—EmerGen(e)tic— Exploring the use of genetic algorithms in emergent distributed systems

Ben Goldsworthy, 33576556
Computer Science BSc

DECLARATION

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.
I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.
All working documents and results can be found at `www.lancaster.ac.uk/ug/goldswor/scc300/`.

Date:
Signed:

Abstract

Adaptive and emergent systems exist to attempt to answer the deficiencies inherent to distributed systems, and the necessarily finite ability of any programmer to predict all possible eventualities in which his software may one day find itself. This paper argues that these systems fail to go far enough, and then proposes a further development—genetic systems—which utilises genetic programming to extend the versatility of a given system massively, if not infinitely. This paper then proceeds to detail the EmerGen(e)tic framework for rapidly testing genetic algorithm modules within emergent systems, as well as an example module pertaining to the cache updating behaviour of a web server. This paper concludes by proposing further avenues of potentially-fruitful research based upon these programs and its findings.

1 INTRODUCTION

Coulouris *et al.* (2012) define a *distributed system* as '[...]one in which components located at networked computers communicate and coordinate their actions only by passing messages.' [1] By the very nature of distributed computing, any system will—as a whole—experience a range of fluctuating conditions and environments that may affect its performance. Although a well-designed, reliable distributed system will be developed to be 'as fault tolerant as possible' [2] and with as many of these potential operating environments as possible in mind, the totality of all possible conditions for all possible futures in which the system remains in use is impossible for the original developer(s) to have been entirely predicted and accounted for. This inhibits the universality of current distributed systems, as well as incurring future maintenance and development costs when they need to be adapted for future circumstances.

Adaptive systems are a subset of distributed systems in which the system is given a number of submodules that define different behaviour, and the system is programmed with policies for determining which submodule to use for a given task. An example may be a file server that has a compression module activate when serving a file over a given filesize. Whilst this adaptive behaviour does increase the system's operational range, it still requires manual programming of policies and thus suffers from the same issues as the general distributed systems.

Emergent systems are a proposed solution to these limitations. Emergent systems are a subset of adaptive systems in which the system itself determines the best configuration to deal with a given situation by trying multiple configurations and assessing their performance against a given metric. This, coupled with some form of storage for recording previously-discovered optimum solutions that can be consulted heuristically for improved performance, provides a more adaptive final system whilst simultaneously reducing development overheads.

Filho & Porter (2016) best outline the problem in response to which emergent systems have been developed, stating that (emphasis theirs) '[human-centric] approaches [to self-organising software] rely on prediction of how a system will respond to environments (which may turn out to be false) and result in *inflexibility* [whilst a] *machine-centric* approach [...] moves the burden of complexity into software itself, avoids the need for prediction of behaviours, and supports total flexibility [...]'. [3]

However, these emergent systems are not too without their limitations. Whilst they do possess far more flexibility with regards to arranging themselves into new and unpredicted configurations than does a simpler adaptive system, they are yet constrained by the finite pool of components they have access to, which are still developed by human hand.

Genetic algorithms rely on metaphors from evolutionary biology such as random mutation and natural selection in order to develop—over the course of multiple iterations and utilising a finite set of operations—improved software, with each generation ideally moving closer towards an optimal solution as specified via the result of a given fitness function $f()$. They have been used to produce everything from spaceship antennae [4] to software bug checkers. [5] In a previous work, [6] this author applied genetic algorithms to the development of simulated ants. Algorithm 1 shows the algorithm by which genetic programming is performed.

On the one hand, we have systems that need to be freed from human development as much as possible in order to be generalisable to the widest range of possible operating environments. On the other, we have a programming approach in which the human developer sets up a number of variables before reclining whilst the computer handles the rest of the time-consuming programming work—the two fields would appear to be ready allies. If genetic programming could be effectively applied to the generation of components

Algorithm 1 Basic genetic algorithm

```
Require: initial population
Require: num. of generations
Require: num. of candidates per generation
  for all generations do
    for all chromosomes in generation do
      run test on chromosome
      return fitness score of chromosome
    end for
    for strongest chromosome(s) in generation do
      copy over to next generation
    end for
    for remaining chromosomes in generation do
      if small probability = true then
        apply mutation/crossover operation
      end if
      copy over to next generation
    end for
  end for
```

for an emergent software system, it would allow the creation of an infinite pool of potential components and allow the system to theoretically surmount any given situation, given enough time.
Thus, the research questions that this project aims to satisfy are:

- can genetic algorithms be productively used to produce optimum components for an emergent system? and
- assuming so, what are the optimal conditions for doing so (i.e. mutation probabilities, generation sizes, etc.)?

From these research questions come the aims for this project; they are:

- to produce a framework for enabling rapid development of genetic improvement tests,
- to test a small, simple program to hopefully demonstrate improvement through application of these techniques and
- to repeat the tests multiple times with a variety of conditions and see what, if any, trends emerge.

This report is divided into eight sections:

1) 'Introduction', in which the broad goal of the project in question is outlined,
2) 'Background', in which the history of the field in question is outlined in more detail,
3) 'Design', in which the design of the present solution is outlined in a non-technical manner, from first principles,
4) two 'Implementation' sections, in which the technical implementation of both the EmerGen(e)tic genetic testing framework and the specific `cachingpolicy` module is detailed,
5) 'The System in Operation', in which usage—and normal output—of the systems above are described,

6) ‘Testing & Evaluation’, in which the results of testing on the systems above are investigated and the project as a whole evaluated, with its successes and failures candidly discussed and
7) ‘Conclusion’, in which the research questions above are revisited and avenues for future research signposted.

2 Background

2.1 Assumptions

This project operates on a foundation of two assumptions, neither of which—though they may both appear to be—are axiomatic.

Assumption #1: Programming is hard

IBM found that the cost of fixing a bug can range from a base cost (for one discovered during the requirements elicitation phase of software design) to 100× that (for one identified in the maintenance phase). [7] With the lengths of written software steadily increasing—1993's Windows NT 3.1 had 4.5m lines of code to 2003's Windows Server 2003's 50m—and an average of 'about 15–50 errors per 1000 lines of delivered code', [8] these problems are being exacerbated.[1] As such we desire to systems that will perform as many programming responsibilities on our behalf as possible, as well as as much of the checking, testing and validation work as we can get away with—repetitive, methodical tasks such as testing especially represent the perfect area for a computer, but not man. We also desire systems that can be constructed out of smaller, simpler parts that can be more rigorously tested prior to deployment.[2] It is these goals that has led to the development of distributed, adaptive and emergent systems, as well as genetic programming.

Assumption #2: Circumstances change

It would be hubris *par excellence* to assume that humanity has learnt all that there is to learn and can be surprised no longer. Bronze age man discovered ironwork, assumptions of classical physics were rent asunder by the discovery of quantum physics and the discovery of non-Euclidean geometry is said to have 'marked the end of an entire line of human thought, one that had dominated intellectual efforts in the West for centuries.' [11] No one programmer—nor a team of programmers—can hope to have perfect, 20/20 foresight encompassing all of the environmental possibilities in which their system may find itself deployed in the future. Even if they could, theoretically, handle all possible situations as are known to them now, there is every possibility of a non-Euclidean geometry-esque upending of the conventional wisdom occurring at an unspecified point in the future, entirely incomprehensible from within the bounds of their Euclidean worldview—they would be akin to the denizens of Abbott's *Flatland*, trying to conceive of the third dimension.

2.2 Distributed and Adaptive Systems

Distributed systems—amusingly described by Lamport (1987) as '[a system] in which the failure of a computer you didn't even know existed can render your own computer unusable' [12]—have their origins in the early days of computing and the concept of multiprogramming. Carr, Crocker & Cerf (1970) outlined ARPANET, 'one of the most ambitious computer networks attempted to date'. [13] As one of the first networks, ARPANET serves an example of one of the first instances of distributed computing and a predecessor to the modern-day Internet. Coulouris *et al.*'s definition of a distributed system as '[...]one in which components located at networked computers communicate and coordinate their actions only by passing messages' [1] may, however, begin to make the drawbacks of such a system apparent. As these computers are in different geographic locations, their needs and experiences may differ—a program written in California may

1. For further exploration by myself of the resultant security risks of this, amongst other factors, see [9].
2. cf. the 'Unix philosophy', best detailed in [10]

work fine on the fast, reliable internet connections that are all its developer may have ever experienced, but less so for the instance of it running in rural India.

One solution to these issues is *adaptive systems*, in which a distributed system is given a number of submodules containing different approaches to its tasks. The developer will then program in various policies for which modules to use in which conditions. Examples of this could include a live online video player that resorts to a different quality of transmission when being broadcast to a client with a suboptimal network connection. Again, the Internet is an excellent example of an adaptive system: with various protocols on offer, a developer can choose the one that best suits a given need. If error detection and correction is important, TCP is the tool for the job; if it is not, then UDP provides a faster transmission with less overhead.

However, these adaptive systems fail to resolve the initial issue of the developer having to predict the myriad situations his program may find itself running in in the future. By still requiring the developer's input in specifying the various module combination strategies and policies, all these systems achieve is to allow the developer to write more generalised, less specific (sub)programs. This may be beneficial for rapid reuse elsewhere, but is otherwise not the solution we seek.

2.3 Emergent Systems

One such proposed solution, however, is the *emergent system*, or an adaptive system in which the program itself (or a framework on top of it) designs its own policies based on trial-and-error and recording performance rates of different combinations of submodules, the name coming from the possibility of such a system discovering optimal behaviour unpredicted by its human creators.

Dana[3] is a component-based language designed specifically for emergent system programming by Porter (2014), described by the author as 'an imperative, procedural, interpreted language, [that] is multi-threaded, and features only interface, record and primitive types' [14]—it is also 'syntactically similar to contemporary languages like Java' [14].

Filho & Porter (2016a) describe a web server written in Dana.[4] The web server receives requests for files—a mixture of file types and sizes—from clients and serves them up. The server can construct behaviour out of a number of components that enable different policies for cache updating, file compression, etc. The authors test the server in every possible configuration and against different styles of workload, measuring the request response time per configuration.

The authors showed that different configurations gave markedly different results—for example, for a 'Workload 1 [consisting] of one client repeatedly requesting only one text-only HTML file, [a particular architecture] performs best because, in this configuration, the web servers always compress the requested files, and once the file is returned to the load balancer, it is stored in a small content cache at the load balancer.' [3] Meanwhile, a cacheless architecture performed better at a workload consisting of 'one client requesting a different text-only HTML file for every request' [3] as caching would provide no benefit for a constantly-changing series of requests and would only add pointless overhead.

In a further paper, Filho & Porter (2016b), the authors go onto elucidate the limitations of a simple adaptive system, arguing that '[the] requirement [for human specification of control strategies] is fundamentally opposed to the core ideas behind autonomic computing, which are borne of the increasing difficulty for humans to understand modern software systems in dynamic environments.' [15] Developing further their web server platform, they implement

3. Dana documentation and downloads can be found at `www.projectdana.com`

4. For an explanation of the function of Filho & Porter's web server, as well as concepts such as 'caching' and 'compression', see §3

a perception and learning system so that the web server can analyse the performance of varied configurations and determine an optimum arrangement for a given task. Finally, in Filho *et al.* (2016), this is consolidated into the single framework RE^{X}, which 'produce[s] systems that are responsive to the actual conditions that they encounter at runtime, and the way they perceive their behaviour in these conditions.' [16]
Other frameworks within the same field of emergent systems exist, such as Elkhodary, Esfahani & Malek (2010)'s FUSION, [17] but I assert the following with this paper: that the very idea of emergent systems as a whole fails to go far enough towards solving the issues it was formulated in response to.

2.4 'Genetic Systems'

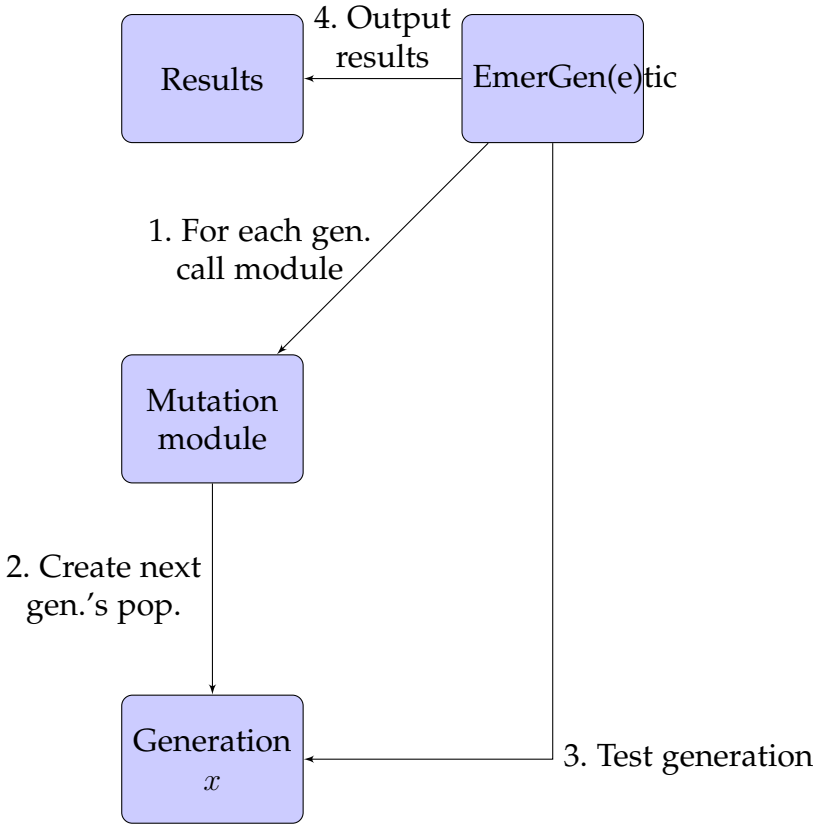
Floyd (1979) said that the following words were written on the wall of a Stanford University graduate student office: 'I would rather write programs to help me write programs than write programs.' [18] This is a succinct description of the goals of genetic programming, in which representations (usually tree representations) of computer programs are modified using a set of mutation and crossover operators and a set of varying probabilities for each to occur. These changes take place across multiple generations and should, hopefully, trend towards better-performing programs. The applications are vast: Le Goues *et al.* (2012) applied them to automated software bug fixers; [5] Hansen *et al.* (2007) to counter-cyberterrorism; [19] Hornby *et al.* (2015) to the shape of spacecraft antennae; [4] and the author of this paper to the production of simulated ants. [6]
Indeed, the wall of Stanford's student office neatly encapsulates the thinking behind the proposal of this paper as to the viability and desirability of applying genetic programming techniques within the emergent system space–what this author will propose to be described as 'genetic systems'. If an emergent system can already assemble configurations of provided submodules, test them and rank them, and the goal is to eliminate the developer from the software development equation as much as possible, then what is next? Why not have the system assemble the submodules themselves? This would increase the pool of potential submodules available to the program to be theoretically infinite, and the same benefits of this emergent behaviour would apply here too. It is this hypothesis that is to be tested within this paper.

3 DESIGN

Within this section the design of the EmerGen(e)tic (from 'emergent' and 'genetic') system is outlined. Following that, the web server and caching system used within the `cachingpolicy` module are outlined, from first principles, for the non-technical user's understanding. For a technical overview, see §5.

3.1 EmerGen(e)tic

The primary goal of this project was to produce a framework for quickly and easily allowing future projects to test the effects of genetic algorithms on varying elements of varying systems. To do this, a master Dana program was created which runs through each generation, calling a separate file (hereafter referred to as a *module*) to handle all of the the evolutionary logic—the mutations, crossovers, selections, etc. This is so that future developers can easily plug 'n' play with their own genetic modification modules in the future. The master program will then run through all the candidates from each generation to test their responses to differing stimuli. The testing functionality is contained within a single method, and so should be the only part of the EmerGen(e)tic code that a future developer needs to modify to implement their own project.
To assist with generalisability, as many settings as possible (e.g. mutation probability, file(s) to test with, number of generations) were left to be externally set via either command-line arguments or config. files.
The below flowchart shows the intended operation of the system:

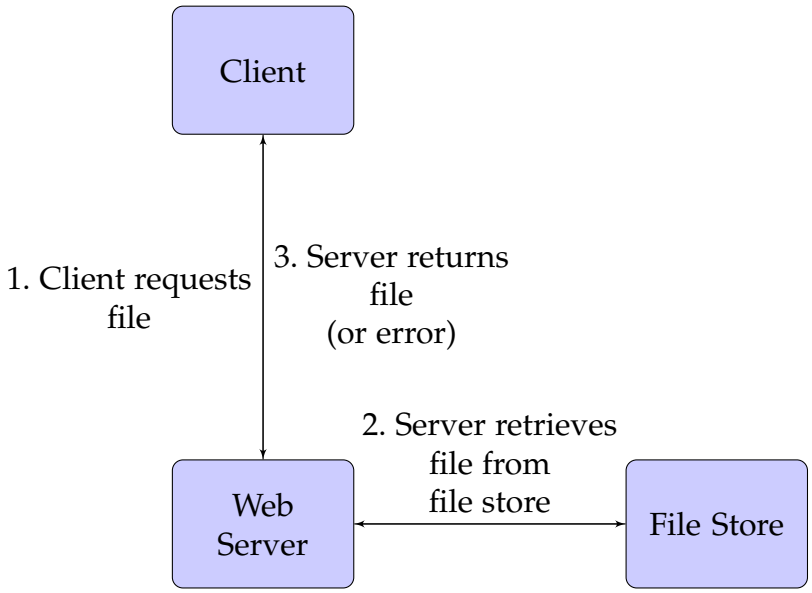


3.2 The Web Server

3.2.1 How a web server works

A *web server* is a type of computer. It can receive requests from other computers (called 'clients') for the files that are located within it via a number of methods, usually across an

Internet connection, such as HTTP. Once it has received a request, which will contain the name of the file requested, it searches its file storage for a match. If it finds it, it sends the data of the file back to the client. If it does not, it should return a message that the client knows to interpret as an error message. Below is a flowchart of how a web server works:



3.2.2 Added extras

This is the most basic form of web server—for every request, it trawls through its files for a match and then returns that to the client. However, there are numerous extra functions that can be added to improve performance, and which can be combined to best suit various situations. For example, files can be compressed through various schemes in order to reduce their filesize before sending, and then decompressed at the client's end, which shortens the amount of time taken to transmit the (now-smaller) file.
However, the one most relevant to this project is *caching*. In caching, a cache is kept—this is a section of storage that is faster to retrieve data from than the regular file store, but which does not have the size to hold the full file store. When a file is returned for a client's request, it is also added to the cache. This means that if a client then requests the same file again, it will be found within the cache and returned faster than if the entire file store had to be searched.
Obviously, the cache will at some point fill up. At this point, various policies for updating a full cache can be implemented. The simplest is to simply start again at the beginning and overwrite the first item. More complex ones can bring in variables like which file was requested least recently, or has been requested the fewest times, in order to try and improve speed. Again, different policies suit different situations, and there is no 'magic bullet'.

3.2.3 A tale of two web servers

The present project is based off of Filho & Porter's Dana web server code. Their project includes a client program, which is given a list of files grouped by various criteria (e.g. all large files, all image files, mixtures of both, etc.). It then sends the server program HTTP requests for each file in the list until it reaches the end. In this particular version of the software Filho & Porter's PAL system is implemented, meaning every possible configuration found is tried—this means that each file is tested with every combination of the provided caching and compression methods, and the response time of the server (in ms) recorded. This response time is the metric by which the PAL system ranks various

configurations.
The goal of Filho & Porter's project was to demonstrate improvements in file handling when using different combinations of policies for different types of request patterns. However, for the present project, the system is too complex to control all of the independent variable required for the tests to be run. As such, the scope of inquiry had to be limited to caching policy only. As such, much of Filho *et al.*'s server project code was removed. This left only the caching functionality and some remnants of the web server code.

3.3 Caching

Cache behaviour is located within the `/cache/CacheHandler*.dn` family of components. Filho & Porter's server code contains six `CacheHandler*` variants, each with different policies for determining which item in a full cache—represented by a fixed-size array—to replace with a newly-requested file, as well as a simple cache system in which the cache is only one item long. The other methods are:

- serial replacement, where each item in turn is replaced, returning to the first item upon reaching the end of the array,
- most- and least-frequently used,
- most- and least-recently used.

For example, `CacheHandlerMFU.dn` contains the policy for replacing the most-frequently used cache item.
Within each of these variant cache handling policy files there is necessarily a lot of duplicated code. Common to all variants are the method signatures (but not bodies) for:

- `updateCache()`, which updates a full cache using the variant's specified policy,
- `clearCache()`, which clears the cache completely, and
- `getCachedResponse()`, which returns either the item that was received from the cache or `null` to the caller.

Cache initialisation is handled as a conditional within `updateCache()` (i.e. if there is no cache present, it creates one and places the current item in cache index 0).
There are also a series of functions that are only present in individual `CacheHandler` files, such as `mostRecentlyUsed()` in `CacheHandlerMRU.dn`. This does what is says on the tin, returning the most recently-used item from the cache to be overwritten. Another example is `random()` (within `CacheHandlerR.dn`), which unsurprisingly returns a random item from the cache.
For the genetic algorithm to work it needs a base file. The base file requires all of the common functionality present and marked as out-of-bounds for the genetic algorithm. Included in these out-of-bounds methods would be the full complement of methods such as `mostRecentlyUsed()`, which could be called upon or not as decided by a given output program.

3.4 The 'Cache Policy' Genetic Algorithm

The basic workings of a genetic algorithm were covered in §1, but here the specifics of the algorithm as applied to this project shall be outlined.
As this project is concerned only with modifying cache updating behaviour, and thus in how to produce an index value given differing formulae, the most important line is line 110 of `CacheHandlerBase.dn`:

```
index = 0
```

The formula on the right-hand side of that assignment is the scope for mutation. From that initial '`0`' must sprout new formulae—these formulae are what is referred to in the remainder of this report as *chromosomes*. A single *generation* will consist of multiple `CacheHandler*.dn` files (or *candidates*), each of which will contain a single chromosome. Within each generation, therefore, shall exist multiple distinct chromosomes. Each of these sets of chromosomes is known as the generation's *population*.
As mentioned before, the basic tools of a genetic algorithm are the operations of *mutation* and *crossover*, which shall be discussed below (along with a heuristic known as 'elite selection'):

Mutation

In mutation, a section of chromosome A is replaced with a new value. For example, from the example chromosome '`0`', an operand mutation operation could change it to '`4`'. For the mutation within the scope of this project, four types of mutation were implemented:

- binary operator mutation (e.g. '`2+4`' into '`2*4`'),
- operand mutation,
- unary operator mutation (e.g. '`nthMostRecentlyUsed(2)`' into '`nthMostFrequentlyUsed(2)`'), and
- subtree creation (e.g. '`2+2`' into '`2+(4*2)`').

The first three simply randomly change elements of the chromosomes in the hopes of finding particularly efficacious variants. The fourth, however, serves the vital role of allowing the chromosomes—which all start as '`0`'—to increase in complexity.

Crossover

In crossover, two chromosomes A and B are selected. A random section of A (A') is then taken and inserted at a random point of B (B') to produce a new chromosome C. For example, take

$$A = \texttt{2+(4+3)}$$
$$B = \texttt{4*3}$$

and an A' of '`+(4+3)`' and a B' between the '`4`' and the '`*`'—we produce

$$C = \texttt{4+(4+3)*3}$$

Variants of crossover also exist, such as multi-point crossover (where multiple A's are copied over to multiple B's). However, for this project, only single-point crossover was implemented due to time constraints.

Elite Selection

There are a number of time-saving tactics (or *heuristics*) that can be used to speed up the performance of a genetic algorithm. One such heuristic, implemented within this project, is *elite selection*. This takes an arbitrary percentage of top-performing chromosomes for any generation (in this instance, the top 10 %) and copies them across to the next generation unchanged. This ensures that optimal solutions are not simply discarded upon the next generation.

4 IMPLEMENTATION: EMERGEN(E)TIC

This section contains a more technically-detailed overview of the implementation of the EmerGen(e)tic program.

4.1 Folder Structure

The base folder structure for EmerGen(e)tic is as follows:

- `/archives/`, where archives of test run data go when after completion of a script's run,
- `/project/`, which contains the mutation project modules,
- `/resources/`, which contains Dana overhead,
- `/results/`, which contains per-generation and per-script results files and
- `/scripts/`, which contains `.script` files for running through multiple tests, grouped by some criterion/criteria.

4.2 `emergenetic.dn`

This Dana file contains the framework for running and testing genetic algorithm projects.[5] Passed to it are a number of command-line arguments, including the module to load and the number of generations to test.

4.2.1 *`App:main()`*

The program is entered through the `App:main()` method, whereupon the passed arguments are validated—if any are invalid, detailed error messages are output and the program terminates. The method runs the setup file `setup.sh` from within the passes module directory (and passes the result to a variable of type `RunStatus`, which can be helpful for debugging), and then iterates through each generation. For each generation, it runs the given module's `mutator.py` program, followed by the `runGeneration()` method.
Upon successfully finishing the iteration of each generation, the program exits with a code of '0'—in the event of invalid arguments, it exits with a code of '1'.

4.2.2 *`runGeneration()`*

The `runGeneration()` calls the `runCandidate()` method for each candidate within the given generation. Once this is done, it appends a newline to the `results.csv` file within the `results` directory.

4.2.3 *`runCandidate()`*

This is the point at which a future developer will implement their own code for running and testing candidates for their own projects. In the base release of `emergenetic.dn`, it is left blank.

4.2.4 *`printResults()`*

This method prints the results of each test run to output. It then also appends the results of the run to the per-generation results file `results`x`.txt`, and to the per-script results file `results.csv` within the `results` directory.

5. See Appendix A

5 IMPLEMENTATION: cachingpolicy

In this section, the technical implementation of the `cachingpolicy` module used for this project is detailed.

5.1 Folder Structure

In addition to the base folder structure of EmerGen(e)tic, the following directories were added.

- `/cache/`, where the generations of various `CacheHandler*.dn` files are kept, each within a directory numbered with their generation number and in files of the format `CacheHandler`x`_`y`.dn`, where x is the number of the generation and y is the number of the candidate,
- `/cachebackup/`, which contains a fresh copy of `CacheHandlerBase.dn` for quickly clearing the `/cache/` directory after a full test run,
- `/htdocs/`, which contains all the files to request from the web server,
- `/project/cachingpolicy`, which contains the files for the `cachingpolicy` module and
- `/resources/cache`, which contains the Dana specifications for the caching components.

5.2 Genetic Algorithm search space

Before a genetic algorithm can be implemented, the search space must be defined:

$$\text{Let } c \text{ represent the cache length}$$
$$\text{Let } l \text{ represent an AST leaf values}$$
$$\text{Let } n \text{ represent an AST node values}$$

$$l \in \mathbb{N}$$
$$n \in \{\times, +, -, \div, MostFrequent\,(), MostRecent\,(), Rand\,()\}$$

$$\text{Let } a\,(n) \text{ be a function that returns the arity of function } n$$
$$\text{Let } r \text{ represent the result of } n\,(), n\,(l) \text{ or } n\left(l_1, \ldots, l_{a(n)}\right)$$
$$\text{Let } i \text{ represent the cache index value to replace}$$

$$r \in \mathbb{N}$$
$$i \in \{x \in \mathbb{N} : 0 \leq i \leq c-1\}$$
$$\therefore i = r \bmod (c-1)$$

In real terms then, the the set of actual Dana operators and methods n_{Dana} used was

$$n_{Dana} \in \{\texttt{*}, \texttt{+}, \texttt{-}, \texttt{/}, \texttt{nthMostFrequentlyUsed()}, \texttt{nthMostRecentlyUsed()}, \texttt{random()}\}$$

and the resolve flag res, which was passed through to `nthMostFrequentlyUsed()` and `nthMostRecentlyUsed()` as a constant in each individual program, was of the value

$$res \in \{\texttt{n}, \texttt{o}, \texttt{r}\}$$

5.3 `CacheHandlerBase.dn`

From the various `CacheHandler*.dn` files within Filho & Porter's `/cache/` were distilled the common elements of each (the methods covered in §3.3), collected into one file.[6]

5.3.1 *`CacheHandler:getGetCachedResponse()`*

This method returns an item specified in a request from the cache if it exists there—otherwise, it returns `null`.

5.3.2 *`CacheHandler:updateCache()`*

This method updates the cache if a requested item is not already in it. If the cache is full, this method contains the formula for determining the cache index to replace. Lines 70–71—demarcated with the comments `//BEGIN` and `//END`—indicate to the genetic algorithm its bounds of operation. Within these lines, the method for determining cache index i is determined for each chromosome.

5.3.3 *`CacheHandler:clearCache()`*

This method clears the cache completely.

5.3.4 *`nthMostFrequentlyUsed()`*, *`nthMostRecentlyUsed()`* & *`random()`*

These methods return the item in the cache that fits their criterion (or, in the case of `random()`, a random item). Including all of the various most/least frequently/recently used methods would lead to an unnecessarily large file, so they were generalised into the methods `nthMostFrequentlyUsed()` and `nthMostRecentlyUsed()`, which each take their n as an argument.

5.3.5 *`resolve()`*

The `resolve()` method takes a flag indicating how to resolve the situation of multiple returns from one of the aforementioned three methods (e.g. in the event that the least-frequently used file is not the only file in the cache with that number of hits). The flag can be set to return either the newest item, the oldest or a random choice.

5.4 `emergenetic.dn`

Within the `runCandidate()` method,[7] the `CacheHandler` component within `CacheHandler*.dn` is loaded using Dana's `RecursiveLoader`, which ensures that all of its dependencies are also loaded. A timer is started and each file within the given script file is requested from the web server. When the script file is finished, the timer is stopped and the overall time taken (and that the value represented ms) is passed to `printResults()`. As `RecursiveLoader` lacks an `unload()` method, each of the loaded components is then looped through and `unload()` called on them in turn.

5.5 `cachingpolicy/mutator.py`

This Python file contains the genetic algorithm logic for the `cachingpolicy` module.[8] It can be run as a standalone Python script or imported as a Python module into other projects.

6. See Appendix C.1
7. See Appendix C.3
8. See Appendix C.2

5.5.1 `getSubLists()`

This method recursively produces a `List` of nested `List`s representing every subtree present within a given chromosome (i.e. every bracketed expression).

5.5.2 `crossover()`

This method performs single-point crossover between two chromosomes passed to it as arguments A and B. It first calls `getSubLists()` on A in order to get a `List` of all the possible subtrees of A. It then randomly chooses points within B until it finds an operand token, at which point it replaces the operand with a randomly-chosen `List` from A.

5.5.3 `mutate()`

This method performs four distinct multiple-point mutations. If it is the first run it imports the individual mutation probabilities from `config.conf`, and if it is the initial population it overrides the mutation probability to be 100 %. It then recursively iterates over each token in the chromosome and has a chance to possibly apply one of the mutation operations. The recursive element comes in when it encounters a token of type `List` (i.e. a bracketed expression), in which case it calls `mutate()` upon the sublist.

5.5.4 `parse()` & `compile()`

The former of these methods takes an expression as a string (e.g. '`2+(4*2)`') and returns a `List` representation (with nested `List`s for bracketed expressions). The latter takes such a `List` and returns the original string expression.

5.5.5 `createInitialPop()`

If the python script is called with a generation argument of '0' it calls the `createInitialPop()` method first. This creates an initial population by copying over `CacheHanderBase.dn` and applying `mutate()` to each candidate with a 100 % mutation probability.

5.5.6 `readChromosomeFromFile()` & `writeChromosomeToFile()`

These methods are simple utility methods for reading and writing chromosomes from and to `CacheHandler*.dn` files.

5.5.7 `hasSubTrees()`

This method returns a boolean indicating whether a passed chromosome has nested expressions or not.

5.5.8 `main()`

The `main()` method takes the n candidates of previous generation m (passed to it as arguments) and selects the top 10 % of them to go across unchanged to generation $m + 1$. Of the remainder, probabilities of mutation and crossover are applied to each and, when those occur, separate probabilities determine the type of mutation or crossover that will occur.

5.6 `cachingpolicy/config.conf`

`mutator.py` reads in its probability values from the file `config.conf`.[9] This file contains a number of name-value pairs. The left hand side of each config. pair is ignored by `mutator.py` and is thus present only for the aid of the human user.

9. For an example `config.conf`, see Appendix C.5

6 THE SYSTEM IN OPERATION

In this section is outlined the process of creating a new module—and running a suit of tests using it—via the EmerGen(e)tic framework.

6.1 Creating a Module

6.1.1 Files

To create a new module for a future research project, create the folder within the `project` directory. Within this directory, create a Python file `mutator.py`, which will contain all of your genetic algorithm and a Bashscript file `setup.sh` to perform any functions prior to each test such as creating or deleting folders (this may not perform any tasks, but the file must still be present).[10]

6.1.2 Implementation

To implement the new module and test the results of it, code must be written within the `runCandidate()` method of the Dana file `emergenetic.dn`. Within this method will be defined the process of testing, how to use script files and the metric by which success is measured. The other methods within the file will remain unchanged.

6.2 Creating Scripts

In order to run a suite of different tests for each candidate, scripts are required. In the `cachingpolicy` instance, these script files contain the filenames of different files grouped by some property or properties. For example, `difffile-html.script` tests a number of different files, all of which are HTML but are a mixture of sizes, whilst `difffile.script` tests different files that are a mixture of HTML and image and have a range of sizes. This can be used to easily showcase the system producing different results that provide optimum behaviour based on different operational conditions.

6.3 Running EmerGen(e)tic

After compiling the `emergenetic.dn` file, the program can be run. It expects four command-line arguments, and can take a fifth optional one. The command is as follows:

```
dana emergenetic <module> <script> <generations> <candidates> [verbose]
```

`<module>` is the name of the module folder that you wish to use for your tests.
`<script>` is the name of the script file for this particular test, located within the `/scripts/` directory and without the trailing `.script` file extension. For example, "`difffile`" for the script file `difffile.script`.
The `<generations>` and `<candidates>` arguments are the number of generations to run the test for, and the number of candidates to produce per generation.
The optional `verbose` argument, if present, produces far more verbose output text. This can be useful for debugging.
For example, below is the console output for a typical run of the `cachingpolicy` module (with the `verbose` flag omitted):

```
ben@metacom-1:~/emergenetic$ dana emergenetic cachingpolicy difffile
    10 10

Response time for configuration 0x0: 19 ms
Response time for configuration 0x1: 20 ms
```

10. For an example `setup.sh`, see Appendix C.4

```
Response time for configuration 0x2: 20 ms
Response time for configuration 0x3: 19 ms
Response time for configuration 0x4: 20 ms
Response time for configuration 0x5: 19 ms
Response time for configuration 0x6: 20 ms
Response time for configuration 0x7: 19 ms
Response time for configuration 0x8: 19 ms
Response time for configuration 0x9: 20 ms

Response time for configuration 1x0: 20 ms
Response time for configuration 1x1: 20 ms
Response time for configuration 1x2: 19 ms
Response time for configuration 1x3: 19 ms
Response time for configuration 1x4: 19 ms
Response time for configuration 1x5: 19 ms
Response time for configuration 1x6: 20 ms
Response time for configuration 1x7: 20 ms
Response time for configuration 1x8: 19 ms
Response time for configuration 1x9: 19 ms

Response time for configuration 2x0: 20 ms
Response time for configuration 2x1: 19 ms
Response time for configuration 2x2: 20 ms
Response time for configuration 2x3: 20 ms
```

...and so on. Here is a sample of console output with the `verbose` flag included:

```
ben@metacom-1:~/emergenetic$ dana emergenetic cachingpolicy difffile
   10 10 verbose

Response time for configuration 0x0: 20 ms
Response time for configuration 0x1: 19 ms
Response time for configuration 0x2: 19 ms
Response time for configuration 0x3: 19 ms
Response time for configuration 0x4: 19 ms
Response time for configuration 0x5: 20 ms
Response time for configuration 0x6: 20 ms
Response time for configuration 0x7: 19 ms
Response time for configuration 0x8: 20 ms
Response time for configuration 0x9: 20 ms

Applying rank selection to top 10% from prev. generation...
      Result: 0_2 copied to 1_2

Mutating 0_3->1_3:
      Mutating: 0
            Result: 0*0
      Mutating: 0*0
            Result: 0/0
      Mutating: 0/0
            Result: 0/random()
0_3->1_3 mutation finish.

Mutating 0_4->1_4:
      Mutating: 0
            Result: 0*1
      Mutating: 0*1
            Result: 0*1
      Mutating: 0*1
            Result: 0*nthMostFrequentlyUsed(0)
```

```
0_4->1_4 mutation finish.

Mutating 0_5->1_5:
      Mutating: 0
            Result: 0-1
      Mutating: 0-1
            Result: 0-1
      Mutating: 0-1
            Result: 0-17
0_5->1_5 mutation finish.

Response time for configuration 1x0: 19 ms
Response time for configuration 1x1: 29 ms
```

...and so on.

6.4 Background Tests

The process can be run as a background process so that the Terminal window/SSH connection can be closed by the user without interrupting the tests. The user can either run the EmerGen(e)tic program itself as a background process, or write a script to run.[11] To run the process in the background, use the commands

```
dana emergenetic cachingpolicy difffile 10 10 &>/dev/null &
disown
```

By redirecting output to `output.txt` rather than `/dev/null`, the user can also track the progress of the tests as they happen.

11. An example script can be seen in Appendix D

7 TESTING & EVALUATION

In this section is outlined the results of various testing of the system. Following that, the successes and failures of the project will be reviewed and how well it satisfies the research goals outlined in §1 assessed.

7.1 Testing EmerGen(e)tic

The first test was one to make sure that the EmerGen(e)tic program was working as expected. Using a very simple `helloworld` module (that simply took a `.txt` file containing the line 'Hello world' and applied a random chance of mutating the letters into others), three tests were run.[12]
The first test, with mutation probability set to 50 %, produced the following output:

```
['Hello world\n', '1_0']
['Hello world\n', '1_1']
['Hello world\n', '1_2']
['Hekjo joaky', '1_3']
['Hello world\n', '1_4']
['hiblm wokly', '1_5']
['Hello world\n', '1_6']
['Hello world\n', '1_7']
['Hello world\n', '1_8']
['yeclo wdvld', '1_9']

['jmdlo warld', '2_0']
['Hello wtrld', '2_1']
['lejlodyowld', '2_2']

       [...]

['Hello wtxid', '6_3']
['Heplo wcrld', '6_4']
['vvkqo wnrzi', '6_5']
['qello worfd', '6_6']
['Hglyo wveld', '6_7']
['Hlulowwdglj', '6_8']
```

As you can see, mutations were successfully applied to some (but not all) candidates on each generation's run, with the generations further from 0 being progressively less recognisable from their origins.
In order to test that the system worked too with varying probability values, tests were run with probability values of 0 % and 100 %.
For 0 %, the results were as follows:

```
['Hello world\n', '1_0']
['Hello world\n', '1_1']
['Hello world\n', '1_2']
['Hello world\n', '1_3']
['Hello world\n', '1_4']
['Hello world\n', '1_5']
['Hello world\n', '1_6']
['Hello world\n', '1_7']
['Hello world\n', '1_8']
```

12. See Appendix B

```
['Hello world\n', '1_9']

['Hello world\n', '2_0']
['Hello world\n', '2_1']
['Hello world\n', '2_2']
['Hello world\n', '2_3']
['Hello world\n', '2_4']
['Hello world\n', '2_5']
['Hello world\n', '2_6']
['Hello world\n', '2_7']

       [...]

['Hello world\n', '6_3']
['Hello world\n', '6_4']
['Hello world\n', '6_5']
['Hello world\n', '6_6']
['Hello world\n', '6_7']
['Hello world\n', '6_8']
```

This shows that no mutation operations were applied.
For 100 %, the results were as follows:

```
['ampcjlcutjs', '1_0']
['xktieosrnyx', '1_1']
['caxedgtiyup', '1_2']
['cqyiujnyapp', '1_3']
['wnkwuyamxxz', '1_4']
['rfotmkuheyv', '1_5']
['eveudzvxhzj', '1_6']
['ftgxinfdomw', '1_7']
['mdjomfeoggy', '1_8']
['fibdllxjmox', '1_9']

['pwfumdlcauk', '2_0']
['bfoxwhovtgm', '2_1']
['gqpztkejijs', '2_2']
['drxavuclcka', '2_3']
['wiycvultuli', '2_4']
['uyxeayumsdc', '2_5']
['mubbtclbees', '2_6']
['xezoaznywmq', '2_7']

       [...]

['skxqhsirryo', '6_3']
['ajoexxutuhd', '6_4']
['umwormhnxlz', '6_5']
['htvrrhqgvzl', '6_6']
['pzitdowgnjg', '6_7']
['aikbtvejkvr', '6_8']
```

This shows that the mutations were applied to every single letter of every single candidate. From these, EmerGen(e)tic is shown to be able successfully implement genetic algorithms and tests thereof. Also, the speed at which modules can be developed is demonstrated: `helloworld` took around 15 mins to write.

7.2 Testing `cachingpolicy`

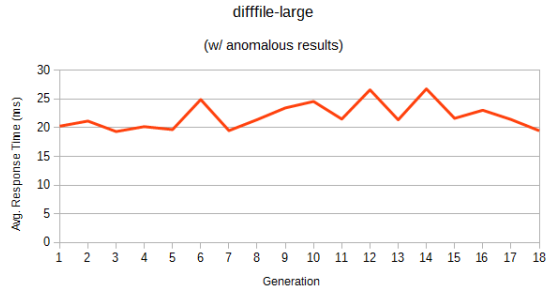


Fig. 1.

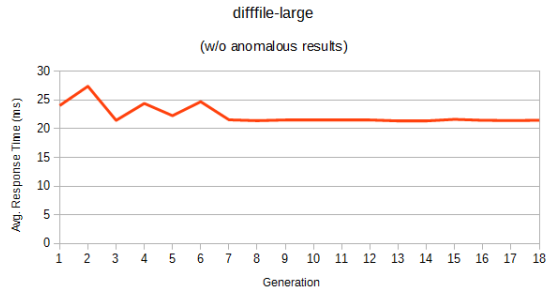


Fig. 2.

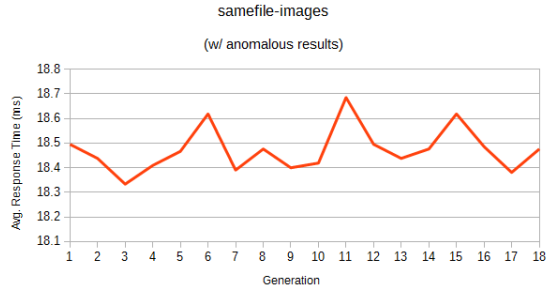


Fig. 3.

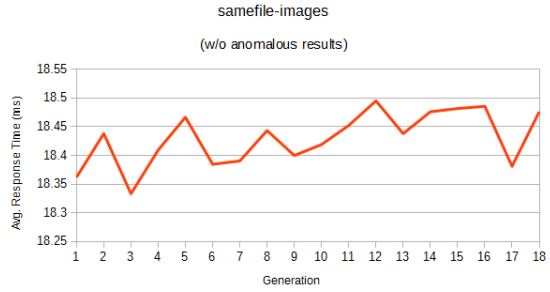


Fig. 4.

With EmerGen(e)tic itself having been demonstrated to work, answering the research questions posed in the introduction to this project could be attempted. The first question was phrased 'can genetic algorithms be productively used to produce optimum components for an emergent system?'

To test this hypothesis, the initial probability variables in `config.conf` were set to the values shown in Appendix C.5—the probability of a complexity-increasing mutation was set so high in the hope that this would produce more rapidly a diverse spread of different chromosomes.

Tests were then run on the test suite `difffile-large`, which requests a series of different HTML and image files, all large. Each test was run for 18 generations, consisting of a population of 35 candidates each, and repeated three times. The results were the averaged across each of the three runs. The average result per generation is visible in Figs 1 & 2.

There were a number of anomalous results that initially made discerning a pattern difficult (e.g. candidate 14_4 from the first run returned 410 ms). Removing any results $\geq$35 ms (only 14 out of a total of 1890 results) produced Fig. 2, where the trending towards faster and faster response times is more obvious. From a wider range of results for the initial few generations (21–30 ms), the components in later generations tended towards consistent level of around 21 ms response time.

It wouldn't have been surprising to see that these higher-performing chromosomes shared similar characteristics, such as all calling `nthMostFrequentlyUsed()`. Indeed, this appears to have been somewhat the case, although not exclusively so—the chromosomes '`11/random()`' and '`nthMostRecentlyUsed(0)*nthMostFrequentlyUsed(0)`' each achieve 21 ms, as did a range of other chromosomes. However, the most common characteristic of all of the generation 18 chromosomes seemed to be the presence of one or both of the `nthMost*Used()` functions—29 out of the 38 chromosomes featured one of these functions, and 8 featured both.[13]

The test suite `samefile-images`—which requests the same image file repeatedly—was then run with the same conditions. The results are visible in figs 3 and 4.

There is no visible improvement in either of these charts. Barring two anomalous results, the response times only varied between 18 and 19 ms, suggesting that no amount of

13. To view all of the gen. 18 chromosomes, see Appendix C.6

mutation made a difference on performance. This was expected, however—after enough runs, every index in the cache will point to identical data and the formula for updating the cache will be meaningless.

7.3 Evaluation

The research aim specified for EmerGen(e)tic was 'to produce a framework for enabling rapid development of genetic improvement tests'. This has been done—EmerGen(e)tic is simple to extend with new modules, being written in the more widely-known Python language (rather than the admittedly more niche Dana). Given a blank copy of EmerGen(e)tic and this report, it is confidently assumed that a researcher could begin deploying their own tests in minimal time. To support this claim, the `helloworld` and `cachingpolicy` modules were developed to demonstrate the framework in action.

The `cachepolicy` module (initially) produced results that appear to support the assertion made in §1—i.e. that genetic algorithms can be usefully applied to the creation of emergent system components. As was expected too, this is not the case for all contexts (e.g. `samefile-images`), which demonstrates that in some instances, the added overhead of running genetic algorithm generations is not rewarded with any tangible improvements.

However, perhaps a scope that would have produced more variance than just cache updating behaviour would have been advisable to choose at the start of the project in order to better prove or disprove this hypothesis. As it stands, a proof, albeit a slightly weak proof, is nonetheless provided.

An additional aim was set, to 'repeat the tests multiple times with a variety of conditions and see what, if any, trends emerge.' However, between the tests featured in §7.2 and when the condition-variance tests were due to be run, access to the research computer was temporarily suspended. Upon returning to it a week later, test runs (even those with identical conditions to the ones previously performed) no longer produced meaningful results—response times barely varied for all tests run, regardless of number of generations. To make sure that this was not a result of the machine's processing speed increasing and minor changes in response time being undetectable, all of the test suites were tripled in length in the hope that minor changes would become more pronounced. Unfortunately this had no effect, and at this late stage in the project no alternative was left but to cease testing of the `cachepolicy` module.

However, the approaches that would have been taken will be outlines in the 'Future Improvements' section at the end of this report.

8 Conclusion

In §1, two research questions were established. In this section, the answers provided will be considered. Then, suggestions for further research will be outlined.

Can genetic algorithms can be productively used to produce optimum components for an emergent system?

This has been shown to be the case. The response times for delivery of certain types of files have been demonstrated steadily decreasing per generation following application of genetic algorithms. However, it has also been shown that there are some instances where genetic algorithms are of little use (e.g. when the same file is requested from the web server repeatedly). In these cases, the overhead of running the genetic algorithm (and each generation takes a non-trivial amount of time to produce, compile and test) is not rewarded.

What are the optimal conditions for doing so (i.e. mutation probabilities, generation sizes, etc.)?

As explained in §7.3, issues that emerged late in the project prohibited exploration of this area.

8.1 Future Development

8.1.1 Completing the original goal of finding the optimal conditions for the genetic algorithm

As previously mentioned, issues prevented this avenue of research from being explored. If these issues were resolved, it would be useful to revisit this second research question. The intended plan was to set the probabilities of the four specific mutation operations to 50 %, run a test suite with mutation and crossover probabilities set at each 10-step interval (i.e. 0 %, 10 %, . . . , 100 %) and compare results to determine the optimum probabilities of both. This could then be re-run with the probabilities set at opposing ends (i.e. 0% and 100 %) and converging towards and then past each other—this would hypothetically display whether similar or distinct probabilities for the two operations (if either) are more desirable.
After these tests have hopefully shown a broad region of optimal probabilities, more fine-grained testing within that region could then take place. This could involve testing the original two probabilities with a range of values across a smaller step size, or testing different combinations of probabilities for the four specific mutation operations. For example, all but one could be set to 0 % and this used to determine how well they work in isolation.

8.1.2 Designing more modules

For a future project or projects, one could develop and run different tests using the EmerGen(e)tic framework provided here. For example, a module that applied genetic algorithms to the file compression options including in Filho & Porter's original web server. Alternatively, such a module could be combined with another (such as `cachingpolicy`) in order to test a broader scope of mutation than just individual functionality and demonstrate the potential interlinked effects of this.

8.1.3 A more 'Danatically'-designed EmerGen(e)tic

When running an EmerGen(e)tic module in which the genetic algorithm is improperly implemented, and thus can produce uncompilable components, the compilation attempt will fail for that component and the entire EmerGen(e)tic test run/suite fail upon trying to

load the non-existent component object file that has thus not been produced. Dana lacks exception-handling functionality, as a component's failure is intended to be signalled to the calling component via a `return`. By re-designing the system in a more idiomatically Dana way and calling the tests as a separate component rather than in the `runCandidate()` method of `emergenetic.dn`, failure of an individual test could be ignored and the test runs continued.
Another benefit of this would be the complete separation of the functionality currently within `runCandidate()`, which module developers must edit in order to implement their modules, and the remainder of the `emergenetic.dn` file, which they should no have cause to touch. As it stands, the `cachingpolicy` module contains a copy of `emergenetic.dn` with the `runCandidates()` implementation within it, which overwrites the existing `emergenetic.dn` within the base distribution. This raises issues—if a module is written for a specific version of `emergenetic.dn`, upgrading it would result in extra, unnecessary workload for the developer. However, if the `runCandidate()` functionality could thus be moved into a separate `runCandidate.dn` file, then issues with EmerGen(e)tic versions would be resolved. Also, this would allow EmerGen(e)tic to be distributed as a black box API, without having to distribute source files (although, as the software is released under the GNU GPLv3, this last point is less relevant), as well as allowing a user to install multiple modules simultaneously.

8.1.4 Issues within the present `cachingpolicy` module

Besides the issues that prohibited testing for optimum conditions, there were a couple other outstanding issues within the `cachingpolicy` modules that could be fixed in the future.
Foremost of these is that test runs will sometimes, with what appears to be no predictability, stop outputting just after finishing a generation but continue to run, draining system resources until the process is killed. `python` can be seen to still be running which, along with the stage in the process where the issue occurs and the lack of an output error message, suggests that the fault may lie in the `cachingpolicy` module. The nature of the issue would further suggest an endless loop occurring somewhere—thus, suspicions fall upon the `while` loop within the `crossover()` method, or the recursive elements of the `crossover()` and `mutation()` methods.

8.1.5 Putting human developers further out of work (through meta-learning)

Finally, and potentially the most interesting avenue for further research of the system based on the problem it set out to fix (as outlined in §1): meta-learning. [20] In this case, the system would have an ability not just to rank candidates by a predefined metric (in `cachingpolicy`'s case the response time in ms), but also to change its metric using the same principles of genetic improvement. Despite removing the human developer from much of the software development via the implementation of genetic algorithms, one is still required to write those algorithms and, most egregiously, outline the metric(s) by which performance is measured. If the system could be left to improve its own algorithm and, if a candidate was found, amend how it judges success of candidates, the goal of the programmerless system will have been even further achieved.

REFERENCES

[1] G. F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2012.

[2] E. Redwine and J. L. Holliday, "Reliability of distributed systems." [Online]. Available: http://www.cse.scu.edu/ jholliday/REL-EAR.htm

[3] R. Rodrigues Filho and B. F. Porter, "Experiments with a machine-centric approach to realise distributed emergent software systems," 2016.

[4] G. Hornby, A. Globus, D. Linden, and J. Lohn, "Automated antenna design with evolutionary algorithms," in *Space 2006*, 2015, p. 7242.

[5] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 3–13.

[6] B. Goldsworthy, "'thanks, ants. thants.': Applying genetic algorithms to simulated ants to produce higher-fitness generations," unpublished.

[7] M. Dawson, D. N. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, no. 6, 2010.

[8] S. McConnell, *Code complete*. Pearson Education, 2004.

[9] B. Goldsworthy, "Critical infrastructures protection: the responsibility of government or of companies?" unpublished.

[10] E. S. Raymond, *The art of Unix programming*. Addison-Wesley Professional, 2003.

[11] D. Muder, *The Unreasonable Influence of Geometry*, 2000.

[12] L. Lamport, "Distribution," May 1987. [Online]. Available: https://www.microsoft.com/en-us/research/publication/distribution/

[13] S. Crocker, "New host-host protocol," Internet Requests for Comments, RFC Editor, RFC 33, February 1970, http://www.rfc-editor.org/rfc/rfc33.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc33.txt

[14] B. F. Porter, "Runtime modularity in complex structures: A component model for fine grained runtime adaptation," in *Proceedings of the 17th international ACM Sigsoft symposium on Component-based software engineering*. ACM, 2014, pp. 29–34.

[15] B. F. Porter and R. Rodrigues Filho, "Losing control: The case for emergent software systems using autonomous assembly, perception, and learning," in *Self-Adaptive and Self-Organizing Systems (SASO), 2016 IEEE 10th International Conference on*. IEEE, 2016, pp. 40–49.

[16] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "Re^{x}: A development platform and online learning approach for runtime emergent software systems," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. USENIX*, 2016.

[17] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: A framework for engineering self-tuning self-adaptive software systems," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 7–16. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882296

[18] R. W. Floyd, "The paradigms of programming," *Communications of the ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[19] J. V. Hansen, P. B. Lowry, R. D. Meservy, and D. M. McDonald, "Genetic programming for prevention of cyberterrorism through dynamic and evolving intrusion detection," *Decision Support Systems*, vol. 43, no. 4, pp. 1362–1374, 2007.

[20] J. Schmidhuber, "Evolutionary principles in self-referential learning," *On learning how to learn: The meta-meta-... hook.) Diploma thesis, Institut f. Informatik, Tech. Univ. Munich*, 1987.

APPENDIX A

EMERGENETIC.DN

```
/*
 *                  EmerGen(e)tic 1.0
 *          Copyright  2017 Ben Goldsworthy (rumperuu)
 *
 * EmerGen(e)tic is a framework for researching the use of genetic
 * algorithms in emergent systems.
 *
 * This file is part of EmerGen(e)tic.
 *
 * EmerGen(e)tic is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * EmerGen(e)tic is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with EmerGen(e)tic. If not, see <http://www.gnu.org/licenses/>.
 */

/**
 ** This program runs the tests that are passed to it as command-line
 ** arguments. It contains methods that can be easily overwritten for
 ** creating new tests.
 **/

/**
 **  @author Ben Goldsworthy (rumperuu) <me@bengoldsworthy.uk>
 **  @version 1.0
 **/
component provides App requires os.Run run,
                            io.Output out,
                            composition.Loader unloader,
                            composition.RecursiveLoader loader,
                            time.Calendar calendar,
                            time.DateUtil dateUtil,
                            data.StringUtil stringUtil,
                            data.IntUtil intUtil,
                            io.File,
                            io.TextFile {

  // Required arguments.
  int argMod = 0
  int argScript = argMod + 1
  int argGens = argScript + 1
  int argCans = argGens + 1
  int minArgs = argCans + 1

  // Optional arguments.
  int argVerb = minArgs
  int maxArgs = minArgs + 1

  /**
   **  Main method. Runs the generations and mutation. This method
```

```
 **  should not need to be modified.
 **
 **  @param params The arguments the program is run with (e.g.
 **  the number of generations to run, and the population size per
 **  generation, etc.)
 **  @return The program exit code.
 **/
int App:main(AppParam params[]) {
  // Checks that there are enough passed array arguments, and that
  // they are valid.
  if ((params.arrayLength >= minArgs) && (params.arrayLength <= maxArgs)) {
    if (stringUtil.isNumeric(params[argGens].string) &&
        stringUtil.isNumeric(params[argCans].string)) {
      // Assigns all of the command-line arguments to variables.
      int generations = intUtil.intFromString(params[argGens].string)
      int candidates = intUtil.intFromString(params[argCans].string)
      char module[] = params[argMod].string
      char script[] = params[argScript].string
      int verbose = 0
      if (params.arrayLength == maxArgs) {
        if (params[argVerb].string == "verbose") {
          verbose = 1
        }
      }

      // Performs any initial setup.
      RunStatus setupResult = run.execute("./project/$(module)/setup.sh
          $(intUtil.intToString(generations)) $(script)")
      // Iterates through the generations, creating them and then
      // testing each of their candidates.
      for (int i = 0; i < generations; i++) {
        // Runs the per-generation mutator.
        RunStatus result = run.execute("python3
            ./project/$(module)/mutator.py $(intUtil.intToString(i))
            $(intUtil.intToString(candidates))
            $(intUtil.intToString(verbose)) $(script)")
        // Runs the generation.
        runGeneration(i, candidates, script)
      }

      return 0
    // If the 'generations' and 'candidates' arguments are non-numeric...
    } else {
      out.println("Invalid runtime arguments: Arguments <generation> and
          <candidates> must be integers.")
      return 1
    }
  // If an invalid number of arguments are passed to the program...
  } else {
    out.println("Invalid runtime arguments: Program should be run
        as:\n\t`dana emergenetic <module> <script> <generations>
        <candidates>`\n\nProgram may also be run with optional `verbose` flag
        as final argument:\n\tdana emergenetic <module> <script>
        <generations> <candidates> verbose`")
    return 1
  }
}

/**
 **  Runs a generation of candidates. This method should not need to
```

```
 **  be modified.
 **
 **  @param generation The generation of the candidate.
 **  @param candidates The total number of candidates.
 **  @param script The name of the script file being used.
 **/
void runGeneration(int generation, int candidates, char script[]) {
  File results

  out.println("")
  // Runs through each candidate.
  for (int j = 0; j < candidates; j++) runCandidate(generation, j, script)

  // For the end of the generation's candidates, appends a newline to
  // the CSV file.
  results = new File("./results/$(script)/results.csv",
      File.FILE_ACCESS_WRITE)
  results.setPos(results.getSize())
  results.write("\r\n")
  results.close()
}

/**
 **  Runs a candidate through a test. Modify this method for changing
 **  what is being tested.
 **
 **  @param generation The generation of the candidate.
 **  @param candidate The number of the candidate.
 **  @param script The name of the script file being used.
 **/
void runCandidate(int generation, int candidate, char script[]) {
  // This method intentionally left blank.
}

/**
 **  Prints the results of a test run. This method should not need to
 **  be modified.
 **
 **  @param generation The generation of the candidate.
 **  @param candidate The number of the candidate.
 **  @param metric The value of the metric by which candidates are
 **    being rated (as a string).
 **  @param unit The unit of the value (e.g. ms, cm, etc.).
 **  @param script The name of the script file being used.
 **/
void printResults(int generation, int candidate, char metric[], char unit[],
    char script[]) {
  File results

  out.println("Result for configuration
      $(intUtil.intToString(generation))x$(intUtil.intToString(candidate)):
      $(metric) $(unit)")

  results = new
      File("./results/$(script)/results$(intUtil.intToString(generation)).txt",
      File.FILE_ACCESS_WRITE)
  results.setPos(results.getSize())
  results.write("$(intUtil.intToString(generation))_$(intUtil.intToString(candidate)):$
  results.close()
```

```
    results = new File("./results/$(script)/results.csv",
        File.FILE_ACCESS_WRITE)
    results.setPos(results.getSize())
    results.write("$(metric),")
    results.close()
  }
}
```

APPENDIX B

HELLOWORLD/MUTATOR.PY

```
#!/usr/bin/python

'''
                    HelloWorld 1.0
               Copyright (c) 2017 Ben Goldsworthy (rumps)

   HelloWorld is an EmerGen(e)tic module for testing the function of the
  EmerGen(e)tic framework.

   This file is part of the HelloWorld EmerGen(e)tic module.

  HelloWorld is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  HelloWorld is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with HelloWorld. If not, see <http://www.gnu.org/licenses/>.
'''

import fileinput
import random
import shutil
import string
import sys

script = ""

def mutate(chromosome):
  """Randomly replaces letters in the string."""
  mutation = [random.choice(string.ascii_lowercase) if random.randrange(100) <
      20 else x for x in "Hello world"]
  return ''.join(mutation)

def createInitialPop(candidates):
  """Creates the initial population for generation 0."""
  for i in range(candidates):
    cID = "0_"+str(i)
    shutil.copyfile("./helloworld/helloworld.txt",
        "./helloworld/"+script+"/0/helloworld"+cID+".txt")

def readChromosomeFromFile(cID):
  """Reads the chromosome from a given `CacheHandler*.dn` file."""
  with
      open("./helloworld/"+script+"/"+str(cID.partition("_")[0])+"/helloworld"+cID+".txt")
      as inFile:
    return inFile.readline()

def writeChromosomeToFile(cID, chromosome):
  """Writes a given chromosome to a `CacheHandler*.dn` file."""
  with
      open("./helloworld/"+script+"/"+str(cID.partition("_")[0])+"/helloworld"+cID+".txt",
```

```
      "w") as outFile:
      outFile.write(chromosome)


def main(args):
  random.seed()

  tabLevel = ""
  chromosome = []
  generation = int(args[0])
  candidates = int(args[1])
  chromosomes = []
  newChromosomes = []
  global script
  script = args[3]

  if generation == 0:
    createInitialPop(candidates)
  else:
    # Gets all of the previous generation's chromosomes.
    for i in range(candidates):
      oldCID = str(generation - 1) + "_" + str(i)
      chromosomes.append((readChromosomeFromFile(oldCID), oldCID))

    # Applies a 30% chance of changing the string.
    for currCandidate, chromosome in enumerate(chromosomes):
      newCID = str(generation) + "_" + str(currCandidate)

      if random.randrange(100) < 30:
        newChromosome = [mutate(chromosome[0]), newCID]
      else:
        newChromosome = [chromosome[0], newCID]
      print(newChromosome)
      newChromosomes.append(newChromosome)

    # Writes this new population to `.txt` files.
    for currCandidate, chromosome in enumerate(newChromosomes):
      newCID = str(generation) + "_" + str(currCandidate)
      writeChromosomeToFile(newCID, chromosome[0])
  return 0

if __name__ == '__main__':
  sys.exit(main(sys.argv[1:]))
```

APPENDIX C

CACHINGPOLICY

C.1 CacheHandlerBase.dn

```
/*
 *                 CachingPolicy 0.9
 *          Copyright  2017 Ben Goldsworthy (rumperuu)
 *
 * CachingPolicy is an EmerGen(e)tic module for testing the use of genetic
 * algorithms as applied to the cache updating behaviour of a web server.
 *
 * This file is part of the CachingPolicy EmerGen(e)tic module.
 *
 * CachingPolicy is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * CachingPolicy is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with CachingPolicy. If not, see <http://www.gnu.org/licenses/>.
 */

/**
 ** This program serves as the base for each mutated CacheHandler,
 ** providing the functionality that is common to all of them.
 **/

/**
 **  @author Ben Goldsworthy (rumperuu) <me@bengoldsworthy.uk>
 **  @version 0.9
 **/
component provides cache.CacheHandler
        requires io.Output out,
                 data.IntUtil iu,
                 util.RandomInt ir,
                 time.Calendar ic,
                 time.DateUtil du,
                 io.File {

  static CachedResponse cache[]
  static int arraySize
  static Mutex cacheLock

  /**
   **  Returns a requested item from the cache.
   **
   **  @param resource The request item filepath
   **  @return The item from the cache
   **/
  CachedResponse CacheHandler:getCachedResponse(char filePath[]) {
    // Converts the filepath to an array of bytes for the file.
    File fd = new File(filePath, File.FILE_ACCESS_READ)
    char resource[] = fd.read(fd.getSize())
    fd.close()
```

```
  // Returns the item if it's in the cache.
  mutex(cacheLock) {
    for (int i = 0; i < arraySize; i++) {
      if (cache[i].resource == resource) {
        cache[i].hits++
        cache[i].lastUsed = ic.getTime()
        return cache[i]
      }
    }
  }
  return null
}

/**
 **  Updates the cache if the requested item is not already present.
 **  If the cache is full, the method of determining the item to be
 **  replaced is subject to genetic mutation.
 **
 **  @param ncr The new item to cache
 **/
void CacheHandler:updateCache(CachedResponse ncr) {
  mutex(cacheLock) {
     // If the item is already in the cache, do nothing.
    for (int i = 0; i < arraySize; i++) {
      if (cache[i].resource == ncr.resource) {
        cache[i].response = ncr.response
        cache[i].resourceAge = ncr.resourceAge

        return
      }
    }

    // If the item is not, it must be added.
    // As `ncr` was created on another component, its details must
    // be copied to avoid a read-only exception.
    CachedResponse newEntry = new CachedResponse()
    newEntry.command = ncr.command
    newEntry.resource = ncr.resource
    newEntry.resourceAge = ncr.resourceAge
    newEntry.mimeType = ncr.mimeType
    newEntry.response = new byte[](ncr.response)
    newEntry.contentSize = ncr.contentSize

    // Initialises the cache if there isn't currently one.
    if (cache == null) {
      cache = new CachedResponse[CacheHandler.MAX_SIZE]()
    }

    // If the cache is full, determines which item to replace.
    if (arraySize == CacheHandler.MAX_SIZE) {
      int index

      // BEGIN
      index = 0
      // END

      cache[index % arraySize] = newEntry
      //out.println("$(debugMSG) replacing: $(iu.intToString(index))")
    // Otherwise, appends the item to the end of the cache.
```

```
    } else {
      cache[arraySize] = newEntry
      arraySize++
    }
  }
}

 /**
  **   Clears the cache completely.
  **/
void CacheHandler:clearCache() {
  mutex(cacheLock) {
    cache = null
  }
}

/*****************************************************************************
 * What follows are utility functions that a given chromosome of the
 * `updateCache()` method may or may not call upon.
 *****************************************************************************/

// This method returns the index of the nth most frequently-used item in
// the cache.
int nthMostFrequentlyUsed(int n) {
  int hits[]
  for (int i = 0; i < arraySize; i++) hits[i] = 0

  int i
  int j
  mutex(cacheLock) {
    for (i = 0; i < arraySize; i++) {
      for (j = 0; j < arraySize; j++) {
        if (cache[i].hits == hits[j]) break
        else if (hits[j] == 0) {
          hits[j] = cache[i].hits
          break
        }
      }
    }
  }

  int k = 0
  int nthHit = 0
  for (i = 0; i < j; i++) {
    if (hits[i] > nthHit) {
      nthHit = hits[i]
      k++
    }
    if (k == n) break
  }

  int itemIndices[]

  mutex(cacheLock) {
    j = 0
    for (i = 0; i < arraySize; i++) {
      if (cache[i].hits == nthHit) itemIndices[j++] = i
    }
  }
```

```
  return resolve(itemIndices, "r")
}

// This method returns the index of the nth most recently-used item in
// the cache.
int nthMostRecentlyUsed(int n) {
  DateTime useTimes[]
  DateTime testTime = ic.getTime()
  for (int i = 0; i < arraySize; i++) useTimes[i] = testTime

  int i
  int j
  mutex(cacheLock) {
    for (i = 0; i < arraySize; i++) {
      for (j = 0; j < arraySize; j++) {
        if (du.equal(cache[i].lastUsed, useTimes[j])) break
        else if (du.equal(useTimes[j], testTime)) {
          useTimes[j] = cache[i].lastUsed
          break
        }
      }
    }
  }

  int k = 0
  DateTime nthUseTime = ic.getTime()
  for (i = 0; i < j; i++) {
    if (du.after(useTimes[i], nthUseTime)) {
      nthUseTime = useTimes[i]
      k++
    }
    if (k == n) break
  }

  int itemIndices[]

  mutex(cacheLock) {
    j = 0
    for (i = 0; i < arraySize; i++) {
      if (du.equal(cache[i].lastUsed, nthUseTime)) itemIndices[j++] = i
    }
  }

  return resolve(itemIndices, "r")
}

// This method returns a random index.
int random() {
  DateTime dt = ic.getTime()
  int msec = dt.millisecond
  ir.setSeed(msec)
  return ir.get(CacheHandler.MAX_SIZE)
}

// This method resolves a multiple return in one of the methods above,
// as per the flag sent along with the list of indices.
// 'n' returns the newest, 'o' the oldest and 'r' a random index.
int resolve(int index[], char flag) {
  DateTime dt = null
```

```
    if (flag == "n") {
      int newestItem = 0

      int i = index.arrayLength-1
      for (int j = 0; j < index.arrayLength; j++) {
        if (dt == null) {
          dt = cache[index[i]].timeAdded
          newestItem = index[i]
        } else {
          if (du.before(dt, cache[index[i]].timeAdded)) {
            dt = cache[index[i]].timeAdded
            newestItem = index[i]
          }
        }
        i--
      }

      return newestItem
    } else if (flag == "o") {
      int oldestItem = 0

      for (int i = 0; i < index.arrayLength; i++) {
        if (dt == null) {
          dt = cache[index[i]].timeAdded
          oldestItem = index[i]
        } else {
          if (du.before(cache[index[i]].timeAdded, dt)) {
            dt = cache[index[i]].timeAdded
            oldestItem = index[i]
          }
        }
      }
      return oldestItem
    } else if (flag == "r") {
      dt = ic.getTime()
      int msec = dt.millisecond
      ir.setSeed(msec)
      return ir.get(index.arrayLength)
    } else {
      return 0
    }
  }
}
```

C.2 mutator.py

```
#!/usr/bin/python

'''
                    CachingPolicy 0.9
               Copyright (c) 2017 Ben Goldsworthy (rumps)

   CachingPolicy is an EmerGen(e)tic module for testing the use of genetic
  algorithms as applied to the cache updating behaviour of a web server.

   This file is part of the CachingPolicy EmerGen(e)tic module.

  CachingPolicy is free software: you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.

  CachingPolicy is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with CachingPolicy. If not, see <http://www.gnu.org/licenses/>.
'''

import fileinput
import math
import random
import shutil
import sys
import os

# This script generates the initial population of candidate programs,
# and performs mutation/crossover for each subsequent generation.

# Mutation probability variables, to be read in from the config file upon the
# first run of the mutation/crossover methods.
# 0: Mutation chance
# 1: Crossover chance
# 2: Binary Operator Mutation chance
# 3: Operand Mutation chance
# 4: nthMost* Mutation chance
# 5: Create Subtree Mutation chance

probabilities = [None for x in range(6)]

with open("./project/cachingpolicy/config.conf") as inFile:
  for i, line in enumerate(inFile):
    if i == 6:
      intRange = int(line.partition(":")[2])

binaryOperators = ["*", "+", "-", "/"]
operands = [str(n) for n in range(intRange)]
for n in range(intRange):
  operands.append("nthMostRecentlyUsed(0)")
  operands.append("nthMostFrequentlyUsed(0)")
operands.append("random()")
verbose = False
```

```
script = ""

def getSubLists(lst):
  """Recursively gets every sub-tree of a given expression."""
  subLists = []
  for token in lst:
    if isinstance(token, list):
      subLists.append(token)
      if getSubLists(token) is not None:
        subLists.append(getSubLists(token)[0])
  if len(subLists) == 0: return None
  elif len(subLists) == 1: return subLists[0]
  else: return subLists

def crossover(parentA, parentB):
  """Crosses over a random subtree from chromosome A to a random point on
  chromosome B."""
  if verbose: print("\tCrossover: Parent A (", compile(parentA), ") x Parent B
     (", compile(parentB), ")")

  subTrees = []
  for i, token in enumerate(parentA):
    if isinstance(token, list):
      subTrees.append(token)
      subLists = getSubLists(token)
      if subLists is not None:
        subTrees.append(subLists)

  changePoint = []

  temp = 0
  while 1:
    # DEBUG - trying to avoid the endless loop error, if this is the cause.
    temp += 1
    if temp > 1000: break
    #/DEBUG
    changePoint = random.choice(parentB)
    if not isinstance(changePoint, list):
      if str(changePoint).isnumeric():
        break
  if len(subTrees) > 0:
    newSubTree = random.choice(subTrees)
    if verbose: print("\t\tResult: "+compile(newSubTree)+" replaced operand
       "+changePoint+" ("+compile([newSubTree if x==changePoint else x for x
       in parentB])+")")
    return [newSubTree if x==changePoint else x for x in parentB]
  else: return [parentB]

def mutate(chromosome, initialPop = None):
  """Recursively applies, to each token, a random chance of mutating
  a given operand, operator, or replacing the former with an entirely
  new expression."""
  if probabilities[2] is None:
    with open("./project/cachingpolicy/config.conf") as inFile:
      for i, line in enumerate(inFile):
        if i < 6 and i > 1:
          probabilities[i] = int(line.partition(":")[2])

  operandMutChance = 100 if initialPop else probabilities[3]
```

```
  for i, token in enumerate(chromosome):
    if verbose: print("\tMutating: "+compile(chromosome))
    token = str(token)

    if len(chromosome) < 3:
      if random.randrange(100) < probabilities[5]:# and "nthMost" not in token:
        if len(chromosome) == 1:
          chromosome.append(random.choice(binaryOperators))
          chromosome.append(random.randrange(intRange))
        elif random.randrange(100) < 50:
          chromosome.remove(chromosome[0])
        else:
          chromosome[1] = [random.choice(operands),
              random.choice(binaryOperators), random.choice(operands)]

    else:
      if isinstance(token, list):
        chromosome[i] = mutate(token)

      if token in binaryOperators and random.randrange(100) < probabilities[2]:
        chromosome[i] = random.choice(binaryOperators)
      elif token.isnumeric() and random.randrange(100) < operandMutChance:
        chromosome[i] = random.choice(operands)
    '''
    elif "nthMost" in token and random.randrange(100) < probabilities[4]:
      if random.randrange(100) < 50:
        chromosome[i] = "nthMostRecentlyUsed" if token ==
            "nthMostFrequentlyUsed" else "nthMostFrequentlyUsed"
      else:
        chromosome.remove(token)
    '''
    if verbose: print("\t\tResult: "+compile(chromosome))

  return chromosome

def parse(chromosome):
  """Given a `str` representation of a chromosome, recursively parses
  it into a `List`."""
  i = 0
  counter = 0
  substring = ""
  result = []
  while i < len(chromosome):
    if chromosome[i] is "r":
      i += len("random()")
      result.append("random()")
    elif chromosome[i] is "n":
      function = ""
      if chromosome[i+7] is "F":
        function = "nthMostFrequentlyUsed"
      elif chromosome[i+7] is "R":
        function = "nthMostRecentlyUsed"
      i += len(function)-1
      result.append(function)
    elif chromosome[i] is "(":
      substring = ""
      for j in chromosome[i:]:
        if j is "(":
          counter += 1
        elif j is ")":
```

```
            counter -= 1
          substring += j
          if counter == 0:
            result.append(parse(substring[1:(len(substring)-1)]))
            i += len(substring) - 1
            break
      elif chromosome[i].isnumeric():
        num = ""
        while chromosome[i].isnumeric():
          num += chromosome[i]
          i += 1
          if i == len(chromosome):
            break
        if len(num) > 0:
          i -= 1
        result.append(num)
      elif chromosome[i] in binaryOperators:
        result.append(chromosome[i])
      i += 1
    return result

def compile(chromosome):
    """Given a `List` representation of a chromosome, recursively compiles
    it into a `str`."""
    string = ""
    for i in chromosome:
      if isinstance(i, list):
        string += "("
        string += compile(i)
        string += ")"
      else:
        string += str(i)
    return string

def createInitialPop(candidates):
    """Creates the initial population for generation 0."""
    for i in range(candidates):
      cID = "0_"+str(i)
      shutil.copyfile("./cache/CacheHandlerBase.dn",
          "./cache/"+script+"/0/CacheHandler"+cID+".dn")
      writeChromosomeToFile(cID, parse(readChromosomeFromFile(cID)))
      os.system("dnc ./cache/"+script+"/0/CacheHandler"+cID+".dn")

def readChromosomeFromFile(cID):
    """Reads the chromosome from a given `CacheHandler*.dn` file."""
    with
        open("./cache/"+script+"/"+str(cID.partition("_")[0])+"/CacheHandler"+cID+".dn")
        as inFile:
      for line in inFile:
        if "// BEGIN" in line:
          return next(inFile).partition(" = ")[2].rstrip()

def writeChromosomeToFile(cID, chromosome):
    """Writes a given chromosome to a `CacheHandler*.dn` file."""
    shutil.copyfile("./cache/CacheHandlerBase.dn",
        "./cache/"+script+"/"+str(cID.partition("_")[0])+"/CacheHandler"+cID+".dn.temp")
    with
        open("./cache/"+script+"/"+str(cID.partition("_")[0])+"/CacheHandler"+cID+".dn.temp"
        as inFile:
```

```
      with
          open("./cache/"+script+"/"+str(cID.partition("_")[0])+"/CacheHandler"+cID+".dn",
          "w") as outFile:
        for line in inFile:
          outFile.write(line)
          if "// BEGIN" in line:
            outFile.write("\t\t\t\tindex = "+compile(chromosome[0])+"\n")
            next(inFile)
    os.remove("./cache/"+script+"/"+str(cID.partition("_")[0])+"/CacheHandler"+cID+".dn.temp

def hasSubTrees(chromosome):
    for x in chromosome[0]:
      if isinstance(x, list): return True
    else: return False

def main(args):
    random.seed()

    tabLevel = ""
    chromosome = []
    generation = int(args[0])
    candidates = int(args[1])
    chromosomes = []
    newChromosomes = []
    global script
    script = args[3]
    global verbose
    verbose = True if int(args[2]) == 1 else False

    if generation == 0:
      createInitialPop(candidates)
    else:
      # Gets all of the previous generation's chromosomes.
      for i in range(candidates):
        oldCID = str(generation - 1) + "_" + str(i)
        chromosomes.append((parse(readChromosomeFromFile(oldCID)), oldCID))

      # Reads in the results of the previous generation's chromosomes.
      rankSelect = []
      with open("./results/"+script+"/results"+str(generation-1)+".txt") as
          inFile:
        for line in inFile:
          rankSelect.append([line.partition(":")[0],
              line.partition(":")[2].partition("ms")[0]])
      # Sorts them by response time
      rankSelect.sort(key=lambda x: int(x[1]))
      # Truncates the bottom 90% of the candidates
      del rankSelect[int(math.ceil(candidates/10)):]

      remainingIDs = [str(n) for n in range(candidates)]

      # Retains the top 10% best-peforming chromosomes for the next
      # generation.
      if verbose: print("\nApplying rank selection to top 10% from prev.
          generation...")
      for currCandidate, oldCID in enumerate(rankSelect):
        newCID = str(generation) + "_" + str(oldCID[0].partition("_")[2])
        for chromosome in chromosomes:
          if chromosome[1] == oldCID[0]:
            if verbose: print("\tResult: "+str(oldCID[0])+" copied to "+newCID)
```

```
            newChromosome = [chromosome[0], newCID]
            newChromosomes.append(newChromosome)
            #chromosomes.remove(chromosome)
            remainingIDs.remove(newCID.partition("_")[2])
            break
      if verbose: print("\n")

      if probabilities[0] is None:
        with open("./project/cachingpolicy/config.conf") as inFile:
          for i, line in enumerate(inFile):
            if i > 1: break
            probabilities[i] = int(line.partition(":")[2])

      # For the remaining chromosomes, creates the next generation's
      # population, whilst having a chance to affect mutation or
      # crossover operations on each.
      for currCandidate, chromosome in enumerate(chromosomes):
        if str(currCandidate) in remainingIDs:
          newCID = str(generation) + "_" + str(currCandidate)

          newChromosome = [chromosome[0], newCID]
          if random.randrange(100) < probabilities[0]:
            haveSubTrees = [x for x in chromosomes if hasSubTrees(x)]
            if len(haveSubTrees) > 1:
              parentB = random.choice(haveSubTrees)[0]
              if hasSubTrees(chromosome[0]):
                if verbose: print("Crossing over into "+newCID)
                newChromosome = [crossover(chromosome[0], parentB, verbose),
                    newCID]
                if verbose: print("Crossover finish.")

          if random.randrange(100) < probabilities[1]:
            if verbose: print("Mutating "+str(generation-1) + "_" +
                str(currCandidate)+"->"+newCID+":")
            newChromosome = [mutate(chromosome[0], True), newCID]
            if verbose: print(str(generation-1) + "_" +
                str(currCandidate)+"->"+newCID+" mutation finish.\n")


          newChromosomes.append(newChromosome)

      # Writes this new population to `.dn` files, and then runs the
      # Dana compiler on the.
      for currCandidate, chromosome in enumerate(newChromosomes):
        newCID = str(generation) + "_" + str(currCandidate)
        writeChromosomeToFile(newCID, chromosome)
        os.system("dnc
            ./cache/"+script+"/"+str(newCID.partition("_")[0])+"/CacheHandler"+newCID+".dn")
    return 0

if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))
```

C.3 `emergenetic.dn`'s `runCandidates()` method

```
    /**
     **  Runs a candidate through a test. Modify this method for changing
     **  what is being tested.
     **
     **  @param generation The generation of the candidate.
```

```
   **  @param candidate The number of the candidate.
   **  @param script The name of the script file being used.
   **/
  void runCandidate(int generation, int candidate, char script[]) {
    LoadedComponents lc
    IDC com
    CacheHandler c
    CachedResponse cr
    DateTime start
    TextFile scriptFile

    // Loads the given candidate file.
    lc =
        loader.load("./cache/$(script)/$(intUtil.intToString(generation))/CacheHandler$(in
    com = lc.mainComponent
    c = new CacheHandler() from com :< CacheHandler

    start = calendar.getTime()

    // Runs all of the files within the given script file as requests.
    scriptFile = new TextFile("./scripts/$(script).script",
        File.FILE_ACCESS_READ)
    char line[] = scriptFile.readLine()
    while (line != "") {
      cr = c.getCachedResponse(line)
      line = scriptFile.readLine()
    }
    scriptFile.close()

    // Records the response time to complete all requests and records
    // the results.
    int responseTime = dateUtil.toMilliseconds(dateUtil.diff(start,
        calendar.getTime()))
    printResults(generation, candidate, intUtil.intToString(responseTime),
        "ms", script)

    // Unloads the component(s).
    c = null
    for (int k = 0; k < lc.loadedComponents.arrayLength; k++) {
      unloader.unload(lc.loadedComponents[k].class)
    }
  }
```

C.4 `setup.sh`

```
#!/bin/bash
# Performs pre-run setup before a test run.

GENS=$1
SCRIPT=$2

cp ./cachebackup/CacheHandlerBase.dn ./cache/CacheHandlerBase.dn
mkdir ./cache/$2
mkdir ./results/$2
for i in `seq 0 $1`; do
  mkdir ./cache/$2/$i
  touch ./results/$2/results$i.txt
done
```

C.5 `config.conf`

```
mutChance:60
croChance:50
binOpMutChance:40
operandMutChance:30
nthMostMutChance:30
incrComplexityChance:75
intRange:20
```

C.6 Gen. 18 chromosomes

```
index = 0
index = 0-nthMostFrequentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostRecentlyUsed(0)
index = nthMostRecentlyUsed(0)-nthMostFrequentlyUsed(0)
index = 0
index = 0*nthMostRecentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostRecentlyUsed(0)
index = 0
index = 0-nthMostRecentlyUsed(0)
index = 0
index = nthMostRecentlyUsed(0)-nthMostFrequentlyUsed(0)
index = nthMostRecentlyUsed(0)*nthMostRecentlyUsed(0)
index = nthMostRecentlyUsed(0)+nthMostRecentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostFrequentlyUsed(0)
index = 0/nthMostRecentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostFrequentlyUsed(0)
index = nthMostRecentlyUsed(0)*nthMostRecentlyUsed(0)
index = 0-nthMostRecentlyUsed(0)
index = nthMostRecentlyUsed(0)/nthMostRecentlyUsed(0)
index = 0*3
index = nthMostFrequentlyUsed(0)+nthMostFrequentlyUsed(0)
index = nthMostRecentlyUsed(0)*nthMostFrequentlyUsed(0)
index = 0+nthMostRecentlyUsed(0)
index = nthMostFrequentlyUsed(0)+nthMostFrequentlyUsed(0)
index = 0
index = 0
index = nthMostRecentlyUsed(0)*13
index = 19*nthMostRecentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostRecentlyUsed(0)
index = 0
index = nthMostFrequentlyUsed(0)/nthMostFrequentlyUsed(0)
index = nthMostRecentlyUsed(0)*nthMostFrequentlyUsed(0)
index = nthMostFrequentlyUsed(0)/nthMostRecentlyUsed(0)
index = 11/random()
index = nthMostRecentlyUsed(0)-nthMostRecentlyUsed(0)
```

APPENDIX D

RUNTESTS.SH

```
#!/bin/bash
# Runs Emergen(e)tic with a series of scripts

GEN="30"
CAN="20"

dnc "emergenetic.dn"

for FILE in ./scripts/*; do
  FILE=${FILE##*/}
  FILE=${FILE%.script}

  dana "emergenetic" $FILE $GEN $CAN

  for DIR in ./cache/*; do
    (cd "$DIR" && rm *.o)
  done
  cd results/ && rm *.txt && cd ..
  tar -jcvf ./archives/$FILE.bz2 cache
  rm -R cache
  cp -a ./cachebackup ./cache
done

echo 'Complete.' >complete.txt
```