Abstract

This report is concerned with the creation of virtual evolutionary worlds. It discusses world design with particular attention to the problem of achieving open-endedness. This is followed by a description of a virtual machine aimed at achieving this goal and a summary of several experiments performed using this system.

Contents

Preamble....2
Introduction....2
Layout....2
Evolution....2
Evolutionary theory....2
Evolution and computers....3
Considerations and criticisms of evolutionary systems....4
Building worlds....4
The first objective....4
The second objective....4
Constricted worlds....5
Open-ended worlds....5
The spectrum....6
The virtual machine....6
A few thoughts on interpretation and computers....6
Data as programs....6
Interpreting data as instructions....6
Operands....7
Self modifying programs....8
Instruction set....9
Limitations and additional notes....9
Experiments....10
Experiment 1....11
Experiment 2....13
Experiment 3....16
Conclusion....19
The last experiment....19
Closing remarks and acknowledgements....20
References....20

Preamble

The project described in this report did not arise out of any desire to solve a particular problem, nor the desire to create a product. In a first instance came the acceptance of the computer as a medium. Investigation of this concept led to many avenues, one of which – the acknowledgement of programs as data – inspired the succession of ideas that form the basis for the project. This report attempts to amalgamate these ideas into a single logical whole. It is worth noting that it is merely one interpretation of what was undertaken.

Introduction

The concept of a virtual evolutionary world touches on many disciplines, as does its implementation. Formulating a report that encompasses all the relevant subject areas is extremely difficult. As such this report focuses on the topics most relevant to the project undertaken and for this reason the treatment of certain topics will no doubt seem cursory to anyone familiar with the subject matter.

Layout

The layout of this report bares no resemblance to the chronology of the project. During the project the experiments and research happened concurrently, the one informing the other. As such there was no obvious order in which to present the two. It suffices to say that, though the experiments are presented last, they greatly informed many of the topics discussed in previous sections.
The topics dealt with can, broadly speaking, be divided into those that relate to evolutionary systems and those that relate to their implementation. This report begins with a discussion of evolution and world design followed by a description of the virtual machine implemented and a summary of the experiments carried out.

Evolution

> **evolution** /ˌi:vəˈlu:ʃ(ə)n/ *n*. **1** gradual development. **2** development of species from earlier forms, as an explanation of their origins. **3** unfolding of events etc. (*evolution of a plot*). **4** change in the disposition of troops or ships. □ **evolutionary** *adj.* [Latin: related to EVOLVE]
>
> **evolutionist** *n*. person who regards evolution as explaining the origin of species.
>
> **evolve** /Iˈvþlv/ *v*. (**-ving**) **1** develop gradually and naturally. **2** devise (a theory, plan, etc.). **3** unfold. **4** give off (gas, heat, etc.). [Latin *Volvo volut-* roll]
>
> The Pocket Oxford Dictionary (1996)

Evolutionary theory

Evolution is often, mistakenly, equated to Darwin's theory of natural selection. Darwin's theory is indeed widely accepted as a useful description of the process; however there is an obvious distinction between evolution – a process – and theories of evolution – imperfect but useful descriptions of how this process occurs. Natural selection is one of the forces that is useful in describing how evolution occurs, however, as this extract from Wikipedia[1] illustrates, it is one among several:

> The prevailing formulation of the theory of evolution is the modern synthesis, which brings together Darwin's theory of evolution by natural selection and Gregor Mendel's theory of inherited characteristics, now called genes. In the modern synthesis, 'evolution' means a change in the frequency of an allele within a gene pool. This change may be caused by a number of different mechanisms: natural selection, genetic drift or changes in population structure (gene flow).

Evolution and computers

There is a well established precedent for the application of evolutionary concepts to computers. Existing examples form a useful reference point when trying to understand how it is possible to abstract evolution in terms of a computer system.

Genetic programming (a summary)

Genetic programming, as the name suggests, is a programming technique. As is true of all programming techniques the objective is to produce an algorithm that solves a specific problem. Genetic programming starts with a high level description of the problem and uses a genetic system to search the vast array of possibilities for good solutions. This search is performed as a series of 'experiments' (runs) and refined through the application of genetic operators.

Performing a run

Preparatory steps

Before performing a run the problem to be solved must be described to the genetic system. This requires the following to be defined:

Terminal Set	The set of (initial) values that can be input into functions.
Function Set	The set of operations that can be used in the algorithm being developed.
Fitness measure	A means of measuring a program's success at solving the problem.
End condition	A condition used to determine when the end of a run has been reached.
Parameters	Other parameters necessary for configuring runs.

The process

A run is performed as follows:

1. A set (population) of random programs (individuals) is generated.
2. The success of each individual at solving the specified problem is evaluated using the fitness measure.
3. The end condition is evaluated to see if the run has finished. If so the most successful program is designated as the result and the run terminates.
4. Otherwise a new population is generated by selecting individuals from the last generation - with a bias for those that were most successful - and performing mutation operations to create new individuals.
5. The process is repeated from step 2 with the new population.

Artificial life (a summary)

Artificial life, unlike genetic programming, is not concerned with the application of evolutionary theory to computers, but the evolutionary process itself. In its simplest form, artificial life involves the definition of a virtual world within which there is the potential for complex systems to develop. This world can then be flooded with random data to form a virtual primordial soup from which systems can evolve, or loaded with existing systems to observe if and how they evolve. As Bedau [2002, p.3] summarises in his article The scientific and philosophical scope of artificial life, such experiments are interesting for many reasons:

> The challenges fall into three broad categories: the origin of life, life's evolutionary potential, and life's connection to mind and culture.

The essay goes on to elucidate ways in which artificial life can lead to a better understanding of these topics. It also presents a cursory discussion of the implications of artificial life for the arts. However, unsupported claims such as "artificial life is radically changing human culture and technology" suggests that this particular discussion may have more to do with science fiction than fact.
Perhaps the most reputed examples of artificial life systems are Tom Ray's Tierra[2a] and a derivative system, Avida[3], developed at the MSU Digital Evolution Laboratory.

Considerations and criticisms of evolutionary systems

The fact that an evolutionary system demonstrably works lends credence to the evolutionary theory from which it was abstracted. However, any such system will be subject to all the limitations and imperfections of that theory.
Genetic programming is a useful tool, however, its validity in terms of investigating the evolutionary process is limited in that it can only serve to test evolutionary theory (and even in this respect it is limited).
Artificial life is involved with the process of evolution and as such is a potentially viable means of investigating it. However, critics of evolutionary systems in general often argue that in a first instance these systems assume too much[1]. This is a valid point especially in the context of artificial life.

Investigating the evolutionary process

For a system to be useful in investigating the evolutionary process, and not simply testing evolutionary theories, it must make no assumptions about how the process occurs. It cannot be designed to allow a specific evolutionary theory to occur since this would bias the experiment. If the system is designed to allow all existing evolutionary theories to occur it still biases the experiment against any process that falls outside the scope of these theories. Thus in designing a system to investigate evolution, evolutionary theory must be disregarded.

[1] Several examples can be found at newCreationism.org[5a] for example Miller's article The Bugs with The Bugs[5b].

Building worlds

In order to investigate evolution it is first necessary to define a virtual world in which this investigation can take place.
Within the context of this project world design involves two objectives that are ultimately contradictory. As will be seen, these objectives form the poles of a spectrum which is useful as both a tool and a benchmark.

The first objective

The first objective is the creation of a world in which evolution can occur. The creation of such a world should not focus on evolution by any particular means. This may sound implausible, but it is not. If evolution is the process by which systems develop, this makes only two requirements on the world:

- It must be possible for complex systems to exist.
- It must be possible for systems to change (i.e. it must be possible for systems that rearrange systems to exist).

The second objective

The second objective is to create a world that is visually tangible. This stems from the fact that visualisation is an extremely useful (if not necessary) tool in comprehending and communicating ideas about a world. In this respect this objective is equally as important as the first since a world that cannot be comprehended or easily discussed is unlikely to offer any insights.

Constricted worlds

It is obvious that the more constricting the laws imposed on a world the more the outcome is limited, to the extreme where the experiment looses all interest because the outcome is obvious.
For example in a world where entities can only move and eat, the only possible outcomes are that an entity always eats, always moves, or moves and eats. Always eating and always moving will ultimately result in starvation so the only possible outcome is entities that move and eat. The way in which the entities move may vary but this presents a very limited set of possibilities.

Open-ended worlds

It is important to understand that open-endedness is not a measure of the number but the range of possible outcomes. The latter is necessarily linked to the first but is not the same thing. To take the previous example, the entities may be able to move in a huge number of ways resulting in numerous possible outcomes, however, the scope of the possible outcomes is extremely limited: entities can only move, eat or move and eat.
This example also illustrates that the range of possible outcomes is dependant on the set of possible interactions (the range of outcomes is limited because the only possible interactions are moving and eating). In other words: the larger the set of possible interactions the more open-ended the world.
Now consider two worlds: one in which interaction occurs at the object level and the other at the sub-object level. It is not possible for interaction to occur at a sub-object level if interaction is defined at the object level, however the opposite is true: it is possible for interaction to occur at an object level if the interaction is defined at a sub-object level. Thus the set of interactions possible at an object level are a sub-set of the interactions possible at a sub-object level. To take a more concrete example: consider an entity kicking a ball. This interaction could be dealt with at the level of the entity and the ball; however a much more generalised solution would be to deal with it at the level of the colliding surfaces. In the first case only one interaction is possible (entity+ball), whereas in the latter case many different interactions are possible (a surface of the entity+a surface of the ball).
This suggests that a world in which interaction occurs at a sub-object level will be more open-ended than a world in which interaction only occurs at an object level. Thus the more fundamental the level of interaction the more open-ended the world.

Achieving open-ended worlds

In the physical world life forms exist within the same physical space and as such their interaction is only defined at the fundamental level of physical interaction. In the virtual domain the most fundamental property is not energy or matter, but memory. It follows that in a similarly open-ended virtual world interaction must occur at the fundamental level of memory manipulation.
Such a world is possible, it equates to a single memory space containing several programs. The entire memory space is run as a single process with multiple threads of execution. A good example of this approach is Corewar[4].

The problem with open-ended worlds

The problem with constructing a world as a single memory space is that it is extremely difficult to distinguish individual entities (or more specifically to define what constitutes an individual). It is possible to visualise the memory space, to observe, in detail, the memory manipulations that occur, but it is almost impossible to interpret them as anything more than a sequence of memory operations. Essentially what we are observing is a single dimensional world in which time and space are replaced by processor cycles and memory. Within this world the possibilities are almost limitless, but it is so radically different from anything we know that we are incapable of comprehending it. To render

such a world more comprehensible necessarily requires the introduction of additional restrictions.

The spectrum

Essentially what this demonstrates is that the two objectives behind world design are ultimately contradictory. At one end of the spectrum is the 'tangible': worlds that are immediately recognisable and comprehensible but with such limited potential that they are devoid of interest. At the other end is the 'open-ended': worlds with only the most fundamental limitations but so abstracted that they are completely incomprehensible.

The virtual machine

The following is not a complete description of the virtual machine implemented, but attempts to describe its distinguishing features.
The fundamental principal underlying the design described is the stripping away of all unnecessary abstraction in order to achieve a system that is both extremely flexible and extremely simple.

A few thoughts on interpretation and computers

Information without interpretation is meaningless.
The shapes on this page mean nothing unless they are interpreted as words. What if they were interpreted as a picture?
Computers make reinterpreting information easy since all information is stored in the same format (binary). Typically this data is sent to a device which translates it into a format that is more readily understood by the user. In theory this allows perception of a piece of information to be radically altered simply by sending it to a different device.
The most important 'device' in a computer is the CPU which interprets data as programs: sequences of instructions. These instructions manipulate the internal state of the computer. Sending the wrong sequence of instructions to the CPU can destabilise the computer resulting in unpredictable behaviour, system failure and even permanent damage.

Data as programs

A basic concept underlying the virtual machine is that it should be capable of interpreting any data as a valid program. For this to be possible it must be able to interpret any chunk[2] of data as a valid instruction. In addition all programs should be completely safe. No program should be capable of compromising the stability of the virtual machine.

Interpreting data as instructions

Any chunk of data can be interpreted as an integer value and
In most computers the CPU associates numbers with basic operations e.g.

1:	0x0001	MOV	move data between memory addresses
2:	0x0002	ADD	add two pieces of data
03:00:00	0x0003	SUB	subtract two pieces of data
...	...	...	

Typically there is a (relatively) small number of instructions so only a small set of numbers represent valid operations. This means that most chunks of data would not equate to valid instructions. Potentially this could be resolved by fixing the size of each chunk thus limiting the range of values, however, this is problematic since it requires that for every chunk of data (**d**) to be meaningful the number of instructions ($\mathbf{I_n}$) must be $2^{(\mathbf{d_s})}$ where $\mathbf{d_s}$ is the number of bits per chunk. Ultimately we do not want to be confined to a specific number of instructions or even a specific chunk size.

[2]For the purposes of this essay a chunk constitues a subdivision of a block of data.

This problem of mapping the set of possible numbers (**D**) into the set of possible instructions (**I**) can be solved using modulo arithmetic. This allows any number to be mapped into a given range[3]:

$$\mathbf{d}\%\mathbf{I_n} = \mathbf{x} \text{ where } \mathbf{x} \in [0, \mathbf{I_n}[\text{ and } \mathbf{d} \in \mathbf{D}$$

E.g. to map all numbers into the range [0, 20] we take modulus 21 of each number:

d	**Calculation**	**x**
0	0%21	0
10	10%21	10
20	20%21	20
21	21%21	0
250	250%21	19
1100	1100%21	8
...	...	...

This method allows any chunk of data to be mapped into the range of possible instructions. In other words it is possible to interpret any chunk of data as a valid instruction given a predefined instruction set.

Operands

Most instructions operate on data e.g.

COPY [*source*] [*destination*] (copy source to destination)

The data being manipulated is typically stored in memory. The way in which this manipulation occurs varies depending on the instruction set and the architecture of the CPU. Typically most CPUs have specific instructions for accessing data in memory rather than letting all instructions manipulate memory directly. Thus in the case of the above example *source* and *destination* would almost certainly be registers within the CPU, not memory addresses. However, this multi-tiered approach is primarily the result of hardware limitations and speed requirements, and in a virtual machine such constraints do not exist. Indeed, the use of registers can be regarded as a special case in which the instruction set has been implemented to manipulate memory in a specific way (and as will become apparent, it is entirely possible to implement such an instruction set if required). Therefore it is safe to assume that in the general case all instructions operate on a single memory space **M** which contains all data. If we define **M** as:

$$\mathbf{M} = [0, \mathbf{M_n}[\text{ where } \mathbf{M_n} \text{ is the number of available memory slots}$$

it is obvious that the method used to interpret any information as an instruction can also be used to interpret any information as a valid memory address.
This provides a very simple method for instructions to manipulate data: the chunks following an instruction can be interpreted as the memory address(es) of the operand(s) e.g.
Given an instruction set:

1: COPY [*source*] [*destination*]

and a program:

1 250 32

This is interpreted as:

COPY 250%$\mathbf{M_n}$ 32%$\mathbf{M_n}$

which copies the data at address 250%$\mathbf{M_n}$ to address 32%$\mathbf{M_n}$ of the memory space **M**.
After an instruction is executed the program counter would typically be incremented by the sum size of the instruction[4] i.e. to point to the memory

[3]For the purposes of this essay the *a*%*b* is equivalent to *a* expressed in a modulo *b*.

[4]The sum size of an instruction is the amount of memory occupied by an instruction and its operands.

address after the last parameter of the previously executed instruction. So in the case of:

1 250 32 25

Instruction 1 (COPY) uses 250 and 32 as parameters. The next instruction to be executed would be 25.
This method means that given an instruction set:

$\mathbf{I} = [0, \mathbf{I}_n[$

and memory space

$\mathbf{M} = [0, \mathbf{M}_n[$

any data can be executed as a valid program.

Self modifying programs

As was stated previously: the distinction between types of information is not implicit in the information, but its interpretation.
In the case of this virtual machine, the distinction between the program and the data it operates on can be seen clearly:

- There is the data that is interpreted as a sequence of instructions and is therefore, by definition, a program (**P**).
- There is a memory space (**M**) which contains the data on which the instructions operate.

This suggests an obvious line of questioning:
Can **P** = **M**, and if so what are the implications?
In other words, can the program and the data on which it operates be one and the same thing, and if so what does this imply?

Can P = M

The answer to the first question is simply yes - in theory there is no reason why a program cannot operate on the memory space it occupies (it is, after all, just data). In a practical sense, this would not be possible on a typical CPU, but in the case of the virtual machine described it can be done easily and without risk to the system's integrity.

Implications

Understanding what this means is less obvious since there is more than one possible interpretation. In a first instance it is worth noting that, combined with the program counter, the data in the memory space entirely defines the state of the virtual machine at all times.

A simple description

A simple description of the data is as a program that has the capacity to modify itself during execution. The degree to which this occurs is dependant on the program. It is possible that a program does not modify itself at all, in which case the program runs as if in a separate memory space to the data (a situation comparable to **P** != **M**).

A more complete description

However, the previous description fails to communicate the fact that the data not only defines a program, but to a certain degree, defines the architecture of the virtual machine (bearing in mind the instruction set is external to this data). For example, consider a virtual machine that uses a standard instruction set: it is entirely possible for the data to define a program that executes as though on a traditional CPU, since it can define specific memory addresses as registers and maintain a strict separation between the data that constitutes the program and the data on which it operates.
It is tempting to state that, as long as the instruction set is complete[5] then it is possible to define any architecture within the data. However, such a statement requires substantial verification, and as such simply remains an interesting proposition.

[5]Complete in the sense of Turing Complete.

Whether or not the above statement is true, it is certainly possible to achieve a diverse set of architectures, and perhaps most importantly, it is entirely possible for the architecture to change during execution.

Instruction set

Although typically the instruction set is included within the description of a virtual machine it is not necessary in this instance. The virtual machine is constructed such that only the instruction set limits the tasks programs are capable of performing. Therefore it makes sense to differ description of instruction sets to within the context of specific tasks.

Limitations and additional notes

Data and programs

As has been explained, in the case of $\mathbf{P} = \mathbf{M}$ the data that is executed can only loosely be considered a program, however, for the sake of simplicity this document will continue to use the term 'program' to refer to this data.

Chunk size

The virtual machine makes no requirements on the size of chunks, however, certain considerations are worth mentioning.

Maximum instruction set size

The virtual machine parses a program by reading a chunk, interpreting it as an integer, and then mapping it into the range of valid instructions. It is obvious that if the size of the chunk ($\mathbf{d_s}$) is such that:

> $2^{(\mathbf{ds})} < \mathbf{I_n}$ where $\mathbf{I_n}$ is the number of instructions in the instruction set.

Then any instructions in the range $]2^{(\mathbf{ds})}, \mathbf{I_n}]$ will not be used. Thus the maximum size of the instruction set is effectively $2^{(\mathbf{ds})}$.

Probability distribution of instructions

In addition to the above, it is worth considering the implications of:

$$\mathbf{I_n} > 2^{(\mathbf{ds})} \text{ and } \mathbf{I_n} \mathrel{!}= \mathbf{n}(2^{(\mathbf{ds})}) \text{ where } \mathbf{n} \in \mathbb{I} \text{ and } \mathbf{n} > 0$$

Obviously, this poses no problem in that it works, however, let us consider a specific example:
Given an instruction set $\mathbf{I} = \{\mathbf{I_0}, \mathbf{I_1}, \mathbf{I_2}\}$ and assuming $\mathbf{d_s} = 3$ the different chunks get mapped to instructions as follows:

d	I
000	$\mathbf{I_0}$
001	$\mathbf{I_1}$
010	$\mathbf{I_2}$
011	$\mathbf{I_0}$
100	$\mathbf{I_1}$
101	$\mathbf{I_2}$
110	$\mathbf{I_0}$
111	$\mathbf{I_1}$

$\mathbf{I_0}$ and $\mathbf{I_1}$ are effectively duplicated three times, whereas $\mathbf{I_2}$ is only duplicated twice. Given a random program this means that it is more likely that instructions $\mathbf{I_0}$ and $\mathbf{I_1}$ are executed than $\mathbf{I_2}$.
This imbalance becomes less significant as $2^{(\mathbf{ds})}$ increases relative to $\mathbf{I_n}$, so with a significantly larger value of $2^{(\mathbf{ds})}$ the imbalance is negligible.

Program counter

Sequential programs are not imposed

When a program is started the program counter is initialised to memory address 0. Previously it was stated that when an instruction is executed the program counter would typically be incremented by the sum size of the

instruction. This is not imposed nor required. The behaviour of the program counter is entirely determined by the instruction executed. This is important as it allows an instruction to jump execution to any other point in the program (and is the basis for control structures).

Circular memory spaces and offset programs

Using modulo arithmetic to ensure that memory addresses are always within range is equivalent to having a circular memory space. A subtle implication of this is that if the program counter reaches the end of the program and wraps around to the beginning it may be offset relative to memory address 0. This means that it is entirely possible for a different sequence of instructions to be executed on subsequent passes through the memory space.

Experiments

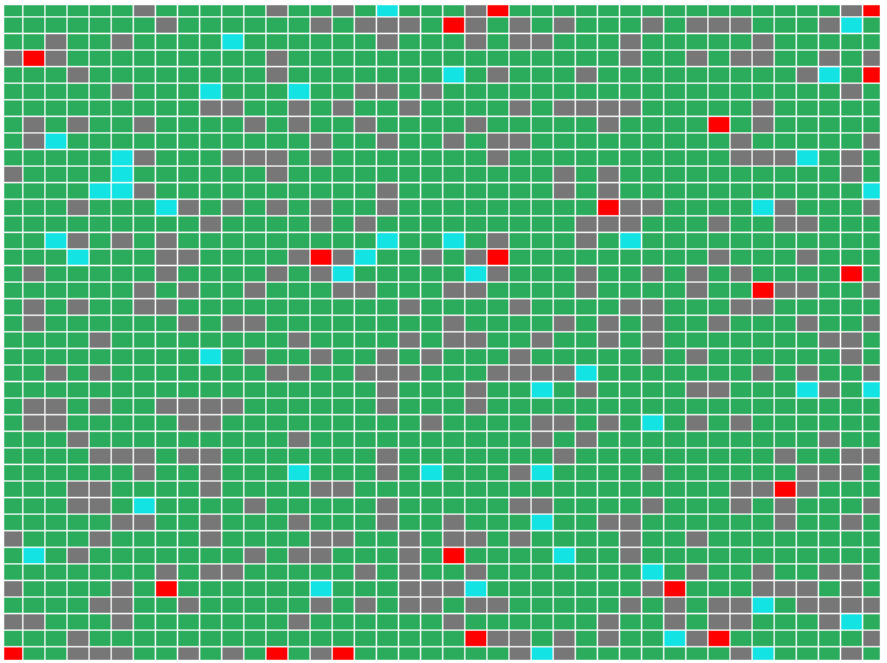
The following section describes several experiments undertaken during the course of the project. These experiments were based on and informed the design of the previously described virtual machine.

Experiment 1

Description

World

The world is defined as a 2D grid. Each square of the grid can be occupied by an entity (red), an obstacle (grey), or food (cyan).
Entities are controlled by separate programs and have an energy pool. Each instruction executed costs the entity one unit of energy. If an entity's energy pool is empty it is considered dead and is removed from the world.
Within the world entities can move horizontally or vertically between adjacent squares. Movement into a square that is currently occupied by another entity or an obstacle is not permitted. If an entity moves into a square containing food, the food is added to the entities energy pool and removed from the world.



screenshot of experiment 1 during a 'run'

Instruction set

All operands are read as 4 byte integers and all instructions increment the program counter by their sum size.

Instruction	Description
BNE [&*test*] [&*addr*]	If *test* is not even execution jumps to *addr*
MOV [&*from*] [&*to*]	*to = from*
ADD [&*a*] [&*b*] [&*res*]	*res = a + b*
SUB [&*a*] [&*b*] [&*res*]	*res = a - b*
DAT [&*type*] [&*dir*] [&*res*]	If *type* is even tests to see if the tile next to the entity in direction *dir* is empty and sets *res* to the result (0/1). If *type* is odd tests to see if the tile next to the entity in direction *dir* contains food and sets *res* to the result (0/1). The direction is determined by *dir*%4 where 0 = north, 1 = east, 2 = south, 3 = west.
EXE [&*dir*]	Attempts to moves the entity in the direction determined by *dir*. The direction is determined by *dir*%4 where 0 = north, 1 = east, 2 = south, 3 = west.

Summary of operation

1. A world is created containing obstacles and a randomly generated population of entities.
2. The entity's programs are executed in parallel (allocating an equal amount or CPU time to each program). During this process food appears in random empty squares within the world. The amount of food appearing is proportional to the number of empty squares. Each piece of food exists for a fixed duration of time before being removed. This process continues until all entities have died.
3. A record is kept of the ten entities that survived for longest.
4. A new population of entities is created by selecting random entities from the top ten and performing a varying amount of random mutation on their programs.
5. The world is repopulated with the new entities and the process is repeated from step 2.

During operation the record that is kept of the top ten programs is maintained between populations. Thus it is not a record of the programs that lasted longest during the last run, but the programs that have lasted longest in all previous runs.

Results

The longest lifetime of any given entity improves periodically (the length of time between successive improvements appears to increase).
When left for a substantial amount of time the entities behaviour alters subtly and they begin to move more rapidly between a larger number of squares.

Observations and criticism

This experiment is flawed in a number of ways:

- The instruction set described is too limited and it is doubtful that it is in any way complete.
- The process is extremely close to that of genetic programming, and this comparison highlights several issues.
    - A first example is that the only genetic operator used to produce new entities is a 'varying amount' of random mutation. If this amount is 0 then an entity is effectively

copied into the new population and if this amount is large it is equivalent to generating a new random program. Thus the operations in use are effectively: direct copy, random mutation and the generation of new programs. It is hard to determine any form of weighting for these operators, however, what is most noticeable is the complete omission of the cross-over operator which is by far the most important operator in genetic programming.

  - Another example is that although maintaining a record of the top ten programs is interesting, generating new populations from this list is questionable. It is almost certain that this method will have a strong tendency towards local maxima.

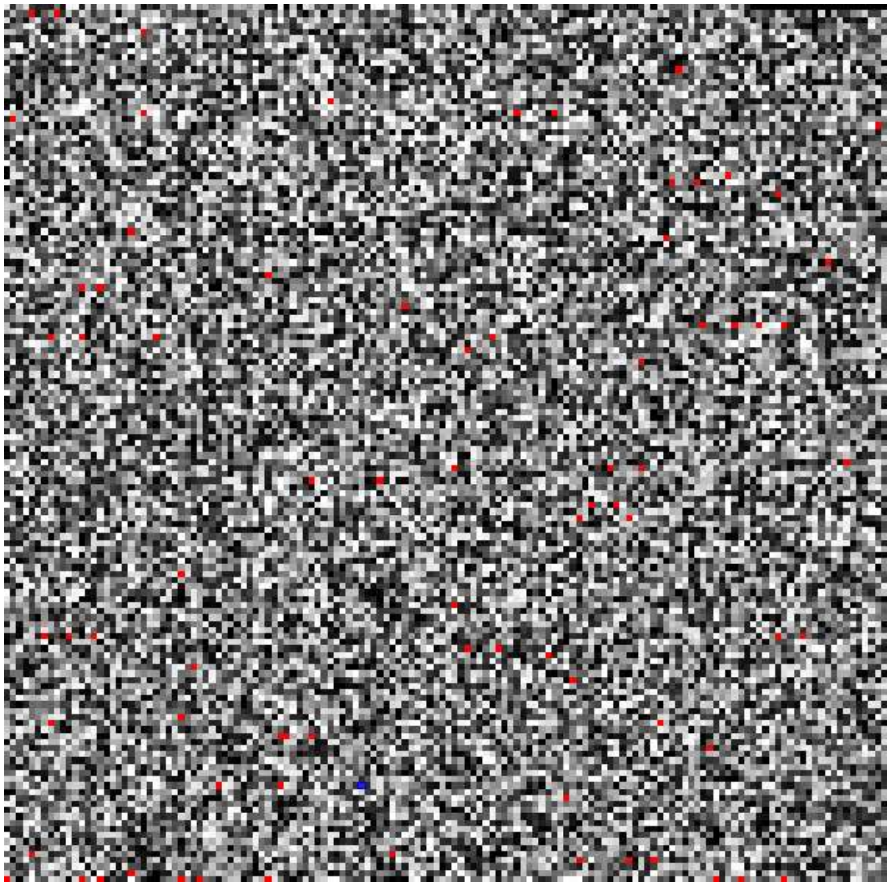
- The fact that the food is generated at random locations severely limits the experiment since the only possible strategy in such a situation is for an entity to cover as much area as possible.

The results of this experiment are predictable and coincide with a fairly obvious best strategy.

Ultimately the system used to find better performing programs is flawed and is little better than a random search algorithm.

One respect in which the experiment does succeed is that the visual description of the world, though simple, is reasonably effective in communicating the results.

Experiment 2



screenshot of experiment 2: a program with multiple threads of execution

In actual fact, this was not a single experiment, but a set of experiments primarily aimed at better understanding the virtual machine. Typically one experiment inspired several other related experiments.
In a first instance these experiments comprised a simple visualisation of a program being executed[6]. A later addition was to provide a primitive means of interacting with the memory space of the program. The incentive was to provide a primitive means of 'debugging' programs; however the end result was intriguing in its own right.

Description

The program being executed is displayed as a grid. Each square of this grid corresponds to a memory location: the bottom left square is the first memory location and successive slots continue to the right of this. At the edge of the screen the subsequent address is mapped to the left-most square of the row above.
Threads are indicated by colouring the square that corresponds to their current point of execution a different colour.
Specific experiments allow the user to set the data in a given memory slot by specifying a value and selecting the corresponding square with the mouse. This copies the value into the memory location.

Instruction set

One of the aims of these experiments was to observe the effects of using different instruction sets, therefore no single instruction set was used. What follows is a list of many of the instructions that were used during the various experiments.

[6] Not to be confused with experiments 1 and 3 which are visualisations of a world containing several entities as opposed to a memory space.

Instruction	Description
null	Does nothing
copy [&*from*] [&*to*]	*to* = *from*
inc [&*val*]	*val* = *val* + 1
dec [&*val*]	*val* = *val* - 1
and [&*a*] [&*b*] [&*res*]	*res* = *a* & *b*
or [&*a*] [&*b*] [&*res*]	*res* = *a* | *b*
add [&*a*] [&*b*] [&*res*]	*res* = *a* + *b*
sub [&*a*] [&*b*] [&*res*]	*res* = *a* - *b*
multiply [&*a*] [&*b*] [&*res*]	*res* = *a* * *b*
divide [&*a*] [&*b*] [&*res*]	*res* = *a* / *b*
jump [&*dest*]	Jumps execution to *dest*
jumpIfEqual [&*a*] [&*b*] [&*dest*]	Jumps execution to *dest* if *a* = *b*
jumpIfNotEqual [&*a*] [&*b*] [&*dest*]	Jumps execution to *dest* if *a* != *b*
jumpIfGreaterThan [&*a*] [&*b*] [&*dest*]	Jumps execution to *dest* if *a* > *b*
jumpIfLessThan [&*a*] [&*b*] [&*dest*]	Jumps execution to *dest* if *a* < *b*
conditionalJump [&*type*] [&*dest*] (…)	Combines all the above jumps into a single instruction where *type* determines the condition and *dest* the where to jump execution to.
spawnThread [&*addr*]	Spawns a new thread at *addr* if the current number of threads is below a predetermined limit
killThread [&*tid*]	Kills the thread *tid%*maxThreads if there is more than one thread executing

Results

The specific results varied from experiment to experiment. The cumulative result was to highlight many issues not previously considered. These included both implementation problems and questions about the larger objective.

Observations and criticism

Cyclic and steady states

A fact rendered extremely obvious by these experiments is that programs have a tendency to collapse into steady or cyclic states. No matter what the initial state of the memory space, programs eventually end up in a continuous loop.

This is not that surprising, since it can be proved that if steady/cyclic states exist within any deterministic system that evolves relative to a parameter t, as $t \rightarrow \infty$ the probability of the system being in a steady/cyclic state $\rightarrow 1$.

One possible conclusion to draw from this is that if such systems are going to learn/adapt, there must be external stimulus. Without such stimulus the programs will stagnate.

The collapse into a steady state is inevitable, however, the speed at which it occurs is not constant, and experimentation suggests that it is dependant on a number of factors. The most important factor is the number of instructions that perform jump operations in the instruction set – if the number is high the program becomes cyclic more quickly. Multi-threading reduces the speed at which cyclic states occur; however the effect is not very noticeable unless the maximum number of threads is reasonably large.

Probability distribution of instructions

The virtual machine ensures that given a random program each instruction in the instruction set is equally likely to be executed. Despite this the instruction set can still be biased towards particular operations they can be performed by more than one instruction. This first became apparent when using an instruction set that included several conditional jump instructions (all of which can emulate a standard jump operation). The result was that the number of jumps was extremely high relative to any other operation.

This can be regarded as an argument for using a reduced instruction set in which the operation performed by one instruction cannot be emulated by any other instruction in the set. However, in some respects the ability to bias the instruction set is not without interest: it is possible that certain weightings may be more conducive to producing complex systems than others.

Such bias can easily be achieved within the existing virtual machine (with very little penalty to performance) by duplicating instructions within the instruction set.

Offsetting

Offsetting, as discussed in the description of the virtual machine, occurs when a thread becomes offset relative to a previously executed area of memory such that when the same area of memory is executed subsequently it is interpreted as an entirely different set of instructions.

Offsetting is not in itself a problem but it is worth considering the ways in which it can occur. One possibility, as previously discussed, is when a thread reaches the end of the memory space and jumps back to the beginning. A similar situation can easily arise with any jump instruction. Offsetting can also arise if an instruction within a previously executed area of memory is replaced. If the instruction is replaced by an instruction of a different sum size the result will be to offset the thread relative to all subsequent instructions.

As previously stated, offsetting is not necessarily a problem. Certainly offsetting as a result of a jump operation does not seem objectionable; however, in the latter case, it is more questionable since an obvious implication is that it is impossible for a single instruction within a sequence to be replaced by another instruction of a different sum size without modifying all subsequent instructions. To a human a dependency such as this would render programming near impossible which suggests that it may be undesirable.

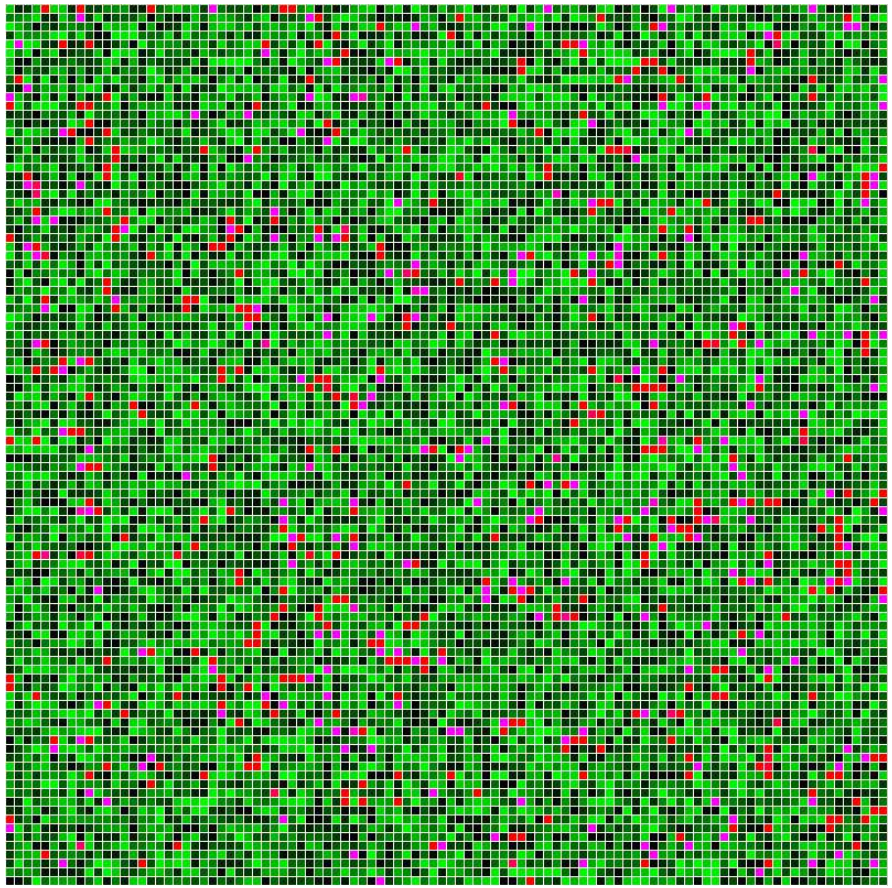
This 'problem' with offsetting occurs for one of two reasons:

- Not all instructions have the same sum size.
- Sequential execution is assumed.

Addressing the latter of these conditions is by far the better solution. Addressing the first means imposing a new requirement (on the size of instructions), whereas addressing the second means breaking free of an existing requirement (that program execution should be sequential).

Breaking the second requirement effectively means making every instruction a jump operation. Perhaps the most obvious way to achieve this is to interpret the first operand of every instruction as the memory address of the instruction that follows. This effectively results in a linked list of instructions.

Experiment 3



screenshot of experiment 3: a world inhabited by several entites

The aim of this experiment was to try and develop an evolutionary world based on the first experiment but taking into account everything that was learned from the second. This experiment was based on the idea that in a virtual world matter equates to memory and energy to processor cycles[7].

Description

The world is represented by a 2D grid of square tiles. The colour of each tile is determined by its height (dark green is low, light green is high). Each tile of the grid also possesses an energy pool.

Any tile can be inhabited by an entity. An entity necessarily has memory and optionally energy. Whilst an entity has energy it will execute its memory space as a program. Each instruction executed incurs a minimum energy cost which is added to the energy pool of the tile the entity inhabits. In addition certain operations may incur an additional energy cost. Each entity may have multiple threads of execution. All threads within the world are executed in parallel (and given an equal share of the virtual machine's processor time).

Entities with energy are displayed as purple squares, whilst entities without energy are red.

Entities can perform a number of memory and energy related operations that range from simple memory manipulation to converting memory to energy. All operations are based on the following fundamental principals:

- Energy is conserved.
- Memory is energy
- Moving memory up requires energy, moving memory down releases energy.

Since "memory is energy" conversion between the two is permitted. Typically one unit of memory is equivalent to several units of energy.

An entity moving between two tiles of the same height incurs no energy cost. Moving an entity to a tile that is higher requires an amount of energy proportional to the amount of memory and the height. This energy is added to the memory pool of the destination tile. Moving an entity to a tile that is lower generates the same amount of energy as is required to lift an equivalent piece of memory the same height. The energy generated is added to the entities energy pool.

[7] This is not a new idea. Notably Tom Ray's Tierra[2b] is also based on the same equivalence.

Instruction set

The following is a summary of the instruction set used.

	Memory manipulation
null, increment, and, or, copy, decrement, add, sub	standard memory manipulation instructions (see previously for description)
	Control
switch	if the value at the specified memory address is true continues to the next instruction, otherwise jumps to a different memory address
	Threading
spawnThread, killThread	(see previously for description)
	Movement
moveFoward	moves the entity forward one tile
rotate	rotates the entity 90° left/right
	Energy transfer
transferEnergy	transfers the specified amount of energy between the entity and the entity in front of it – the direction of transfer is determined by whether the amount is greater or less than 0 (if there is no entity in front no energy is transferred)
absorbEnergy	transfers energy between the entity and the tile it inhabits – the direction of transfer and amount transferred depends on the amount of energy in the tiles energy pool. If the tiles energy pool is full energy is transferred from the tile to the entity and vice versa.
	Memory transfer
giveMemory	transfers memory from the entity to the tile or entity in front depending on whether the tile in front is inhabited – if the tile in front is higher the entity supplies the energy required to raise the memory – if the tile in front is lower the memory being transferred gains the energy that is generated

takeMemory	transfers memory to the entity from the entity in front – if the tile in front is higher the entity gains the energy generated – if the tile is lower the entity supplies the energy required to raise the memory
	Energy ↔ memory conversion
convertEnergyToMemory	converts a specific amount of energy into memory
convertMemoryToEnergy	converts a specific amount of memory into energy

Results

To date: lots of dead entities, lots of plants and a few live births from memory being dropped off cliffs.

Conclusion

The last experiment

As yet the last experiment has not produced anything spectacular but it has revealed several new problems that need to be addressed. As such it seems fitting to conclude with a brief discussion of some of these issues and ways in which they might potentially be resolved.

Stagnation

In its current state the last experiment clearly suffers from the problem of stagnation (as described in experiment 2). In a first instance the world is extremely active however it quickly settles into a much more static state. There is no obvious solution to this occurrence (and, at this stage, it is not clear if it constitutes a problem or not). One possible cause is the lack of instructions that allow entities to investigate their environment. Without such instructions the external stimulus on the entities is practically non-existent. Results from the second experiment suggest that collapse into a steady state is inevitable without outside influence.

Plants

The entities that survive the initial flurry of activity and persist tend to behave like plants. These plants subsist in areas where energy is plentiful by continuously absorbing energy from the environment. As the energy in the environment depletes a steady state is reached: any energy the entity looses to the environment is immediately reabsorbed. This is an interesting result, but ultimately it points to a flaw in the world design since it renders survival too easy. Finding a way to replace the absorb instruction is not as simple as it may sound, since it presents the only means by which energy released into the environment can be reacquired by entities.

Searching for evolutionary worlds

It is apparent from this experiment that drastically different results can be achieved by altering initial parameters. The rate at which entities die and their behaviour can be altered radically by changing parameters such as the conversion ratio between energy and memory, the maximum number of threads per entity, the initial quantities of energy and memory and the weighting of the instruction set. One idea that might be worth exploring is

the use of a genetic algorithm to search for initial parameters conducive to evolution. In other words, searching for evolutionary worlds using a genetic algorithm.

Better visualisation and interaction

Another problem that is apparent is the need for a much more comprehensive means of visualising and interacting with the world. The original intention was to have a 3d representation (i.e. where the height of tiles is not represented by colour but displacement in the third dimension). Several other additions would be extremely beneficial:

- It would be useful if it entities were easily identifiable.
- A system similar to that used in experiment 2 that allowed investigation (and potentially modification) of an entity's program during execution would be a useful tool.
- Loading and saving of instruction sets, entities and even entire populations/worlds to disc is almost essential if progress is to be made.
- Exposing much of the configuration through a scripting interface would also be extremely beneficial as it would forgo the need to recompile the system each time a change is made.

Closing remarks and acknowledgements

From a personal perspective the progression of this project from the initial ideas to its current unfinished state has involved an enjoyably self-indulgent approach which consisted of pursing the ideas that interested me. This process has inevitably left many rocks unturned, and I hope that, at some stage, there will be time to explore some of these avenues and ultimately develop a working product.
For all that has been achieved to date I am indebted to Eike Anderson for his enthusiasm and support; Michael Beeson for several insightful conversations and Jun Shimoda for the lone of David Attenborough's Life on Earth.

References

[1] Wikipedia, Evolution (Scientific theory) [online], no date. Available from: http://en.wikipedia.org/wiki/Evolution [accessed 5 March 2005]

[2a] Ray, T. Tierra [software], http://www.his.atr.jp/~ray/tierra/ [accessed 5 March 2005]

[2b] Anon. What Tierra is [online], no date. Available from: http://www.his.atr.jp/~ray/tierra/whatis.html [accessed 5 March 2005]

[3] The Digital Evolution Laboratory of Michigan State University, Avida [software], ca. 2001. Available from: http://devolab.cse.msu.edu/ [accessed 5 March 2005]

[4] Dewdney, Alexander Keewatin, Corewar [software], no date. Available from: http://www.corewar.info/, [accessed 5 March 2005]

[5a] newCreationism.org [online], Available from: http://www.newcreationism.org/index.html [accessed 5 March 2005]

[5b] Miller, Kevin The Bugs with The Bugs [online], no date. Available from: http://www.newcreationism.org/CreationArticle15.html [accessed 5 March 2005]

Bedau, Mark, The scientific and philosophical scope of artificial life [online], no date. Available from: http://citeseer.ist.psu.edu/cache/papers/cs/28596/http:zSzzSzwww.reed.eduzSz~mabzSzpaperszSzleonardo.pdf/unknown.pdf [accessed 5 March 2005]