Dissertation

# Ted Gress

ABSTRACT

AN AIM TOWARDS COMPUTATIONAL INTELLIGENCE BY

MUTATION OF MULTIPLE SOURCE CODE

By

Walter Gress

Master of Science in Computer Science

Everything begins with Proof. Every Proof begins with Theorems. Every theorem begins with Axioms. Every Axiom begins with an idea. Every Axiom begins with an Idea, a Concept that is verifiable. Every Concept begins with am Observation.

Self Producing Genetic Pool Theorem

Given an environment (pool), a sufficient number of operations, and sufficient time, a logical program will generate from individual constituent keywords.

Genetic Mutating Application (GEMA) is an application in which one or more working pieces of viral code are input. In this case , the viral code are custom designed so that the various viral programs infect each other in a rate at random. Once one program is infected, the language of that program (in this case Java), will go through a parse checker to see if the language is still valid. If not, it is placed back in the pool of virus programs. If still valid, we have an instance of genuine program evolution. As we progress, we hope to see more and more valid programs. We conjecture that as time goes by, the programs will become more complex. This is similar to the combination of chemicals, then amino acids, then DNA, then more complex carbohydrates, cells, etc. in mammals. It ensure we don’t just result in gibberish returned, instead we swap keywords (ie., for might be swapped with while) hoping to return better, readable, workable, results.

A monkey hitting keys at [random](https://en.wikipedia.org/wiki/Randomness) on a typewriter keyboard for an [infinite](https://en.wikipedia.org/wiki/Infinity) amount of time will [almost surely](https://en.wikipedia.org/wiki/Almost_surely) type any given text, such as the complete works of [William Shakespeare](https://en.wikipedia.org/wiki/William_Shakespeare)

The infinite Monkey Theorem

Should the infinite monkey theorem be true, life would never have evolved. Since essentially monkeys typing are similar to the primordial soup’s chemicals interacting. The constituents of life combining to form functional biology are like the monkey hitting randomly on a typewriter and producing Shakespeare. Thus if the infinite monkey theorem is bounded to a rational amount of time and consists of writing random keywords that form an organize logical corpus, so then can the self-producing genetic pool theorem, the topic of this paper. Because the self-producing genetic pool consists of random keywords that form an organized logical corpus, that is, a program.

TABLE OF CONTENTS

INTRODUCTION

RELATED WORKS

PROBLEM STATEMENT

MATHEMATICAL APPROACH

ALGORITHMS AND DATA STRUCTURES

APPROACH

TESTING THE PRIMORDIAL OOZE

DATA SET (test keyword sets - experiements)

TESTING RESULTS (results of each experiment)

CONCLUSION

REFERENCES

APPENDICES

TERMINOLOGY

SAMPLE OUTPUT

# Introduction

Alexander Oparin first postulated his theory in Russian in 1924 in a small pamphlet titled *Proiskhozhdenie Zhizny* (*The Origin of Life*).[[12]](https://en.wikipedia.org/wiki/Primordial_soup#cite_note-oparin24-12) According to Oparin, the primitive Earth's surface had a thick red-hot liquid, composed of heavy elements such as carbon (in the form of iron carbide). This nucleus was surrounded by the lightest elements, i.e. gases, such as hydrogen. In the presence of water vapour, carbides reacted with hydrogen to form hydrocarbons. Such hydrocarbons were the first organic molecules. These further combined with oxygen and ammonia to produce hydroxy- and amino-derivatives, such as carbohydrates and proteins. These molecules accumulated on the ocean's surface, becoming gel-like substances and growing in size. They gave rise to primitive organisms (cells), which he called [coacervates](https://en.wikipedia.org/wiki/Coacervate).[[7]](https://en.wikipedia.org/wiki/Primordial_soup#cite_note-lazcano10-7) In his original theory, Oparin considered oxygen as one of the primordial gases; thus the primordial atmosphere was an oxidising one. However, when he elaborated his theory in 1936 (in a book by the same title, and translated into English in 1938), he modified the chemical composition of the primordial environment as strictly reducing, consisting of methane, ammonia, free hydrogen and water vapour—excluding oxygen

J.B.S. Haldane independently postulated his primordial soup theory in 1929 in an eight-page article "The origin of life" in *The Rationalist Annual*.[[7]](https://en.wikipedia.org/wiki/Primordial_soup#cite_note-lazcano10-7) According to Haldane the primitive Earth's atmosphere was essentially reducing, with little or no oxygen. Ultraviolet rays from the Sun induced reactions on a mixture of water, carbon dioxide, and ammonia. Organic substances such as sugars and protein components ([amino acids](https://en.wikipedia.org/wiki/Amino_acids)) were synthesised. These molecules "accumulated till the primitive oceans reached the consistency of hot dilute soup." The first reproducing things were created from this soup

As to the priority over the theory, Haldane accepted that Oparin came first, saying, "I have very little doubt that Professor Oparin has the priority over me.

One of the most important pieces of experimental support for the "soup" theory came in 1953. A graduate student, [Stanley Miller](https://en.wikipedia.org/wiki/Stanley_Miller), and his professor, [Harold Urey](https://en.wikipedia.org/wiki/Harold_Urey), performed an experiment that demonstrated how organic molecules could have spontaneously formed from inorganic precursors, under conditions like those posited by the Oparin-Haldane Hypothesis. The now-famous "[Miller–Urey experiment](https://en.wikipedia.org/wiki/Miller_experiment)" used a highly reduced mixture of gases—[methane](https://en.wikipedia.org/wiki/Methane), ammonia and [hydrogen](https://en.wikipedia.org/wiki/Hydrogen)—to form basic organic [monomers](https://en.wikipedia.org/wiki/Monomer), such as [amino acids](https://en.wikipedia.org/wiki/Amino_acids).[[15]](https://en.wikipedia.org/wiki/Primordial_soup#cite_note-15) This provided direct experimental support for the second point of the "soup" theory, and it is around the remaining two points of the theory that much of the debate now centers.

Apart from the Miller–Urey experiment, the next most important step in research on prebiotic organic synthesis was the demonstration by [Joan Oró](https://en.wikipedia.org/wiki/Joan_Or%C3%B3) that the nucleic acid purine base, adenine, was formed by heating aqueous [ammonium cyanide](https://en.wikipedia.org/wiki/Ammonium_cyanide) solutions.[[16]](https://en.wikipedia.org/wiki/Primordial_soup#cite_note-16) In support of abiogenesis in eutectic ice, more recent work demonstrated the formation of s-[triazines](https://en.wikipedia.org/wiki/Triazine) (alternative [nucleobases](https://en.wikipedia.org/wiki/Nucleobase)), [pyrimidines](https://en.wikipedia.org/wiki/Pyrimidine) (including cytosine and uracil), and adenine from urea solutions subjected to freeze-thaw cycles under a reductive atmosphere (with spark discharges as an energy source)

# Related Works

# Problem Statement

# Although we have made great strides in the science of Artificial Intelligence, we still are unable to produce a piece of software we can call truly “Intelligent.” Indeed, our intelligent machines are artificial.This problem can only be addressed by the axiomatic method.

# Axiom 1.0

An atom is equivalent to an atom

Axiom 2.0

A like atom produces a like atom.

Axiom 3.0

Combining atoms produce a collection of atoms

Axiom 4.0

Combined atoms produce a specific nature dependent upon the number of atoms

Axiom 5.0

Exchanging atoms in a collection of atoms produces a different number of atoms.

Theorem 1.0

Exchanging different varieties of atoms produces different natures.

Axiom 4.1

A keyword in the primordial soup is an atom.

Theorem 1.1

A keyword is an atom. In a primordial soup there are multiple keywords. The keywords produce keywords. The keywords produce a collection of keywords. The keywords produce a variety of natures.

Axiom 6.0

An environment can be described as a space containing atom(s).

Axiom 7.0

The term atom can be similar to a “keyword”. A keyword containing all the properties of an atom.

Axiom 8.0

A behavior can be defined as the change of the environment, the environment composed of keywords.

Axiom 9.0

A nature is the representation of a keyword in the environment.

Axiom 10.0

Unique natures produce unique behaviors.

Axiom 11.0

Natures, when affected by the behaviors of other, natures will produce different behaviors.

Axiom 12.0

A nature’s behavior perturbed randomly will produce random behavior.

Axiom 13.0

An increase in perturbation will cause an increase in random behavior

Axiom 14.0

Random behavior produces a variety of behaviors

Theorem 1.2

The combination of keywords in random will produce a variety of behaviors. These behaviors will interact.

Axiom 15.0

The more natures present in the environment the more interactions present.

Theorem 1.3

The more natures present the more likely an interaction will occur.

Axiom 16.0

A successful nature is a nature whose production grows in constituent atoms.

Axiom 17.0

The more interactions occur, the more likely a successful nature will be produced.

Axiom 18.0

Emergent Behavior Axiom. The more complex interacting behaviors, the more intelligent behavior is produced.

Theorem 1.4

The more successful natures occur, the more likely an environment will not fail. The more successful natures occur, the more likely it is to parse properly,

Theorem of Primordial Soup

The greater the value of n the more likely for the machine to not fail, parse properly, and possibly show emergent behavior.

# Mathematical Approach

*Initialize*

*Genetic Interaction Functions*

. *Essentially sexual reproduction.*

*a portion of w and a portion of q combine to form a new gene.*

(link “viruses”:protein chains together)

→Փ

*Let p = the parsing function*

*p(u) = true | false*

*The gene(s) parse or don’t parse.*

*E = { 𝞼0, 𝞼1, 𝞼2,...𝞼n }*

*Where E is the set of Genes making up the “Soup” and makes up the environment and 𝞼 is a gene or chain of genes.*

*Populate Environment*

*The initial environment is populated by random genes. At a random timing , more keywords are inserted into the soup.*

*Competition: This involves viral injection, mutation, crossover, and conjunction functions. If a gene(s) survives a predators attack, mutation, or crossover, it becomes more in abundance. If the gene breaks down, then that gene is removed from E.*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform viral infection*  *gene a is injected into b*

*Perform the parsing function p(a∪b)*

*p(a∪b) = false remains as an individual gene,.*

*p(a∪b) = true.. Perform*  *(conjunction function)*

*Virus infects (takes over) part of genetic code.*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform the parsing function p(z)*

*p(z) = false remains as an individual original gene*

*p(z) = true. Mutates old gene to new gene.*

*Damage to gene from external, unknown source.*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform the parsing function p(w∪v)*

*p(w∪v) = false remains as two separate individual gene(s)*

*p(w∪v) = true Perform*  *(conjunction function)*

*Produces two individual, new, genes.(s)*

# 

# Testing the Primordial Ooze

The first step would be to create the primordial (soup) ooze., This consists of multiple virus programs that mutate other viruses. Since we aren’t at the stage where we can mutate executables, we will test it by creating a program that intentionally alters the source code of another small application (virus). We perform the mutation randomly. Replacing keywords from a bank of other keywords. The implementer (coder) will review the results by pressing any key, in which another mutation occurs, the results output, and so on. Eventually this process will be automated, the mutations occurring over and over again until a language parser for the language declares the language as successful, i.e., it isn’t going to crash. When this occurs, we have a partial success. The following is the two applications, the two virii for the first test. Eventually the virii should all attack each other, the virii that do not parse are considered evolutionary losers and discarded.

**Victim**

This is the program that is attacked by the virus.

public class Main

{

public static void main(String args[])

{

System.out.println(“Testing primordial ooze.”);

Sysemt.out.println(“I think therefore I am”);

int testint = 5 + 5;

}

}

**Attacker**

public class Main

{

public static void main(String args[])

{

//load victim into buffer

while(true)

{

//mutate keyword

//output new code

}

}

}

# Algorithms and Data Structures

There is no common genes between the two.

Set keywordsA = new Set();

Set keywordsB = new Set();

ArrayList keywordsD = new ArrayList();

choose(kwdX, keywordsA)

choose(kwdA, keywordsB)

Choose two random genes.

Replace a random keyword with a source keyword from a random language. That is, replace a random chemical with another random chemical.

Let keywordsA be a random language A and keywordsB be a random language B.

Then choose(kwdX, keywordsA) and choose(kwdA, keywordsB) are random keyword X and random keyword B. kwdA and kwdX are swapped.

Replace a random keyword with a random keyword in the same language.

choose(kwdZ, keywordG), choose(kwdU, keywordG)

kwdZ is swapped with kwdU

Swap keywords from two languages. Take two keywords, each from its own respective language, and exchange them.

exchange(choose(kwdZ, keywordH) choose(kwdC, keywordG))

(link “viruses”:protein chains together)

→Փ

The conjunction function is the union of two successful genes (keyword(s)) You can see the success function below. Where *D* is

Where D is a set of genes,

D.append(kwdR);

The success function is the most complicated portion of the algorithm. Given several genes (keyword(s)) connected together, how do we determine determine if it will succeed? The question is that of will it “survive” in its “environment.” Part of succeeding is if the genes parse correctly. Parsing correctly does not necessarily guarantee success but indicates that structurally the genes are valid. For a chain of genes to succeed it must have something to compete against. To have something to compete against you must have an environment for the competition to occur.

*Let p = the parsing function*

*p(u) = true | false*

*The gene(s) parse or don’t parse.*

boolean parsed = P(textfile);

*E = { 𝞼0, 𝞼1, 𝞼2,...𝞼n }*

*Where E is the set of Genes making up the “Soup” and makes up the environment and 𝞼 is a gene or chain of genes.*

Set A

A.append(*𝞼0, 𝞼1, 𝞼2,...𝞼n)*

*Populate Environment*

*The initial environment is populated by random genes. At a random timing , more keywords are inserted into the soup.*

ArrayList environment = ArrayList();

for (timer)

randomint = random(1, 50); *random int - choose from keywords txt*

environment.randomAppend(keywords.txt)

*Competition: This involves viral injection, mutation, crossover, and conjunction functions. If a gene(s) survives a predators attack, mutation, or crossover, it becomes more in abundance. If the gene breaks down, then that gene is removed from E.*

*Viral Injection*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform viral infection*  *gene a is injected into b*

*Perform the parsing function p(a∪b)*

*p(a∪b) = false remains as an individual gene,.*

*p(a∪b) = true.. Perform*  *(conjunction function)*

*Virus infects (takes over) part of genetic code.*

Choose kwdA from keywordsW

z = temporarily inject(kwdA, keywordsP)

if (p(z)) true keep z

if (p(z)) false keep original

*Mutation Function*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform the parsing function p(z)*

*p(z) = false remains as an individual original gene*

*p(z) = true. Mutates old gene to new gene.*

*Damage to gene from external, unknown source.*

Choose kwdA from keywordsW

Choose kwdB from keywords.txt

m = temporarilyswap(kwdA, kwdB)

if (p(m)) true keep m

if (p(m)) false reject keep original

*Crossover Function*

*Select at random a=𝞼0 and* b=*𝞼1.*

*Perform the parsing function p(w∪v)*

*p(w∪v) = false remains as two separate individual gene(s)*

*p(w∪v) = true Perform*  *(conjunction function)*

*Produces two individual, new, genes.(s)*

Choose kwdA from keywordsW

Choose kwdB from keywordsZ

e = Swap(kwdA, kwdB)

if (p(e)) true keep e

if (p(e)) false reject e keep original

# 

# Approach

In this paper we take the inductive approach to analysis to the theorem at hand. Given the proof and mathematics supporting it, we hope to prove that the theorem is valid and functional. We will show that the Self Producing Genetic Pool Theorem holds true.

# Parser

An important aspect of proving the Self Producing Genetic Pool Theorem is the Parser. The parser decides if or not if the language produced is valid and functional. Because of the uniqueness of the language being generated, it is necessary to code a custom parser.

First Step: Tokenize the input string

Second Step: For each rule in BNF code a function

Third Step: Extract a token from the tokenized input string apply to function

Fourth Step: Continue extracting tokens from the input string and applying to function

Fourth Step: If the input string has been consumed then it has been parsed

Fifth Step: If there is no path to take function-wise then the parsing has failed

# Data Set

The Data Set is what is composed of the Environment. Thus, the Data Set contains all of the possible atoms (keywords). The keywords are not your typical programming language keywords, but carry more data and weight. Semicolons and braces are left out.

The first testing set is essentially your “hello world” and consists of the following:  
*main*

*class*

*Main*

*void*

*println*

*“hello world”*

# Testing Results

# Conclusion