipulate various simulation variables. This is done in consensus with the original authors ideas for extending the software, as stated in their project paper (McGrail et.al.).

Related work

The Genome Generator project is a rational reconstruction of an earlier software project created by Robert W. McGrail, S. Rebecca Thomas, Michael Tibbetts at the Division of Natural Sciences & Mathematics at Bard College.
In their project paper McGrail et. al. state that *"While there are obvious similarities between this program and genetic algorithms, our goal is entirely different. Rather than using an evolutionary model in order to solve some optimization problem, we are instead simulating a very simple model of molecular evolution. So, for example, we do not have multiple individuals in each generation. We check only for viability, rather than for any measure of fitness, and we have no operation comparable to either crossover or transposition"*. (McGrail et. al).

The original Genome Generator is a very simple and easy to use program. The Genome Generator is created using Scheme, a low level programming language which uses simple syntax and semantics. The program code is very simple and uses low-level methods and algorithms. The software user environment is very simple, but provides the user with all the necessary controls for this simplistic program.



Number of Generations 2000

Average Genome Length: 813.09

Average Number of Genes: 48.91

Average Coding Region Percentage: 29.05

Graphs by Generations

Illustration 1: The original Genome Generator User Interface

The user interface allows the user to input the desired number of generations generated and choose between three types of comparison methods, genome length per generation, genes per genome for each generation and coding region percentage for each generation. The software displays statistical information on the average genome length, the average number of genes and the average coding region percentage, within the user interface, as well as displaying the statistical comparison as a simple plot graph in an external window.

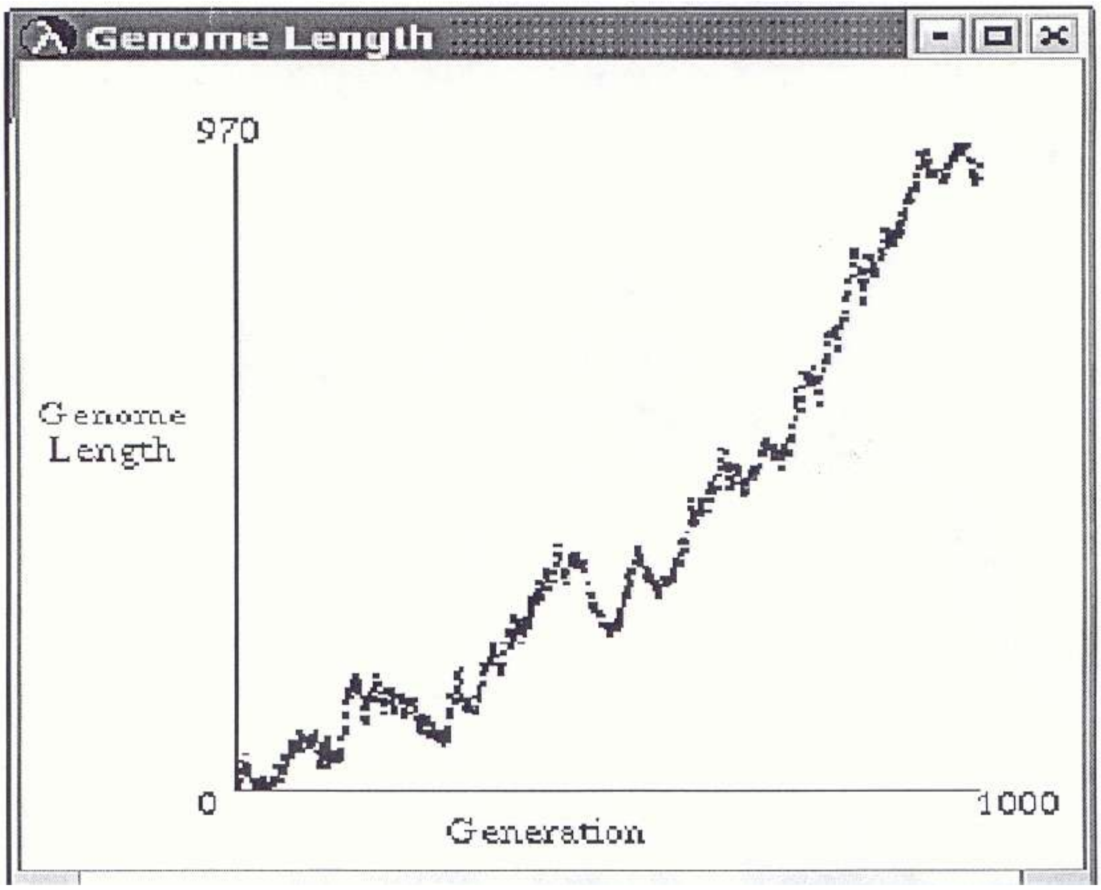


Illustration 2: The original Genome Generator graph window

The current software is not a stand-alone application, and must be executed from within Dr. Scheme, which compiles and runs the source files.

The authors of the original program state that their work is based on the research done by William F, Loomis and Michael E. Gilpin, which was published in their paper titled *'Multigene Families And Vestigial Sequences'* in 1986.
In their research, Loomis and Gilpin simulated the development of junk DNA using a computer generated simulation. They observed that a random number of duplication and deletion sequences create genomes carrying a large amount of dispensable sequences.
In their paper Loomis et.al. State that "*Many related DNA sequences, such as those coding for a- and f3-globins, produce proteins with related functions, while other related sequences have sustained deleterious mutations or deletions such that they are no longer functional and are referred to as pseudogenes*" (Loomis et. al.).

The simulations run using The Genome Generator support the findings of Loomis and Gilpin and assert their suspicions that complex DNA such as the human DNA contains huge amounts of non-functional genes or codons and further more that "when the size of the genome is not critical to survival, as appears to be the case within limits in most eukaryotic organisms, the genome carries vestigial sequences that are no longer functional and that many genes are present in multigene families by chance"(Loomis et. al.).

Although the work of McGrail et.al. is based on the research by Loomis and Gilpin, there are some fundamental differences in the approaches used for simulations.
In both cases each event consisted of a randomly chosen duplication or deletion of a substring of the genome, 1-20 units in size. However in the work of McGrail et.al. a third operation, reversal, had been added to the simulation. This operation adds to the variety of operation and therefor it is used in the Java version of the Genome Generator.
The position of the event was randomly generated within the genome in both cases and when a duplication occurred, the new sequence was positioned next to the existing sequence.

Loomis and Gilpin represented a vital gene was as 4 units consisting of 1 start unit (promoter), 2 coding units, and 1 stop unit (terminator). In the original Genome Generator, any number of coding units, larger than one, preceded by a promoter and followed by a terminator is legal. With regard to the coding region percentage of each generation, the method followed by McGrail et.al. delivered a much higher number then the original method. However where genes consist of pairs of codons, it was decided that the Java version of the Genome Generator would follow the former method and allow only genes consisting of a promoter, two codons and a terminator.

The initial genome in both cases contained only a single gene represented by a promoter, two codons and a terminator.
In the work of Loomis and Gilpin, deletions in only a single copy of a gene were considered lethal and terminated the evolution of that genome. However in the work of

McGrail et.al. such an operation results in the software looping around and choosing a different operation on the gene. The Java version of the Genome Generator follows the latter method.

Design

This section covers an overview of the design. It will focus on the proposed project and the tools that will be used and how they will help in converting and expanding the Genome Generator. In conclusion this document will show how the proposed user environment of the Genome Generator might look.

The Genome Generator is designed using Java. This is done due to the fact that Java is platform independent and easy to implement over a network. One of the ideas surrounding the Genome Generator is that in the future, the software could be internet based and the user environment could be an applet on a web-page. Another reason to implement this project using Java is that as an algorithmic language, Java is quite restricted, so this introduced a challenge. However, graphically, Java is a premier language so all the graphical implementation is easier in Java then many other languages, especially languages such as Scheme or Lisp, which are more often then Java used for simulation purposes.

The design of the Genome Generator can be divided into 3 structural parts, the genetic algorithm, the graphical user interface, and the graphical information display. As a result the author will discuss each of these parts individually.

The Genetic Algorithm

The genetic algorithm is the heart of the software. This code is responsible for all calculations and string manipulations needed for a successful simulation. This part of the software was the most time-consuming and had to be reconstructed twice during the process of development, due to slight miscalculations and misunderstandings regarding implementations in the original research.

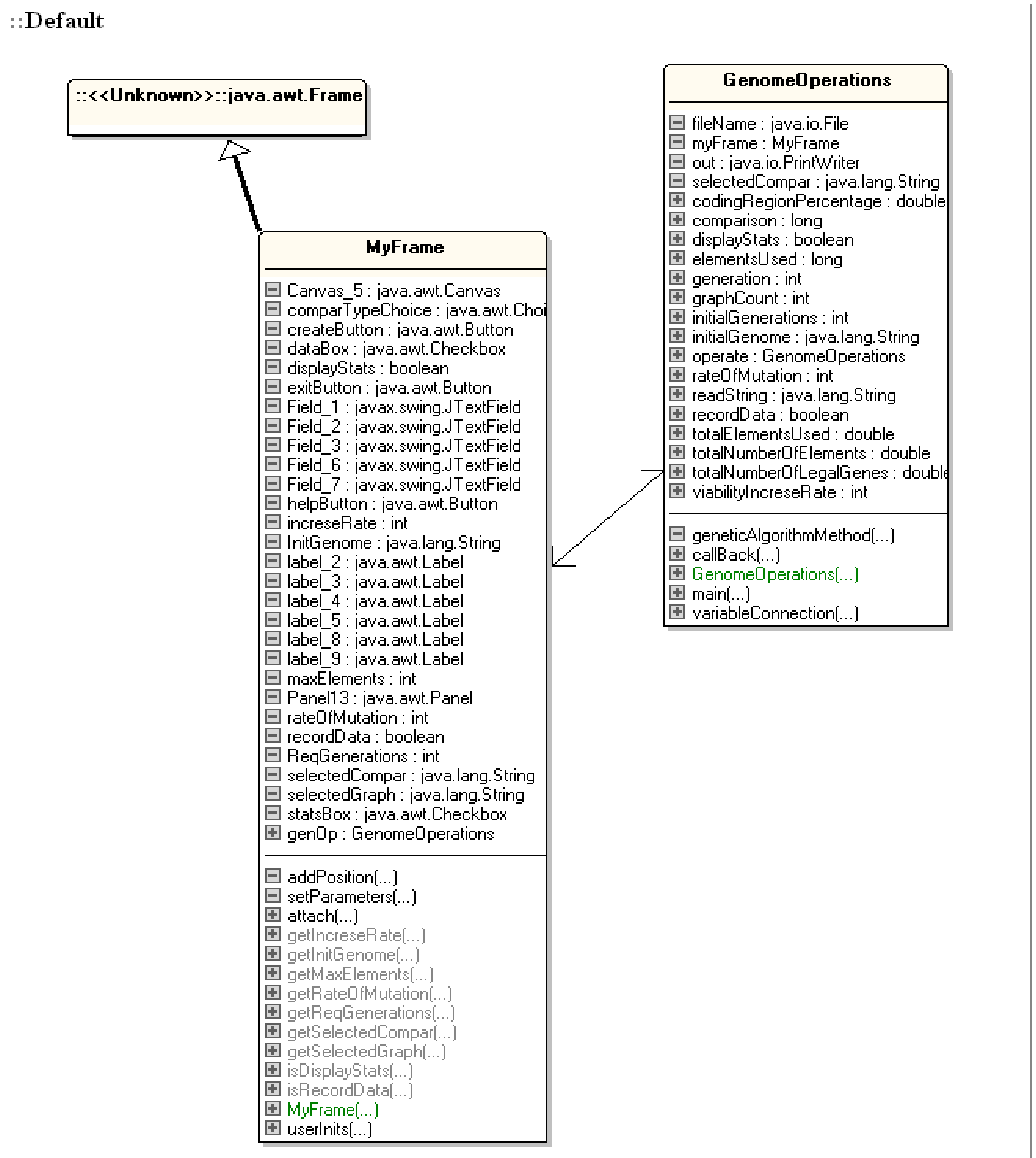


Illustration 3: Genome Generator Class Diagram

The genetic algorithm can be divided into several distinct segments, substring selection, substring genetic manipulation, substring mutative manipulation and string verification.

The substring selection part of the algorithm selects a substring of the genome being manipulated, and ensures that the substring is of legal length, not shorter than one element and not longer then the desired number of elements, or if the genome has less elements in it than the desired length, not longer than the genome's length. The length of the substring is arbitrarily set to 20, as in the original Genome Generator, but this number can be manipulated by the user through the user interface.

The genetic manipulation part contains the necessary elements for manipulating the selected substring in three ways: Insertion, deletion and reversion. These operations are the same as in the original program.

The insertion part of the algorithm copies the selected string and concatenates the copy onto the selected string, creating a new string consisting of these two copies of the string. This new string is then inserted into the genome to replace the original substring chosen. The deletion part of the algorithm simply deletes the chosen substring from the genome. The reversion part copies the substring chosen from the genome, inverts the substring and then replaces the original substring with the inverted copy.

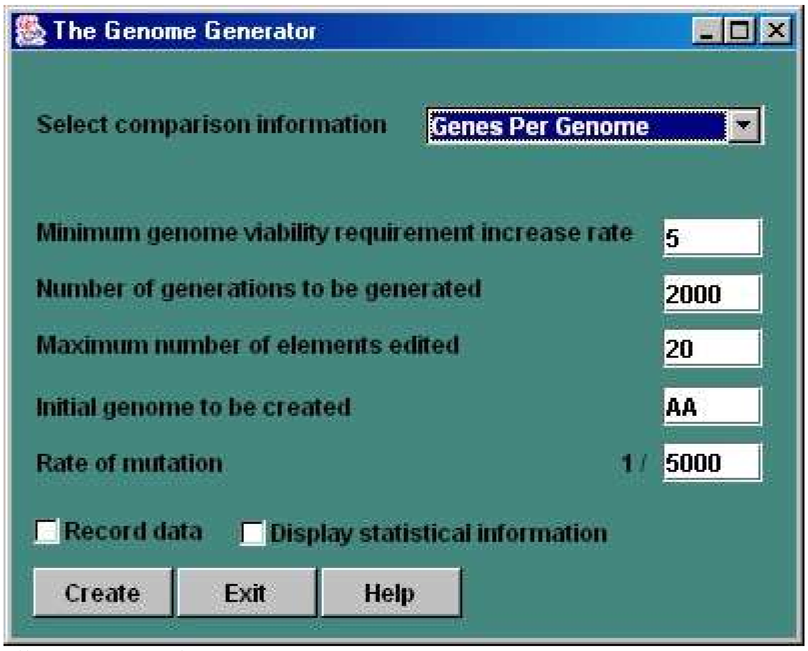
In the case of deletion and reversal operations the substring is read into a string buffer, and manipulated from there. This simplifies the operations required for the deletion and the reversal of the substring, saving on calcuational effort and code.

The mutative manipulation section of the code checks every element of the code in turn and individually decides if that element should be mutated, using a probability which is originally set to 1/5000, but can be manipulated by the user through the user interface. If an element is marked for mutation the algorithm runs through the following procedures: If the element to be mutated is not a codon in the genome, the mutated element is converted to a non functional element, to indicate that this element is not a part of a legal gene. If on the other hand, the element is a legal codon, the algorithm chooses between two equally likely operations, a case-change of the element, from upper case to lower case or vice versa, or a mutation to a different element, retaining case. Where as the original program used only characters from the English alphabet to indicate legal codons, this reconstruction of the software does the same. The mutation between elements calls upon the next element in the alphabet to replace the element to be mutated, with the exception that if such a mutation occurs once the alphabet has reached its end, the element mutated reverts to the first element in the alphabet. In this case the alphabet only contains 11 elements from A to K due to restrictions in the Character.forDigit() method of Java, which allows us to use integers to represent characters from the alphabet.

Once all manipulations on the string have been performed, the string verification section of the code adjusts the string for legality, the code creates a second string of equal length to the genome string consisting only of junk codons. All legal genes are then copied over to the new string, leaving all illegal genes behind. The code keeps count of all the elements copied between the strings and evaluates if the resulting genome string is illegal or not. If a genome has no legal genes, it is considered illegal, and discarded. The number of legal genes increases according to the minimum genome viability requirement increase rate, which is by default set to 5, meaning that if a genome has held it's legal gene-count above the previous requirement for 5 consecutive generations, the requirement is increased by one ensuring that evolution takes place.
Each legal genome is recorded in a file which is by default named 'Genome.rtf' and is stored in the programs root folder. The user can choose to store the results in a different location before running the simulation.

The GUI (graphical user interface)

The graphical user interface is designed with simplicity in mind. Through simple choice menus the user can select the method method of comparison used for displaying the information gathered through the simulation. Simple text boxes give the user access to the variables used for the simulation.
The basic graphical user interface was generated by Visual Drafter® for Java which is a free GUI generator. All functionality of the interface had to be implemented manually.



Select comparison information: Genes Per Genome
Minimum genome viability requirement increase rate: 5
Number of generations to be generated: 2000
Maximum number of elements edited: 20
Initial genome to be created: AA
Rate of mutation: 1 / 5000
☐ Record data
☐ Display statistical information

Illustration 4: The Genome Generator GUI

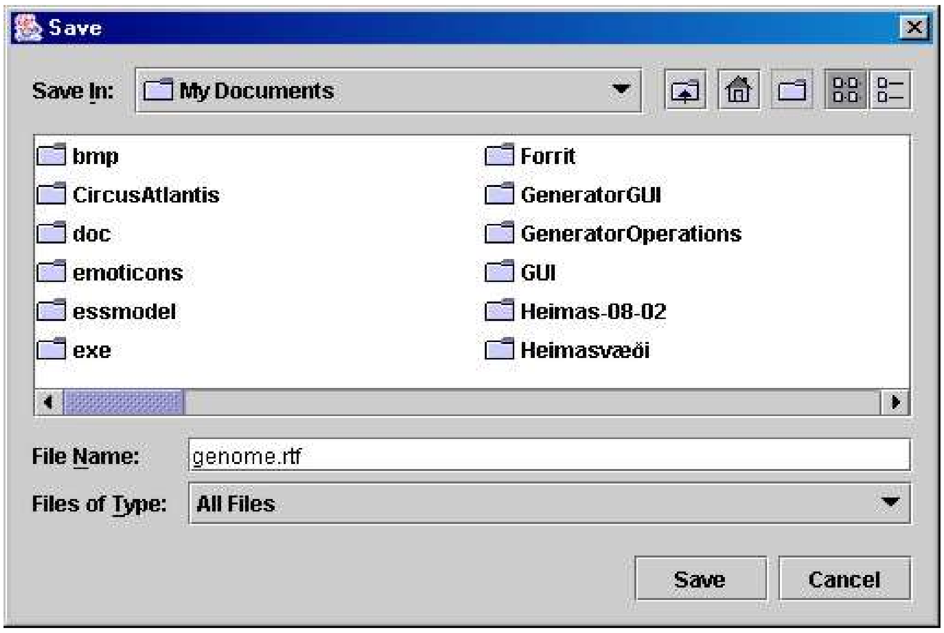
The 'Minimum genome viability requirement increase rate' textbox allows the user to change the number of generations the simulated genome must hold the minimum number of legal genes before that minimum number is incremented. Minimum genome viability requirement increase rate is arbitrarily set to 5.

The 'Number of generations to be generated' allows the user to change the number of generations used in the simulation. Number of generations to be generated is arbitrarily set to 2000.

The 'Maximum number of elements edited' textbox allows the user to alter the maximum length of the genome substring to be edited. Maximum number of elements edited is arbitrarily set to 20.

The 'Initial genome to be created' allows the user to change the elements used for the initial genome, no more than 2 elements can be used as initial gene. The initial genome to be created is by default 'AA'.

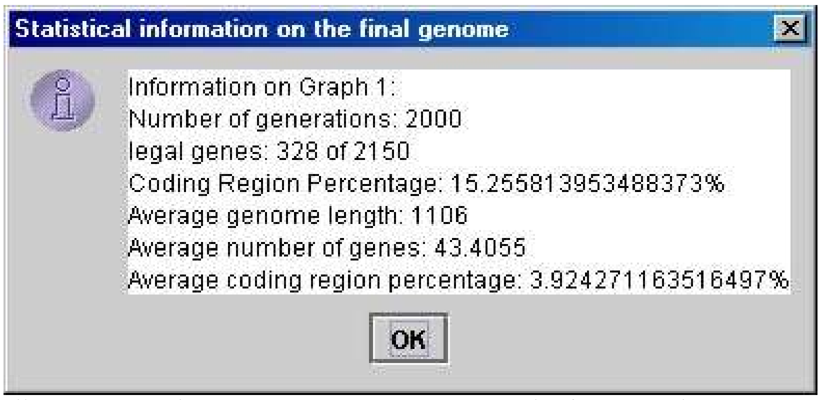
The 'Rate of mutation' textbox provides the user with access to alter the probability of mutation per element for every generation of the evolutionary simulation. The rate of mutation is set to 1/5000 by default.



File Name: genome.rtf
Files of Type: All Files

Illustration 5: The Genome Generator Save to file dialogue

Two checkboxes allow the user to chose if the data generated by the simulation should be recorded to an external file, and if statistical information should be displayed after the simulation has terminated.



```
Information on Graph 1:
Number of generations: 2000
legal genes: 328 of 2150
Coding Region Percentage: 15.255813953488373%
Average genome length: 1106
Average number of genes: 43.4055
Average coding region percentage: 3.924271163516497%
```

Illustration 6: The Genome Generators Statistical information box

To keep the graphical user interface as clean as possible the statistical information is displayed in an external window.

The Graphical Information Display

The most problematic part about the Genome Generator was the Graphical information display. Several freely distributed, but quite professional graphs packages were evaluated. But in the end there were two packages that stood out.

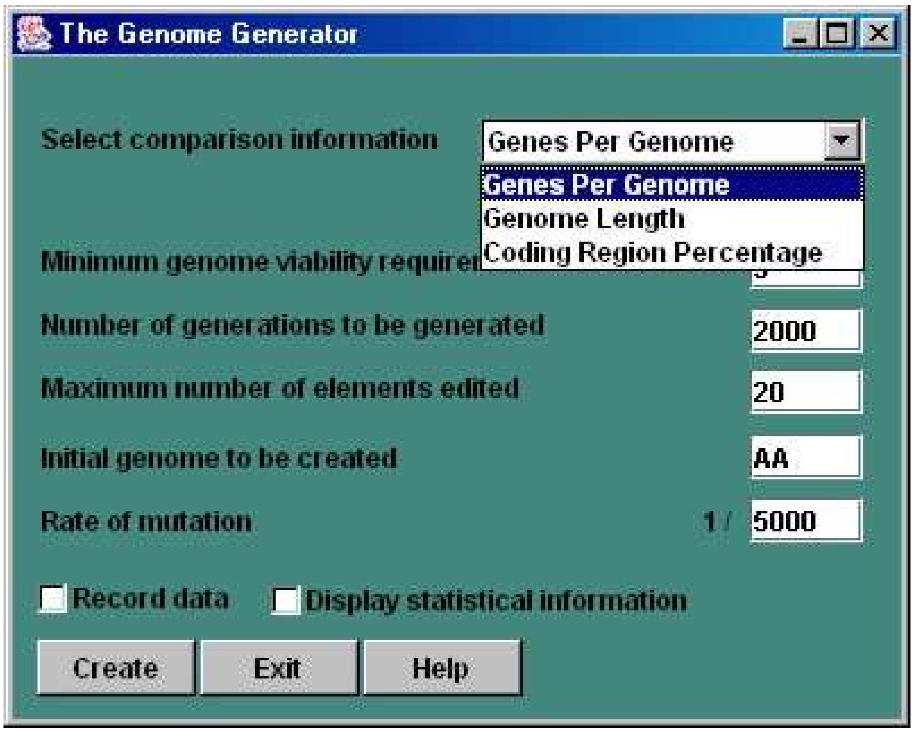
JfreeChart is a professional graphs package that is downloadable freely from the internet and ready to use. However when all that was left was to link it to the project, a problem regarding the user manual accompanying the package arose. Upon further investigation it became clear that large sums of money had to be paid to gain access to such a manual from the authors.

JCCKit is a different package which is mainly meant for use over the internet. However it is relatively easy to use, completely free of charge and comes with the user manual. However due to problems with the algorithm class during these last couple of days, this package could not be instantiated, where the project suffers from a lack of usage examples and sample code.

JavaGently's graph class is the one that finally got used. It is freely available on the internet and the book displaying examples of use was on my shelf.
The package is easy to use and very simple, however some editing of the Graphs class was needed before it suited the purpose of this project.

The user can choose between three comparison methods. These display the gene count per genome, the length of the genome or the coding region percentage on the vertical axis. All graphs use the number of generations as a comparison on the horizontal axis.

There is no need to close the graphs between runs where several graphs can be open at the same time for comparison. All data is re-calculated every time that a simulation is run.


The Genome Generator

Select comparison information: Genes Per Genome

Minimum genome viability requirement

Number of generations to be generated: 2000

Maximum number of elements edited: 20

Initial genome to be created: AA

Rate of mutation: 1 / 5000

Illustration 7: the comparison methods choice menu

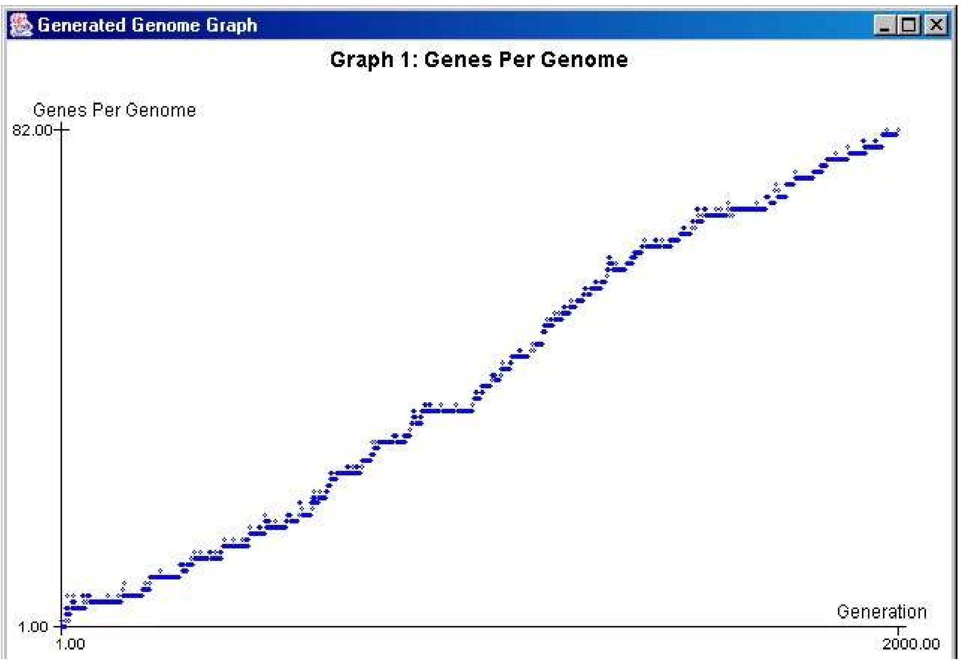


Illustration 8: Graph comparing Genes per Genome with generations

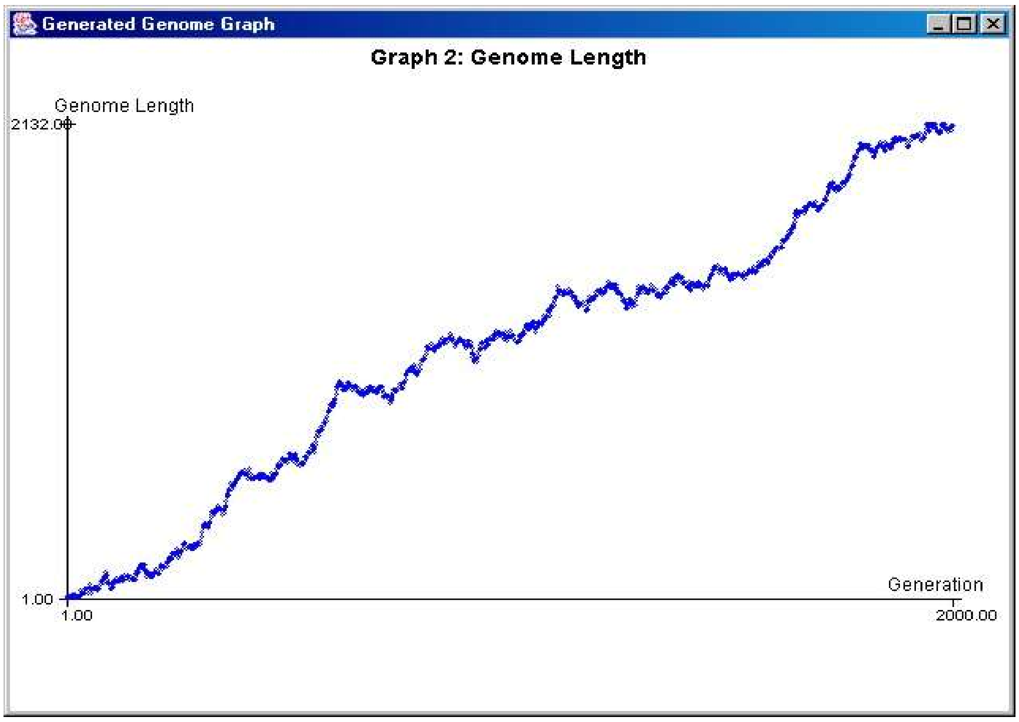


Illustration 9: Graph comparing Genome length and generations

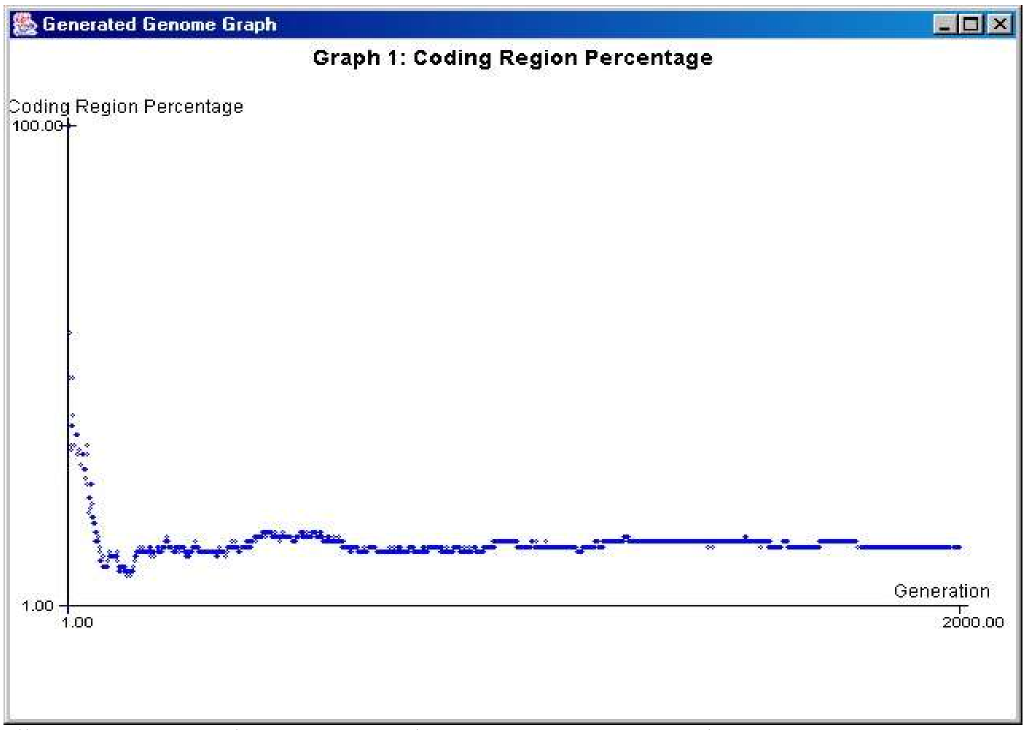


Illustration 10: Graph comparing Coding Region Percentage with generations

Implementation

The Genome Generator is written in Java using IntelliJ Idea. It makes use of the Java 1.4.1 environment and should run on any operating system available. As stated earlier, Java is not the most logical language to implement simulation software in, due to the fact that logic is more difficult to implement in Java then in specialized AI languages such as Scheme and Clipse. However Java has very good graphical capabilities and is easy to implement over network.

During the research stage of the project, my whole idea about the implementation of the graphical part of the software changed rapidly. At first my idea was to implement a 3 dimensional model of a gene, which the user would be able to see growing. However once I had realized how huge a genome is, this idea was calmly set aside to make room for more realistic implementation. Another factor that played a big part in the way the project turned out to be implemented was total lack of time. My research was delayed somewhat, due to the fact that I had to get Loomis and Gilpin's research article sent from Denmark. After studying their research article and the original Genome Generator program, I found that simplicity was the key. During my weekly meetings with my project supervisor Dr. Andrew Brooks, I found that he agreed with me, that a rational reconstruction of the program was the first step, and that any additional functionality would be a bonus.

During Christmas holidays I made some progress on the GUI and the algorithm, however I found that the Graphs package that I had chosen to use, JFreeChart, had no user manual, and there were no code-samples or detailed instructions accompanying the package. Upon a visit to their web-page I found that a detailed user manual exists for the price of 260 dollars. Following a meeting with my project supervisor Dr. Andrew Brooks I started searching for a new graphical package to use.

In February 2004, I was told of the JCCKit graphical package. Due to other projects and slight errors in the algorithm I had developed, I needed to postpone all experiments with JCCKit until the end of march 2004. In the beginning of April 2004 I had completed the reconstruction of the algorithm. By that time I started to work on connecting the user environment to the algorithm and try to connect the graphs package as well. I soon found out that the JCCKit package is mainly meant for internet usage and is also very poorly documented. On the afternoon of the 15th of April I decided to use the JavaGently graphs-

class instead, where I am familiar to it and have access to all material relating to it.
The original idea was to be able to show dotted graphs, line-graphs and dot and line-graphs, but there seems to be a glitch in the JavaGently graphs-class that only allows me to display dotted graphs.

Evaluation

So far only basic tests have been run on the software, but it has passed most of them. It passes all tests based on the original program data. It has been run for up to 50.000 generations and all individual functions have been tested.

Graphical printouts have been compared with original data from Loomis and Gilpin and the graphs generated by the software have been compared with graphs generated by the original Genome Generator. Some difference can be seen when data regarding the coding region percentage is compared, but that is natural, where the Java version of the Genome Generator only allows four elements per gene, but the original Genome Generator allows four and more elements per gene.

There seems to be a problem with the JavaGently graphs-class regarding small numbers. If I run a test using less then five generations the graph seems to print out wrong. Sometimes the axis are upside down and sometimes the graph window displays an error message.

I have talked to the headmaster of my old grammar-school and she wants to take part in stress-testing this program, so intensive testing may start next semester.

References

1. Robert W. McGrail, S. Rebecca Thomas, Michael Tibbetts. *The Genome Generator: Simulating the development of junk DNA*. Division of Natural Sciences & Mathematics Bard College Annandale-on-Hudson.

2. William F, Loomis and Michael E. Gilpin. *Multigene Families And Vestigial Sequences*. Department of Biology, University of California at San Diego, La Jolla, CA 92093 Proc. Natl. Acad. Sci. USA Evolution Vol. 83. pp. 2143-2147. April 1986

3. Claes Sterner, 2003. *Visual Drafter® for Java®.*[online]. Available at: <URL: http://www.users.wineasy.se/cst/vd/VDcommon.htm> [Accessed 28 November 2003]

4. JetBrains, 2003. *IntelliJ IDEA®JDevelope with pleasure* [online]. Available at: <URL:http://www.intellij.com/idea/> [Accessed 28 November 2003]

5. JFreeChart, 2003. *JFreeChart® project homepage* [online]. Available at: <URL:http://www.jfree.org/jfreechart/index.html > [Accessed 28 November 2003]

6. JCCKit, 2004. *JCCKit project homepage* [online]. Available at: <URL: http://jcckit.sourceforge.net > [Accessed 20 January 2004]

7. JavaGently, 2001. JavaGently homepage [online]. Available at: <URL: http://javagently.cs.up.ac.za/jg3e/ > [Accessed 15 April 2004]

8. Judith Bishop. *JavaGently, Third Edition.* Addison-Wesley. 2001