The University Of Sheffield.

Animating Search-Based Optimisation Algorithms in the AVMf

Ben Scott
Supervisor: Phil McMinn
COM3610 Dissertation Project
1/05/2019

This report is submitted in partial fulfilment of the requirement for the degree of Master of Computing (MCOMP) by Ben Samuel Scott.

Declaration

"All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

(Ben S Scott)"

Abstract

The AVMf is an opensource Java library dedicated to implementing the ‘Alternating Variable Method’ (AVM), a local search-based optimisation algorithm. This report introduces a design and build style software engineering project in the field of SBSE to create a visualiser module for the AVMf. This will animate the perceived real-time progress of the AVM algorithm as it moves through a fitness landscape. The project aims to design and implement a lightweight software architecture using file-based data transfer to adapt the AVMf codebase to accommodate visualisation functionality. And to ultimately, using an opensource development workflow, contribute the visualiser to the original AVMf source code.

The software architecture specification and implementation are presented, along with the design and implementation of the visualiser module, its GUI and animation sequences. A detailed account of the adaptations made to the existing AVMf code is also included.

Acknowledgements

My supervisor Phil McMinn for all the help, direction and advice.

My housemates for also working late and for the memorable midnight kitchen conferences.

My coursemates for reminding me that we're all afloat in the same boat.

Hope City Church for my community of faith and breakthrough.

My Grandparents for being exemplary in work, prayer and generosity, always inspiring me to keep working hard.

My Mother for always encouraging and lending her ear.

My Dad for having walked the road before me.

And to God who got me through the hardest times of this project.

Table of Contents

Table of Figures

1. Introduction

1.1 The Brief

The following is a copy of the original project brief and is included in this introduction for reference and completeness [1].

> *Search-based optimisation algorithms can be used to find "good enough" solutions to hard problems in a reasonable amount of time. They do this through a "generate and test" method. Potential solutions are generated (at random, initially) and evaluated them using a problem-specific "fitness function". A fitness function provides a numerical score of how "good" a potential solution is for the problem at hand. The optimisation algorithm will then try to improve the fitness of a potential solution by making a series of changes to it. If these changes result in worse fitness scores, the solution may be thrown away, else the solution may be subject to further changes with the goal of further improving fitness.*
>
> *The aim of this project is to implement a visualiser that animates the real-time progress of search-based optimisation algorithms as they optimise solutions for some instance of a problem. For example, they could show how the algorithm is considering different solutions, moving along the 2D or 3D surface of a fitness function, as it evaluates the fitness of those solutions. It could show, graphically, which aspects of the solution are being improved by the algorithm. It could also relay further real times statistics — perhaps in the form of graphs — such as how many fitness function evaluations have been considered by the search process, where or what the best solutions found so far are, etc.*
>
> *These animations would be implemented to work with an open-source library dedicated to a particular effective search-based optimisation algorithm called the "Alternating Variable Method" (AVM). The library is called the "AVMf"* [2] *and is freely available on GitHub* [3].
>
> *Currently, all output produced by the framework is limited to the console only. This project therefore offers the opportunity to contribute to and improve an open-source tool, which could be a valuable addition to your CV.*

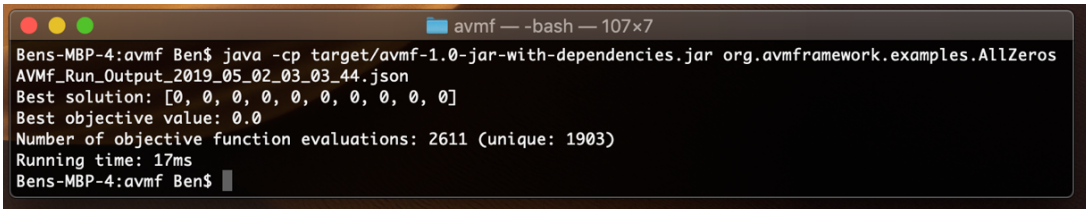

Figure 1: Current output of AVMf to a console window.

The brief begins by introducing search-based optimisation algorithms as an area of Computer Science, describing their purpose and working mechanics. It goes on to describe a project with the aim of animating the real-time progress of these algorithms encapsulated in a visualiser with suggestions of what aspects to visualise. The brief then introduces a project scope, namely the AVM algorithm and the AVMf library. The nature of the work offers the opportunity to contribute to and improve an open source tool.Figure 1

This brief outlines a well-defined software engineering style project, where the objective is to design and build software to a set of requirements and within a scope. The aims of the project will be defined in this direction.

1.2 Project Background

A major application of the discipline of Computer Science is the solving of problems using computational techniques. Arriving at these solutions in an amount of time that can be considered 'reasonable' by human standards remains a key issue in the field since these problems can scale up to a high level of complexity and difficulty. A search-based approach using search-based optimisation algorithms can achieve solutions in on reasonable timescale. They do this by finding local solutions that can be considered "good enough" for all intents and purposes rather than attempting to find the global solution. Perfectly optimal global solutions in practice tend to be a bit 'elusive'. Finding them may be impractical (due to time constraints or computation resources), intractable or even impossible. A local search-based approach offers a practical 'best fit' solution within a reasonable time frame.

In general, search-based optimisation algorithms work by using a "generate and test" method. They randomly generate potential solutions and then evaluate an objective function (fitness function) which gives it a numerical score (fitness) representing how optimal the solution is. Objective functions are written specific to the problem domain. The algorithm manipulates the potential solution in some manner and re-evaluates its fitness. This manipulation and re-evaluation process is repeated in the direction of fitness improvement until a local optimum is found. All search-based optimisation algorithms guided by a fitness function generate what is known as a fitness landscape. This is simply the surface of a fitness function. [4]
It is a hidden feature until visualised and can be drawn by plotting solutions against their fitness.

There is a subset of local search-based optimisation algorithms known as 'Hill Climbing' algorithms [4]. This project is concerned with an algorithm belonging to this set called the "Alternating Variable Method" or "AVM" [4] that works on the general principles laid out above. The specific details and mechanics of the AVM Algorithm will be discussed later in Chapter 2.

A problem solvable by a search-based approach might fall inside a much larger software engineering project. These problems are known as search-based software engineering (SBSE) problems. The use of a search-based algorithm might be inside program logic but there is a notable application for use in testing. For example, the AVM algorithm can applied to automatically generate various types of test data. [5]

An open source library called the "AVM Framework" or "AVMf" [2] is an up to date Java implementation of the AVM algorithm deployable in search-based software engineering projects. It contains much of the recent work and improvements [5] to the AVM. A full introduction to this framework will be given in Chapter 2. It is worth noting that this project's supervisor is one of the primary authors and developers of the AVMf. A brief disambiguation of terms is important here: within the AVMf "fitness" is referred to as "objective value" and "fitness function" is referred to as "objective function". These terms will be used interchangeably throughout the project.

1.3 Project Aims and Scope.

Here the specific aims of the project are stated, and the project scope is defined. Both have been constructed by interpreting the brief and through discussion with the project supervisor.

The objective of this project is to design and build an animated graphical visualiser tool implemented to work specifically with the AVMf. The visualisation is concerned with animating the real-time progress of the Alternating Variable Method, the search-based optimisation algorithm operating within the framework.

The term ‘real-time’ is being interpreted to mean the visualiser requires ‘perceived real-time’ animation by its user. This loose interpretation is needed or severe limits are placed on design possibilities. Some of these are discussed in Chapter 4.

The visualisation should incorporate:

- A representation of the fitness landscape of an AVM algorithm run within the AVMf.
- A representation of each solution are considered at each step of the algorithm.
- Reporting of both static and dynamic statistics.

Part of the visualiser’s purpose is to extend the reporting functionality of the AVMf. Currently the only output is limited to the console (see figure 1) which conveys statistics about an algorithm run but nothing about what’s going on algorithmically. The visualiser aims to help the user understand the mechanics of the underlying algorithm and has intended use in the potential SBSE projects the AVMf might be deployed in.

The project involves finding solutions to the following problems and tasks:

- Designing a software architecture that can be used to adapt the existing code base of the AVMf.
- Using a development workflow that allows straightforward contribution to the AVMf opensource codebase.
- Designing the infographics; the visualisation techniques and animation sequences that must communicate effectively and clearly.
- Designing a GUI in which to display the visualiser.
- Incorporating a system of saving and returning to a past algorithm output to re-visualise it. This allows for review and comparison of algorithm instances, a powerful reporting feature.

The project scope is limited to engineering design and implementation inside the AVMf. It does not extend further out into SBSE.

1.4 Report Structure

Chapter 2: Literature Survey and Background, will discuss the project background and the suitability of potential technologies to be used in achieving the projects aims.

Chapter 3: Requirements and Analysis, breaks down the project aims, requirements and constraints giving consideration to how the projects success can be evaluated.

Chapter 4: Design, gives a detailed account of the project’s software architecture and its design process.

Chapter 5: Implementation and Testing, describes in detail the implementation of the visualiser, looking at its code, elements and feature and how these meets the project requirements.

Chapter 6: Results, Discussion and Evaluation, summarises the achievements of the project and the aims it failed to achieve. It also outlines suggestions of possible further work in and around the project scope.

Chapter 7: Conclusions, summarises the project and closes the discussion of the previous chapters.

1.5 Project Relationship with Personal Degree Programme

This project is a bridge between the theoretical computer science area of algorithms and practical software engineering. Software engineering is a common industrial application of a computer science degree, and this project is advantageous in preparation for that application. It gives the opportunity to work on an existing system and codebase, a commonplace task in industry. Learning how an existing system architecture is put together through examination of theory, code and questioning architects is good practise for industrial work involving maintenance, improvement and feature addition to existing systems.

This relationship also creates a few challenges. There is no team working aspect to this project unlike the usual situation in industry. This affords the opportunity to develop the self-disciple and proficiency needed to work solo on a project. There is also the challenge of loose external project management, as aside from regular supervision meetings I am solely responsible for managing the project. This gives the opportunity to further develop personal project management skills.

2. Literature Survey and Background

This chapter will review literature on and discuss the AVM, the AVMf and the broader field they sit in. It will also place focus on discussion and analysis of potential technologies to be used in achieving the aims of the project. One purpose of a literature survey is to avoid repeating work in a project that has previously been done. As the scope of this project is specific to the AVMf and that currently has no visualisation functionality, the risk of repeated implementation does not exist. There is no evidence of any previous work conducted with similar aims as this project in its scope.

2.1 The AVM and the AVMf

2.1.1 The AVM

The scope of the visualisation in this project is for the progress of the Alternating Variable Method (AVM) algorithm and its variations. The AVM is a local search-based optimisation algorithm belonging to the ‘Hill Climbing’ family. Some of its variations come from recent development work around the AVM.

The paper introducing the AVMf [2] written by the authors/developers: ‘AVMf: An Open-Source Framework and Implementation of the Alternating Variable Method’ [5] lays out the process of the Original and AVM goes on to discuss recent improvements. Harman and McMinn have shown AVM to be quick and effective in local searching [4].

The AVM works by taking a vector of any number of variables that together represent the problem (originally using integers). Each variable in this vector (typically initialised randomly) are handled in turn by a variable search algorithm that applies an objective function. The objective function outputs a value that refers to the ‘fitness’ of the variable in regard to the optimum solution of the problem. This is also called the ‘objective value’. In the area of variable search algorithms recent improvements have been made to the AVM.

The Original AVM implements an algorithm named ‘Iterated Pattern Search’ (IPS). The initial step of IPS is to make an ‘exploratory move’ used to establish a direction for making further ‘pattern moves’. An ‘exploratory move’ is made by increasing and decreasing the variable by 1 and evaluating both these numbers using the objective function. If the objective value improves, a positive or negative direction for pattern moves is chosen. This could potentially be positive or negative because an increase or decrease in the variable could improve the objective value. After this direction is established, subsequent pattern moves are made. A ‘pattern move’ is a change of the variable’s value in the direction chosen, increasing in size each with each iteration. This process repeats until a pattern move does not improve the objective value. This event is most likely caused by making a pattern move larger than the difference between the current variable value and the optimum solution. From here IPS goes back to the ‘exploratory move’ step to ‘zero in’ on the optimum solution. If ‘exploratory moves’ do not lead to any improvement in objective value, the current value of the variable is considered its optimum solution and the next variable in the vector is considered. Once every variable in the vector has gone through this process, a final pass of the vector is done and if there is no improvement in objective function AVM is lodged in a local optimum.

At this point there is the option to restart with a fresh initialisation in the hope of finding a different local optimum that might be a better fit. The AVM can be run in this way until a set of termination constraints are satisfied.

2.1.2 Recent improvements to the AVM

McMinn and Kapfhammer [5] briefly describe and summarise two alternative variable search algorithms recently proposed by Kempka [6] [7], ‘Geometric Search’ and ‘Lattice Search’. Geometric Search works in the same way as IPS until the optimum solution is overshot. Geometric Search uses previous moves to bracket the lower and upper limits of the value where the optimum must lie for the variable. A binary search is then used to find the optimum. Lattice search uses Fibonacci numbers as ‘pattern moves’. Both these alternative algorithms can be shown to run faster than IPS in certain conditions [5].

The original work on the AVM only allowed vector variables to represent integers. Harman and McMinn [4] extended this work to allow the AVM to handle variables representing fixpoint numbers and strings [5].

Analysis and understanding of the variations of these algorithms from standard IPS is required because for a visualiser of the AVMf to be complete it must include the ability to display these variants.

2.1.3 The AVMf

The AVM Framework (AVMf) [2] is a Java implementation of the AVM algorithm including recent improvements. It is an open source library, free to use under the MIT licence and the first general, publicly-available library incorporating all the recent developments in the AVM. The AVMf has been implemented as a Maven project [3].

The AVMf library was designed to be easily incorporated into search-based software engineering projects where the AVM algorithm is core or as a component of some more complex custom technique [5]. Configuration and adaptation of the framework is actively encouraged [3]. One major intended application of the AVMf is in the automatic generation of test data for a test harness, consistent with original work by Korel in 1990 [8].

The AVMf also aims to help researchers and practitioners to understand the algorithms underlying the AVM [2]. However, the only way to achieve this understanding currently is to

examine the source code or to read the documentation and research. A visualisation tool for the AVMf algorithms as an additional component of the AVMf toolset would significantly aid users to gain this understanding.

2.2 Maven

Maven is an open source project managed by Apache [9]. It is a build automation tool for Java projects. It has been opensource since 2003, is now regarded as highly stable and has a rich diverse feature set [10]. A central concept to Maven is the use of the Project Object Model (POM), this is represented by marking up project details in an XML file (pom.xml). Details such as project dependencies are described here [10]. Dependency management is an integral Maven feature. Maven can connect to remote repositories containing the code and automatically download the specified versions in pom.xml. AVMf documentation directs the user to use the package feature of Maven to build the project [3]. This command packages compiled code into a JAR distributable format [11]. Any future AVMf dependencies should be managed using Maven.

2.3 JavaFX

To create a visualiser for the algorithms operating within the AVMf, Java libraries that can enable development of GUI's, graphics and animation library are required. Ideally the solution to this would consist of a single library with all the functionality required. This leads to analysis on the suitability of JavaFX for this task.

Oracle defines JavaFX as a library containing a set of graphics and media packages enabling the development of rich client applications [12]. It's designed to operate consistently across diverse platforms.

First introduced in 2007 effectively as a replacement for Swing as the client UI library for Java [13], JavaFX was fully Open-sourced in 2011 after Oracle acquired Sun [12]. As of September 2018 JavaFX will be removed from the Java JDK starting with JDK 11 onwards, a decision made mostly due to the rise of "mobile and web first" application design and development. Commercial support for JavaFX in JDK 8 will remain until 2022 [12].

Support of JavaFX has moved from Oracle to the OpenJFX [14] project in the OpenJDK [15]and is now maintained by interested third parties as a separately distributable open-source module. However, Oracle still continues to contribute towards its open-source development. [12] The Oracle website now redirects the JavaFX section to OpenJFX on the OpenJDK Wiki site (see appendices for details).
Future focused projects should align with Oracle's roadmap utilising JavaFX through the OpenJFX route. Any attempt to use JavaFX through the JDK would require a legacy distribution (JDK 8) and would be tied into a four-year commercial support term due to terminate in 2022.

The removal of JavaFX from the JDK and its discontinuation of support from Oracle raises concerns about usage of the technology in this project and makes it important to examine what the future of JavaFX looks like before deciding to base a project around it.

Donald Smith, (Senior Director of Product Management at Oracle) says in an interview [16] that the development of JavaFX applications remains mostly the same as before its removal from the JDK. The main difference now is JavaFX is treated as any other external stand-alone library. This is standard practice in Java application development and almost all production Java projects rely on third party libraries [17]. Dirk Lemmermann in the same interview [16]

likens using JavaFX to adding JUnit [18](the industry standard unit testing library) to a project. Essentially JavaFX requires no special treatment whatsoever.
One article sees JavaFX's decoupling as an advantage [17], claiming that it will yield finer-grain control in managing dependencies and allow it to develop outside of the JDK release cycle.

The open source nature of JavaFX means it's highly likely to stay available.Contributions to OpenJFX [19] at the time of writing are regular and recent, showing active support from its community. The activity is actually higher than in many open source projects that are relied upon commercially. Oracle also appear to be committed to contribution in some capacity [12]. There is still a place for client, desktop GUI development, its demand just not as widespread as web-based approaches. One such place is in the scientific community (NASA) [20] where this project sits.

OpenJFX is free software licenced by a 'GNU General Public License with Classpath Exception" [14] making it appropriate for use in academic software.

JavaFX can be used to develop modern GUI's, charts, 3D graphics and animations. All of these are required by a project to animate algorithms in the AVMf. Using JavaFX means all of these can be done using API's supplied in a single library. The animation API is particularly powerful; any node (GUI element) can be animated by changing its properties over time. Animations can be run simultaneously and sequenced together. Using a single library has an advantage in lowering the number of project dependencies making the end result easier and simpler to build. An alternative approach would be to use the legacy Java Swing library for building a GUI. Swing although still ubiquitous and supported by Oracle, is now inherently dated and lacks the extra functionality required to build the graphics (charts and animation) needed for the project. It would still need to be coupled with extra third party libraries; JavaFX provides all required functionality this in a single library.

The JavaFX library is proven to be suitable in highly dependable industrial applications. The library has been used by NASA software engineering teams to develop tools that support missions and scientific operations [20]. It has been described as having an essential role in developing analysis and visualisation tools [20]. The tool this project aims to build, though not nearly as complex as a NASA mission critical software, does have a similar basic scope; the visualisation of data for analysis. It's logical to conclude that the JavaFX library acquired through OpenJFX will be more than adequate for the requirements of this project.

2.4 GSON

GSON an open source Java library that can be used to convert Java Objects into their JSON representation, and to convert JSON strings into equivalent Java objects. It is maintained by Google on GitHub but is not an officially supported Google product. [21]

GSON has easy inclusion into a java project using Gradle or Maven managed dependencies. It has very simple code syntax providing two main methods to parse Java objects and JSON: `toJson()` and `fromJson()` [21]. It has an advantage over other opensource libraries allowing the conversion of Java objects whose source code is not available. GSON is considered a standard and high-performance JSON library for Java [22].

Licensed under the Apache Licence 2.0, GSON is appropriate for use in this project. The licence permits commercial use, modification, distribution, patent and private use [23], all possible use cases of application of the AVMf in SBSE. The main licence conditions are preservation of copyright and licence notes [23].

2.5 Opensource development, Contribution and Workflow

The process of contributing to an opensource project hosted on GitHub beings with a fork of the repository [24]. A fork is a copy of another project stored in your personal account. They bridge the gap between the original repository and your own [25]. After forking a project, a local copy of the forked repository can then be obtained by Git cloning. The project is ready for work and development after it has been setup and configured to work in the local environment [24]. Any development should be done following a branching model.
When a contribution is ready to be made to the original project this can be done using a GitHub feature called a 'pull request'. This is a request for the changes to be reviewed by the project maintainers and then merged into the codebase. They might ask you to review your code before merging or rejecting it. It is important to make pull requests with clear documentation to make this process smooth [24].

A popular Git branching model to achieve a disciplined version control workflow is Git-Flow [26]. The Model has two main branches that have an infinite lifespan: master and develop. Master always holds source code in a production-ready state. Develop contains the latest added features for the next release. Develop is merged into master whenever it reaches a stable point, and this is considered a production release. The model has a number of supporting branches of limited lifespan where the development work actually happens. The general rule is, for bug fixing, master is branched and for feature work develop is branched [24] [26]. When work is complete on these branches, they are merged back into the appropriate main branch ready for release.

3. Requirements and Analysis

This chapter will break down the requirements and objectives and problems of this project in detail and analyse Constraints put on the project as a result. It won't go into design solutions as this is reserved for discussion in Chapter 4. Instead it will loosely suggest general approaches to solutions.

3.1 Motivation for Building a Visualiser

A key question to answer is what motivation exists for building a visualiser for the AVMf. Following is an analysis of the motivation.

The current statistical output is generated as part of the reporting functionality of the AVMf using the monitor class. This is limited to the console only and can display: the best solution, best objective value, the number of objective function evaluations, the number of restarts and the execution time of the search in milliseconds. Though these statistics are interesting they are all post search execution data and convey very little about what is happening algorithmically see figure 1.

There are at least two clear use cases for a visualiser and both are far from purely aesthetic motivations. The first relates to one of the original aims of the AVMf; to enable users to understand the algorithmic behaviour in AVM's internal workings. This is useful when wanting to deploy the AVMf in an SBSE project. As previously discussed, the only way of gaining this understanding in the AVMf's present state is by examining its code or reading it's documentation. Enabling users to actually see a visual representation of how the algorithm is

working will go a long way to helping achieve this aim, providing another avenue to gain understanding of the AVM. When processes are described visually they are often much simpler for the brain to grasp as opposed to textual description.

The second case is as a problem solving aid while configuring an adaptation of the AVMf for a search-based software engineering problem. A major part of the configuration process lies in constructing a vector that adequately represents the problem and in writing an effective objective function to evaluate the fitness of this vector. Vast amounts of data can be presented in a single graphic allowing comprehension of an entire dataset in this way. Relationships can naturally and easily be discovered when data is presented visually. In the AVMf, after all of the data collated about each potential solution and their objective values are drawn up visually the result will be a 'fitness landscape'. A tool to visualise the fitness landscape of a particular implementation of the AVMf could be revealing of its effectiveness. It has possible uses in debugging highly dimensional vector problem representations. The user will be able to compare what is expected with what actually going on shown by the visualiser.

3.2 Remaining within the Java paradigm

A major project requirement is to remain within the Java programming paradigm. Contributing to an already existing Java project is inherently simpler by adapting existing code and writing new functionality. Powerful visualisation tools libraries may exist outside of this world but using any non-Java programming languages would create an issue in interfacing between domains. This approach might end up with two almost sperate applications communicating over an interface rather than a single, integrated application negating a project goal of contributing to an opensource tool. This project avoids added complexity and time cost by limiting itself inside the bounds of Java. And as previously discussed, there are tools readily available in the Java world that can be leveraged to achieve this projects requirements. One such tool is Maven [9].

3.3 Integration with Existing Codebase

The aim is to contribute to the AVMf open source library by integrating the visualiser tool with it. The proposed way of doing this is by adding a visualiser package that connects to the existing code. Therefore, all code produced should stay within the Java paradigm. The most appropriate design choice would take the form of a desktop GUI to display simple and clear animated infographics conveying rich information from presented data. Any GUI implemented should be intuitive and lead a user though it with hints and implicit suggestions on how to use it. A software architecture should be designed to solve how to do this integration.

Having made the decision to make this project a contribution to an existing open source Java tool, consideration needs to be given to how this will be done. The proposed solution is to utilise the JavaFx library in a new Java package named visualiser. The contained classes will give all the functionality required to achieve the projects visualisation goals. The data to be processed and visualised inside these classes will be passed through using Java Listeners added to the existing codebase. This method will allow lightweight alterations to the existing framework avoiding large scale changes to it.

The visualiser should be launchable from within and without an AMVf based application. Inside using a similar principle to how the Monitor class currently works using method calls. This could possibly use command line arguments to run setting an option to start the visualiser tool. Outside of a search-based application the console could be used to launch the visualiser.

This function is also required because a previous data set from a certain configuration of the AVMf can then be loaded, reviewed and compared with others, independently of the current configuration. Hence the visualiser needs a way of saving and loading data for future reference.

3.4 Visualisation and Animation

Animation of AVM progress in perceived real-time through a fitness landscape is required in this project. The basic architecture for visualisation will be one of capture and playback. While the AVMf is running a search, the visualiser will capture all relevant data for display. Post execution, this data will then be presented graphically, and any animation of the search process played back. The reason for choosing this kind of architecture is to make playback happen at a human readable rate. The time taken to run an AVM search will vary depending on the complexity of computing the objective function and termination policies. This time could range from a few milliseconds to much longer. Post execution playback will give illusion of a real time process but at a human readable speed.

The most important aspect of the AVMf to visualise is the ‘fitness landscape’, as it conveys more useful information than any other aspect. The major issue here is visualising high-dimensional vector representations (any more than 3 dimensions get tricky to visualise). The proposed way of doing this takes inspiration from the AVM algorithm itself in the way it optimises each vector variable in turn. The visualiser will first aim to model the fitness landscape of all considered solutions for each vector variable. This model will take the form of an animated two-dimensional line graph with solutions on the X axis plotted against fitness on the Y axis. There could be set of axes one per variable in the vector or the data might all fit on the same set of axes. The range of values plotted might be very high, so a solution to viewing all the data might be needed.

Each graph will be annotated graphically highlighting some of key features of the fitness landscape. These include: the current best solution, a visual gap between current best solution and the potential optimum solution, and what solution is currently being considered. The graphs and annotations will be animated to grow and develop during playback. Specific consideration will be given to the animation and presentation of the ‘exploratory move’ step of the AVM algorithm, making very clear what is going on. The graphs will also have a zoom function giving the user the ability to examine specific parts of a search in more detail.

It is also proposed that an overall representation of the fitness landscape for the whole vector be presented in a similar fashion. However more work and research need to be completed be for a solution to this be designed.

3.5 Opensource Development Workflow and Contribution

The AVMf codebase is stored in a repository on GitHub, an ideal location for opensource contribution. This project’s ultimate end goal is to through a pull request on GitHub contribute the visualiser to the original source code. To make this possible a disciplined end to end opensource workflow should be used in code development. A possible solution is to use a custom branching model inspired by cutting down the popular GitFlow [26] discussed in Chapter 2 into something more lightweight. As this project is being developed solo it might only need two branches; a master and a develop branch.

4. Design

This chapter introduces the software architecture intended for use in this project and outlines the decision-making process and analysis in its designing. It then goes on to describe in detail the function of each major section in this intended architecture.

4.1 Online Architecture vs Offline Architecture.

The foundational design decision of the project is how to adapt the existing code of the AVMf to incorporate visualiser functionality. All further decisions are based on this architecture choice and so it has significant consequences for further design. Consideration was given to two major software architecture design approaches, referred to as online and offline architectures within the scope of this project. An online solution integrates visualisation into the mechanics of the algorithm and in an offline solution, visualisation is detached from the mechanics. Their principles and potential implementations differ considerably, with the main contrast being about where the visualisation processing should take place. A discussion of their advantages and disadvantages follows.

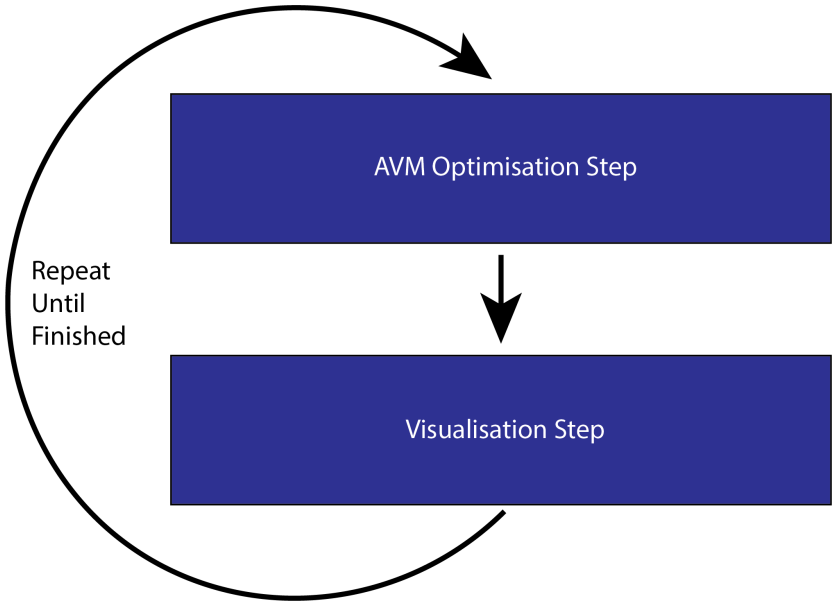

Figure 2: Diagram showing a possible online software architecture solution

Figure 2 shows a possible but basic architecture for an online solution. It follows a simple two step cycle of optimisation and the visualisation of this optimisation until the algorithm is complete. A vector would subjected to change in its variables, evaluated using the objective function, then the resulting data would be animated visually in the visualisation step. An online solution has the immediate advantage (by the nature of design) of fulfilling the requirement for algorithm progress to be animated in perceived real-time by actually doing this in true real-

time. This however is about where the advantages of the design end. A number of its disadvantages are now discussed.

Another requirement is for animation playback to be synchronised with human attention span, and therefore it should be at readable pace. An online solution runs the risk of not reliably meeting this requirement due to the unpredictability of computation time in the AVMf. The time taken to execute the optimisation step is dependent on the complexity (and therefore time) of computing the custom problem specific objective function implemented in an instance of the AVMf. Time taken is also dependent on hardware, though except from in extreme cases is unlikely to have as much effect as the complexity of objective function.

The examples supplied with the AVMf run in a handful of milliseconds, with each computation of their objective functions taking only a fraction of that. This number, though an intuition of run time should not be taken as a predictor of AVMf average run time. The examples are relatively simple, and applications of the AVMf scale up to far more complex search-based problems.

As long as the time taken to complete the optimisation step is under a certain threshold, the animation would still feel responsive the viewer and is therefore acceptable. If, however this time strays above this threshold due to a highly complex objective function, an online solution runs the risk of poor synchronisation with human attention spans and appearing unresponsive or slow. To know if this solution would consistently (on average) meet the human readable pace of animation requirement, the average time to compute an objective function must be less this threshold.

Without concrete statistics on runtime found through testing the AVMf on search-based problems of varying complexity (a near to exhaustive set of problems would be needed for certainty), It Is not possible to make this conclusion aside from guesswork. This research would be a large undertaking, requiring an implementation of the AVMf for every problem considered. It would require time and resource unavailable in this project. There are no other advantages apart from making conclusions about animation synchronisation to warrant such an investigation either. It is likely that at best these figures would only really be able to predict consistency for average performance and not global performance, as problems with abnormal complexity levels might on occasion appear. These figures cannot take into account any future search-based problems or their applications which could be more complex than any current ones known.

In an online solution the user would also never truly be able to change the rate of animation. This is because no control is available over the duration of the computation stage, only the visualisation stage.

To implement an online solution an involved and heavyweight adaptation to the existing codebase of the AVMf will be required. This comes with the disadvantage of needing more time to implement than a simpler solution. A successful implementation would also need a deeper understanding level of the AVMf (possible as much expert knowledge the authors) because the adaptations might need to go as far into the frame work as the mechanics of the algorithm.

An online architecture is not suitable for future review of the data output by the AVMf, its more suited to showing what is happening during a run. If the suggested architecture in figure 2 was modified so that visualised data could be saved, further work would have to be done to adapt the visualiser to load this data back in separately from a run of the AVMf. This starts to have elements of an offline architecture.

Since both tasks of visualisation and search-based optimisation processing are intertwined, an online approach holds on to both system resources at once. In an offline approach these tasks are done sequentially so that at any one time less system resources are being used.

An online solution is inherently dependent on the AVM processing stage leading to a number of limitations. Many of these are undesirable since they do not meet the project requirements, or they cannot guarantee requirements will be met reliably. In light of this conclusion an alternative should be considered that also meets the requirement that algorithmic progress is animated in real time.

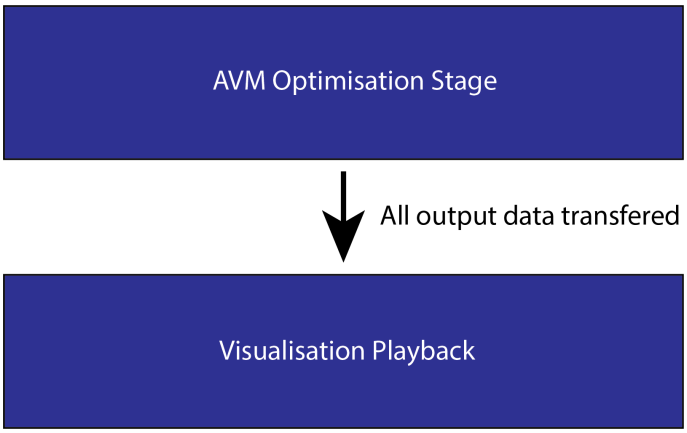

Figure 3: Diagram showing a possible offline software architecture solution

An offline solution (figure 3) where the visualisation processing is detached from the algorithmic mechanics of the AVMf does meet this requirement but in a slightly different manner. Since the visualisation happens post-optimisation, it does not display the animation in 'true real-time' but does give the impression that it is. The visualisation presented to the user would be exactly the same as if it had happened in real-time except that it's a playback. A disadvantage is that performance of the algorithm (corresponding the complexity of and possibly how well written the objective function is implemented) is not represented in the animation and therefore not visualised. (This would not have been well visualised in an online solution either. Poor performance would have appeared as stutter or delay between animating considered solutions.) A mitigation for this disadvantage could be the inclusion of performance reporting statistics in the visualiser GUI.

Synchronisation with human attention spans is easily achievable using this architecture because complete control over animation is available. Animation and visualisation processing is not dependent on any other part of the system. This opens up the possibility of implementing custom user control over animation with controls like pausing, skipping to certain parts of the animation and changing its rate. In an online design for instance, doing any of these actions would also have to set the AVM algorithm to the same state and in many case would not be possible.

An offline solution is much more lightweight in terms of modifications to existing code modules. It alters existing code only to capture data needed for visualisation and then all other logic is encapsulated in the addition of new modules. Smaller changes make updating a SBSE project already incorporating the AVMf as a library/API much easier to do. The visualiser could be designed in such a way that no existing functionality is compromised. This is also more suitable for achieving the goal of contributing to an open source system. Where fewer changes exist to the starting codebase the task of maintenance for the original authors is

simplified. A highly complex modification such as an online architecture might have diverged too far to be considered a viable contribution to an open source project. A pull request on GitHub for instance might not be approved and the work might stay as a fork of the original project. This should be avoided since as the original project continues to be developed further divergence between the fork and the main project happens and could continue until the fork is no longer relevant or useful.

The nature of offline design takes all data visualisation data in as a whole, therefore integrating support for saving and loading previous data sets for future review, visualisation and comparison becomes fairly straightforward.

An offline architecture is considered the best engineering design option available to this project. It is overwhelmingly advantageous in meeting the project requirements over an online architecture and has no disadvantages that are not either minor or cannot be easily mitigated by further design choices.

4.2 Proposed Offline Architecture

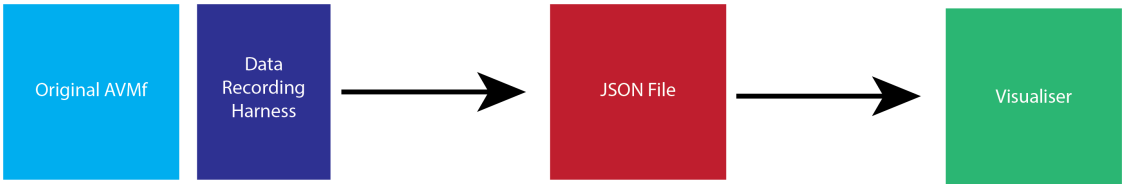

Figure 4: Diagram showing the project's proposed offline architecture solution

Figure 4 shows the offline software architecture design to be used in this project. There are 3 constituent parts:

- The Data Recording Harness
- A JSON file
- The visualiser module itself

The Data recording harness is attached to the original AVMf. It captures data needed for visualisation and outputs it to a JSON file. This data file is then loaded into the visualiser module and the animation happens.

4.2.1 Data Recording Harness

This is the only part of the offline architecture that must modify already existing AVMf code, enabling the solution to be lightweight. It is an interface that extracts the data needed for visualisation and records it to a file. Consideration was given to writing a new fresh class system to attach or adapting existing AVMf classes with the base functionality required.

The first solution considered was an event driven approach needing a fresh class system. The harness would consist of two parts. In the first part the AVMf would be adapted to recognise when certain events occurred; for example: the computation of an objective value, The end of a variable's optimisation or a restart in the AVM algorithm. These events would generate data

that would be encapsulated in an event object. The second part running asynchronously would be listening for these events and then handling them appropriately. The data encapsulated in the event object would be extracted and recorded to file. This approach would require the writing of custom events, triggering these in the right places inside the AVMf internal workings. A set of custom handlers for the events would need writing for the handling and recording system. See figure 5 for an diagram of the approach.

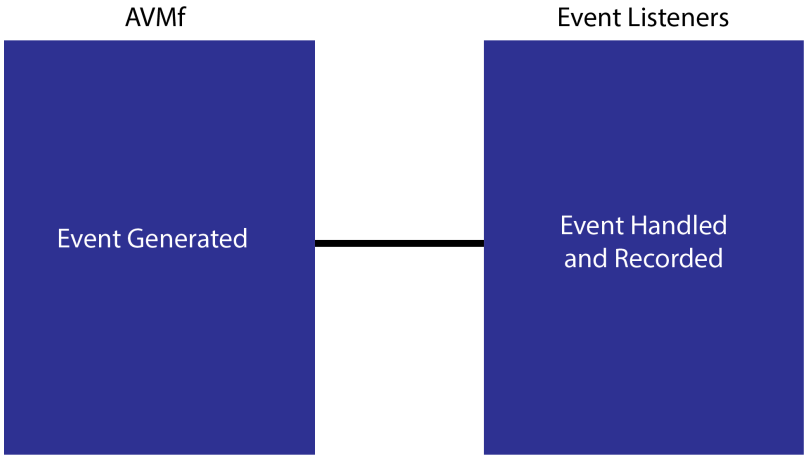

Figure 5: Diagram of a possible event driven data recording harness

Research and investigation into the existing architecture of the AVMf lead to a simpler alternative solution. The Monitor class in the original AVMf is an integral part of the algorithm implementation, designed with the task of observing events in the algorithm, logging and reporting. Much of the base functionality required for a data recording harness is already implemented. It can already extract data from the internal workings of the AVMf and is wired into many of the prospective points the harness would need to be. Monitor's functionality can be extended to include extraction and encapsulation of visualisation data and file recording.

This solutions simplicity is advantageous over an event driven approach because it will produce less code and therefore better achieve project requirements by being even more lightweight. There will be no entirely new class attached to the AVMf and there will be no large set of events and event handlers to maintain. The time savings can but put into focusing on the visualisation side of the implementation.

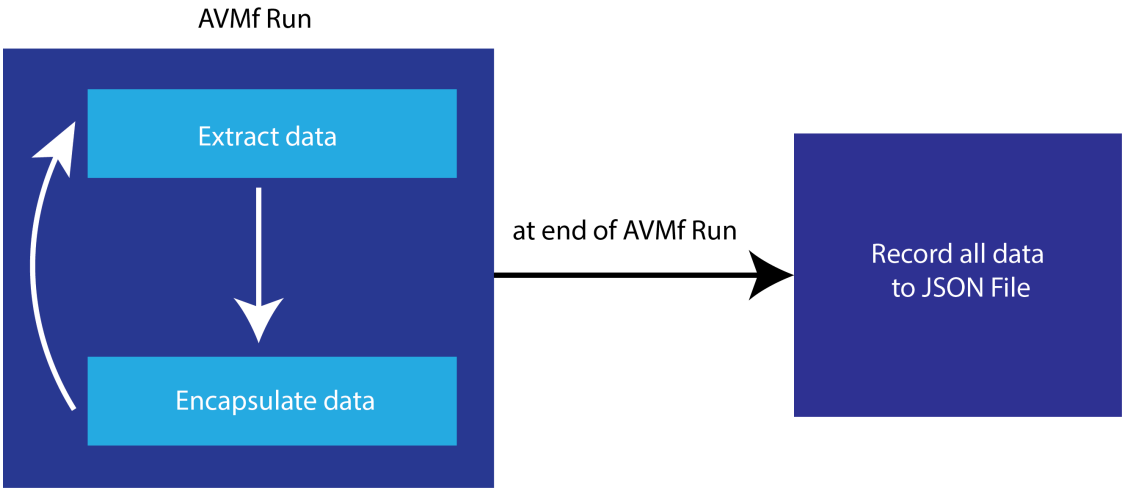

Figure 6: Diagram showing the data capture process in data recording harness

Figure 6 shows the data capture process to be implemented in an adaptation of Monitor. As the AVMf generates data needed in the visualiser it is extracted and encapsulated within a Java object. At the completion of an AVMf run these objects will be converted into JSON and written to file once.

4.2.2 JSON Files – Data Interface

The middle section of the software architecture is a JSON file used as a data interface to transfer data from the AVMf to the Visualiser.

A file-based approach has some advantages as opposed to transfer data within program logic. Data can be saved and can kept for review at a later date, allowing revisualisation and achieving a project aim. Files can also be transferred which would allow the same data to be visualised on different machines in different geographical locations, bringing benefits to teams using the AVMf.

There was some early experimentation with using custom txt files that quickly revealed drawbacks. The stored data was not semantically understandable as it relied on an unlabelled data structure. Using the file relied on detailed knowledge of its structure and what each line represents. To use it effectively would have required accompanying documentation or the inclusion of key pairs to label the data.

These drawbacks lead to the consideration of JSON files which have a few advantages:

Data is labelled, semantically understandable and straightforward to parse.
Sophisticated libraries like GSON exist for handling JSON in Java projects, so using it removes the need to write a parser for a custom txt file.
JSON is a standard highly interchangeable format and opens up the possibility of importing data output from the AVMf in other programming paradigms that can parse JSON. Web technology where JSON is a standard format and graphical innovation is currently ongoing has strong potential paradigm. The project's opensource contribution requirements limit this advantage but it could be useful for any future work.

The structure of the JSON files will be the same as the Java object the visualisation data is encapsulated inside; starting with a header containing statistics and body containing the output of each iteration of the AVMf. Refer to figure 8 in Chapter 5 for to see this structure.

4.2.3 The Visualiser

This module is the front end GUI that contains the visualiser and presents the animation of AVMf progress to the user.

There are two intended starting points to launch the visualiser GUI. It could be launched from an implementation of the AVMf like the supplied examples working in a very similar way to how the Monitor class currently reports. Or it could be launched standalone outside of an implementation. This would require the loading of a previously output JSON file. The best way of locating a file would be to use file browser built in to the GUI.

The Fitness Landscape Graph:

If just the mechanics of the AVM algorithm were to be visualised and animated the design approach could have been more abstract. But the requirements state that a solution to visualising the fitness landscape of an output is needed. A natural solution this is to use a line graph. For each vector considered by the AVM an objective value is calculated. Splitting this vector into its variables and pairing each one up with the objective value for the vector constructs x and y data pairs. The plotting of these data pairs draws the 2D fitness landscape of each vector variable. All these 'sub-landscapes' can be plotted on the same line graph to see the overall fitness landscape of the vector. This solves multidimensionality issues discussed in Chapter 3.
This proposed way of dealing with potential large value ranges of data on this graph where scale might cause loss of detail is to implement zooming and panning. This allows detailed inspection across the entire data range regardless of scale.

Animation:

There two aspects that will be animated. A representation of the vector being optimised and the fitness landscape of this vector. The animation of these elements together fulfils the project requirement of animating the progress the AVM algorithm. An animation step looks like this:

- The vector representation is updated to show the current values being considered.
- A sequence of frames animating this point on the fitness landscape graph is played.

The animation sequence was finalised through tinkering with the JavaFX animation API. Initially the concept was to show the fitness landscape growing as each variable's value was changed, objective value computed, and data point added to graph. It soon became apparent that this was not possible to do using JavaFX's built in animation API. After a series has been created and added to a line chart, it's represented by a line going thorough but separate to the data points. The line does not change/shrink/grow dynamically when data points are animated. This is inherent in the structure of JavaFX line chart design. There was a potential option of writing an adaption/own API using JavaFX to make the functionality available. This would need to have its own controls, and this would have taken considerable time and effort. The undertaking could be viewed as a project in its own right.

The JavaFX animation API has a number of built in controls for animation, (pause, play, and jumping to cue points within animation sequences) that would be of significant benefit in terms of functionality and implementation time. The original design solution was adapted to allow the use of the API in the project.
Because of all the data pairs are loaded from a file and into the line graph, the whole fitness landscape is available at the beginning of animation. It's generated by drawing a line through all connected points. This solution is actually a more effective graphical communicator than the initial concept because it makes the user think about the fitness landscape as a whole before any points on it are considered. The only disadvantage is that the evolving knowledge

about the landscape during the AVMf run is not captured as part of the animation. The animation for each variable has a two main parts. Firstly, its whole fitness landscape animated in. Then secondly every point on that landscape is sequentially animated in. A summary of the two concepts solutions can be seen in figure 7:

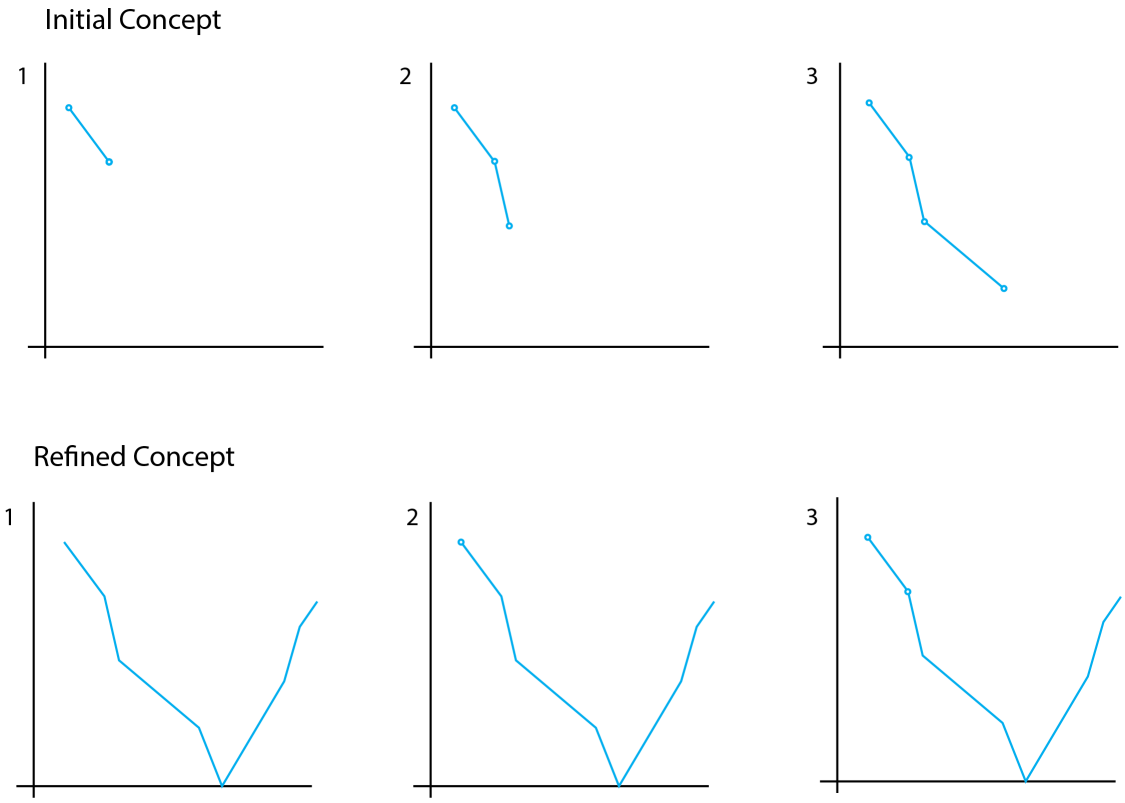

Figure 7: Mock-ups of the two concepts to animate a fitness landscape

An area on the GUI will be dedicated to Animation control, including controls such as:

- pause,
- play,
- restart
- skip to next variable
- etc…

Reporting:

Another area an area will be given to reporting statistics about:

- the configuration of the AVMf
- the data produced from the run of the AVMf
- reporting on the real-time progress of the AVM algorithm.
- reporting on the progress and status of animation:

5. Implementation and Testing

This chapter describes in detail the implementation of the visualiser, looking at its code, elements and features and how these meets the project requirements.

5.1 Opensource Development

Development on this project began by setting up an opensource workflow to allow straightforward contribution the AVMf. The GitHub repository [3] was forked, cloned and the AVMf configured to run correctly locally. The branching strategy for version control was kept simple but loosely based on Git-Flow [26]. A master branch was kept for the original state of the code while all development work was done on the develop branch to be merged back into master at the end of the project when all added features are stable.

5.2 Visualiser Java Package

A new Java package named visualiser has been added to the AVMf codebase containing the following classes:

- AvmfRunHeader
- AvmfIterationOutput
- AvmfRunLog
- Launcher
- GUI

The visualiser package has two new dependencies, the JavaFX and GSON libraries. They are managed using Maven and have been added to the pom.xml file.

`AvmfRunHeader` contains data for reporting statistics on: the configuration the AVMf instance including termination policy and also statistics about an algorithm run.

`AvmfIterationOutput` contains data representing a full single iteration of the AVM algorithm. A representation of the vector and its corresponding objective value are encapsulated along with associated integers identifying the current restart number and the current variable being optimised.

`AvmfRunLog` encapsulates an `AvmfRunHeader` object and an array list of `AvmfIterationOutput` objects. this object that gets serialised to a JSON file. See figures 9 and 11.

`Launcher` – handles different entry points for visualiser launch and code for deserialising a JSON file.

`GUI` – contains all logic for the GUI, visualisation and animation.

Figure 8: Diagram showing the structure of an AvmfRunLog java object

5.3 Data Recording Harness

Purpose and design of the harness is covered in Chapter 4. This illustrates how the lightweight changes to the existing AVMf codebase implement data capture, encapsulation and file writing and meet project requirements.

The harness utilises three objects from the new visualiser package: `AvmfRunHeader`, `AvmfIterationOutput` and `AvmfRunLog`. It also modified the following existing AVMf classes.

Monitor:
The main logic additions have been made here. Monitor has a new instance of an `AvmfRunLog` object called `runLog` for recording captured data into. Its constructor has been overloaded to take two more arguments: a Boolean flag `useVisualiser` and a string for the name of the search type to be used in reporting. Overloading allows the optional use of the visualiser meeting project requirements.
A few methods have also been added. The `recordKeyPair()` method – creates a new `AvmfIterationOutput` object and adds it to the array list of the `runLog` object.
The `generateFileName()` method generates a dynamic file name using the format: Avmf_run_output + timestamp.json.

The existing `observeTermination()` method has been modified on algorithm termination the GSON library is used to serialise the `runLog` object to a JSON file with a name generated by the `generateFileName()` method. See figure 10 for reference.

AVM:
The search method in AVM returns a new instance of Monitor so has also been overloaded to make it compatible with the arguments needed by the overloaded Monitor constructor.

ObjectiveFunction:
An int variable `iteration`, with methods for setting and incrementing is added. this variable is to keep track of the variable currently being optimised. A call to the `Monitor.recordKeyPair()` method has been added inside `ObjectiveFunction's evaluate()` method, to allow the extraction of visualisation data (vector and objective value pairs) at the required points. Iteration is an argument in this call.

5.4 Launcher

To meet project requirements the visualiser needs two launch pathways:

- launching from within an instance of the AVMf, or
- launching standalone.

To launch the visualiser from an instance of the AVMf two changes need to be made to its configuration. The first is changing the call of the `avm.search()` method to its overloaded version, supplying the arguments shown below. Passing in the Boolean true tells the Monitor class to output a JSON file and the string supplies the search type for reporting.

```
Monitor monitor = avm.search(vector, objFun, true, SEARCH_NAME);
```

The following method call must be added to the configuration afterwards to launch the visualiser.

```
Launcher.launchVisualiser(Monitor.getFileName());
```

It is possible to omit this line but the AVMf will only output a JSON data file but will not launch the visualiser.

To launch the visualiser outside of an instance of the AVMf the following terminal command should be run. Providing the AVMf has been built using Maven in the manner shown in its original documentation [3], this launches the main method of the Launcher class.

```
java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.visualiser.Launcher
```

There is simple logic in the Launcher class to handle these pathways. A static Boolean `fileLoaded` is initialised as true and a static method `loadRunLog()` is defined that uses GSON to de-serialise the file into a static `AvmfRunLog` object called `runLog`.

When launching from an AVMf instance, the `launchVisualiser()` method must be called, supplying the name of the file to be loaded (retrieved from `Monitor`) as an argument. A file-path is not needed here because the program will look in the root of the AVMf for the file as this is the default set directory where the data recording harness saves files. If `fileLoaded` is true (it will always be true here as it's initialised as true and not modified before the conditional) the JSON file will be loaded using the `loadRunLog()` method and parsed into the object runLog. This object's data will be referenced statically when used by the `GUI` class. GUI.launchUI() is then called to start the visualiser GUI.

When the visualiser is launched standalone the main method of launcher will be called. This sets `fileName` to a placeholder value, sets `fileLoaded` to false, then calls the `launchVisualier()` method. In `launchVisualiser()`, if `fileLoaded` is false then the call to `loadRunLog()`is bypassed and `GUI.launchUI` is called without loading any data from file. The first thing that happens at GUI launch is a check to see if `Launcher.fileloaded()` is true. If it's false, before setting up the main visualiser GUI, a JavaFX file browser (figure 9) is launched to find the absolute file-path of a file to be loaded. Pressing cancel will abort the visualiser launch. The file-path is then passed to `Launcher.loadRunLog()` to load and parse the JSON file. From here the GUI resumes its launch with data loaded.

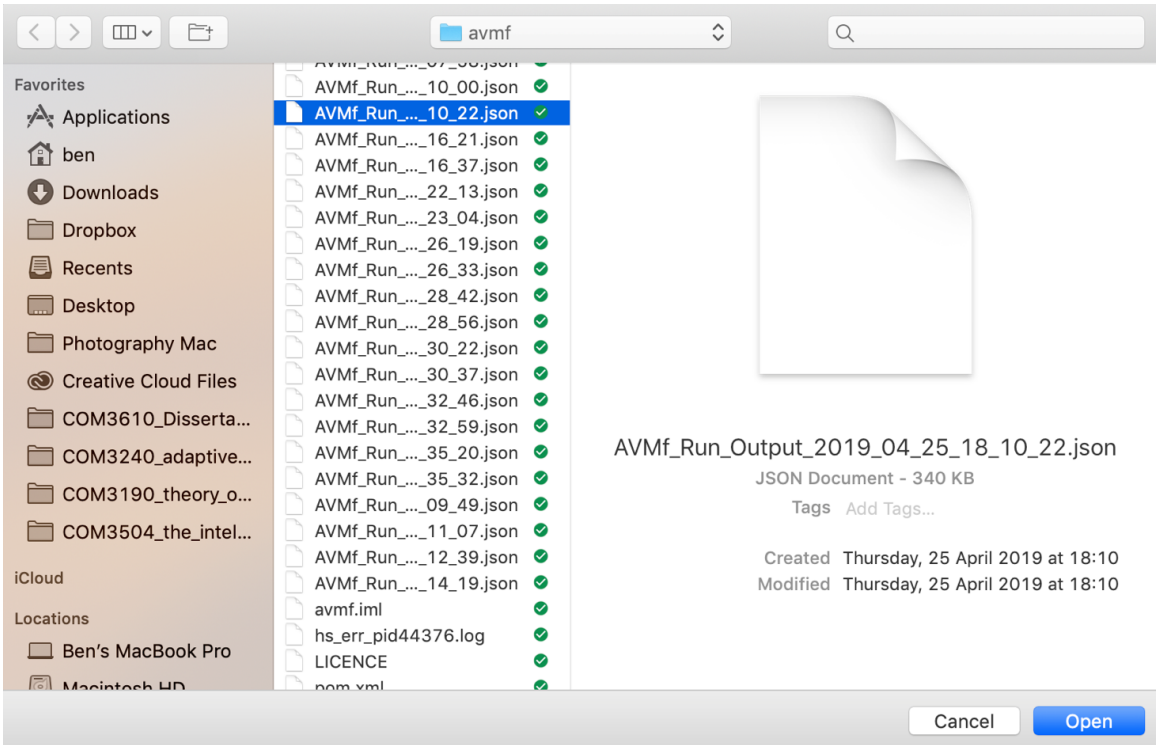

Figure 9: Screenshot of JavaFx file browser to locate and load JSON file.

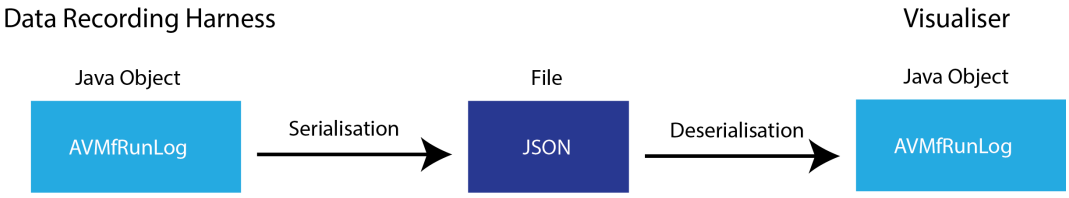

Figure 10: Diagram showing the parsing process of Java objects to and from JSON

5.5 The Visualiser GUI and Animation

5.5.1 The fitness landscape line chart

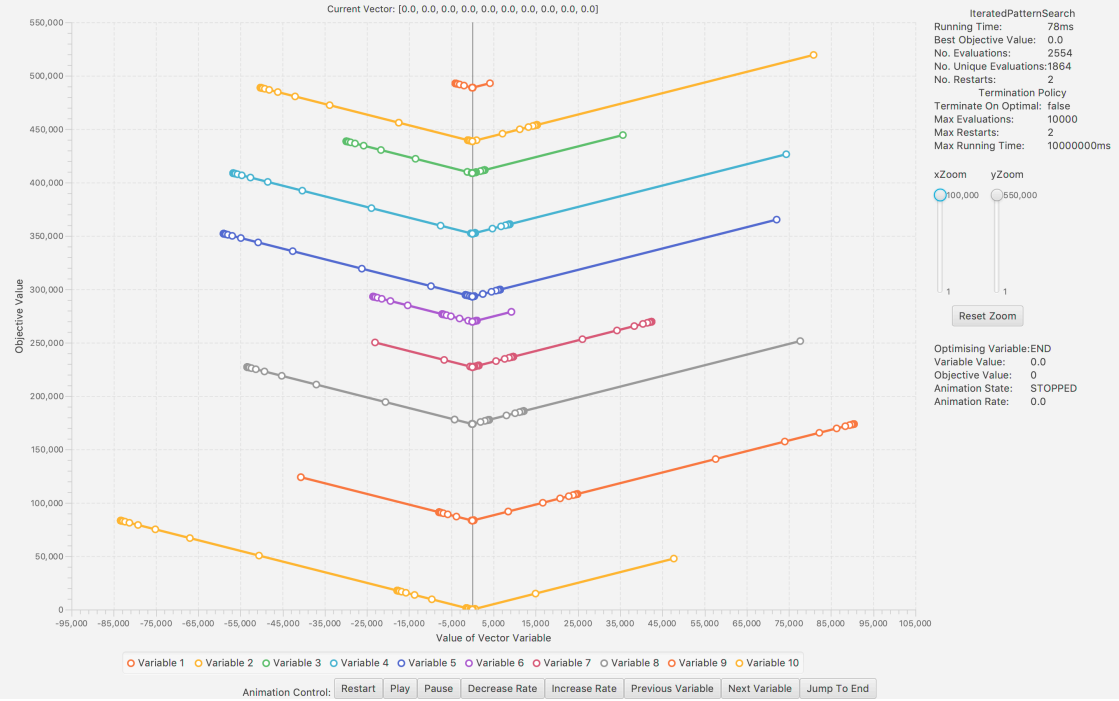

Figure 11: Overview of the GUI at the end of animation

Figure 11 shows the entire visualiser GUI at the end of an animation run. The fitness landscape of each variable is displayed on a 2D JavaFX line Graph.

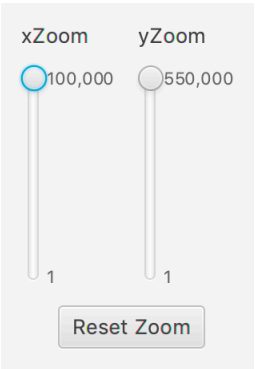

Figure 12: Screenshot of graph zoom slider controls

The sliders shown in figure 12 zoom the graph logarithmically. The value is read from the slider between its maximum and minimum values and is converted into a logarithmic value used to change the bounds of the respective graph axes. The code containing maths for this can be found in the appendix in figure 22. After zoom the graph can be panned by clicking and dragging on the chart area. This allows the user to see every part of the fitness landscape at any zoom level.

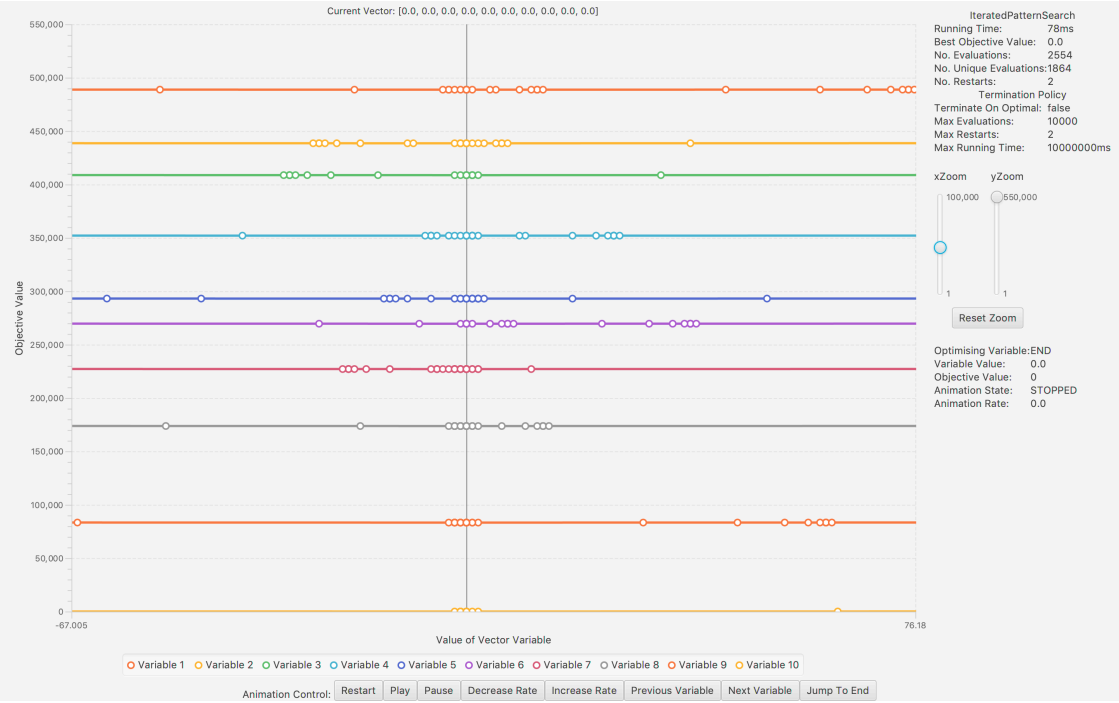

Figure 13: Example of graph with the X axis zoomed

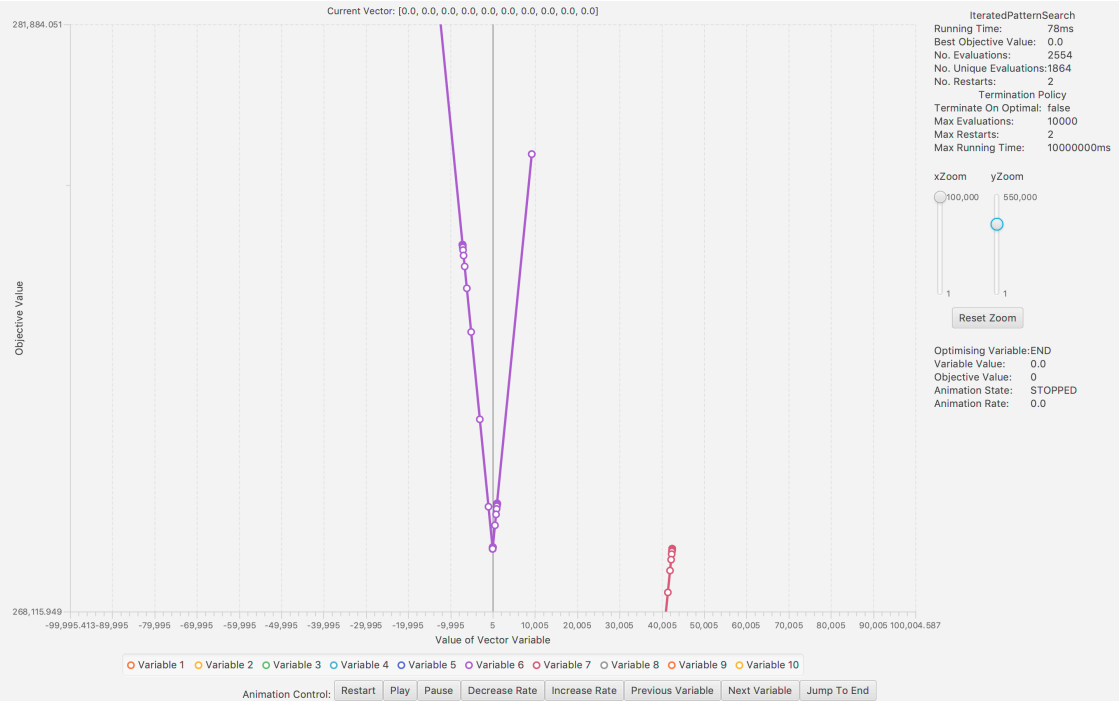

Figure 14: Example of graph with zoomed Y axis

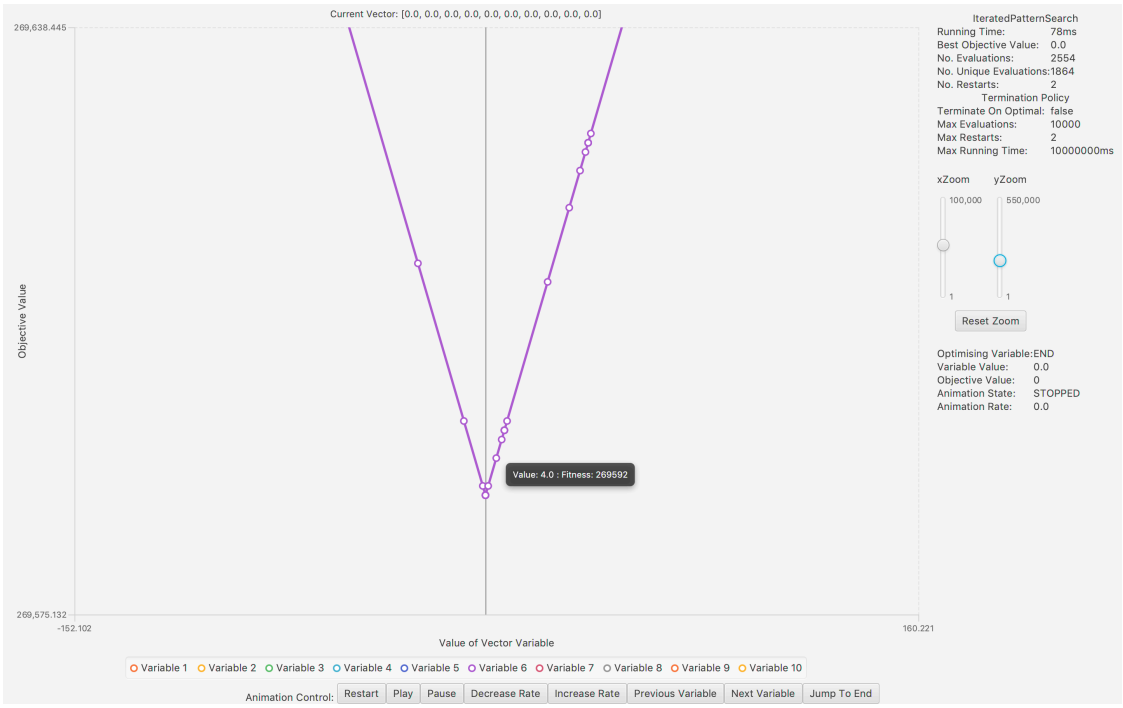

Figure 15: An example of both x and y axis zoom, graph panned to a specific area of interest and tooltip shown for data point

figure 16, shows how each series can be turned on and off by clicking on its legend icon. This functionality is required to make sense of the graph when there are many variables in a vector and the chart area gets crowded or when two series overlap so both can be inspected separately. Figure 17 shows how the legend icons are greyed out when not visible

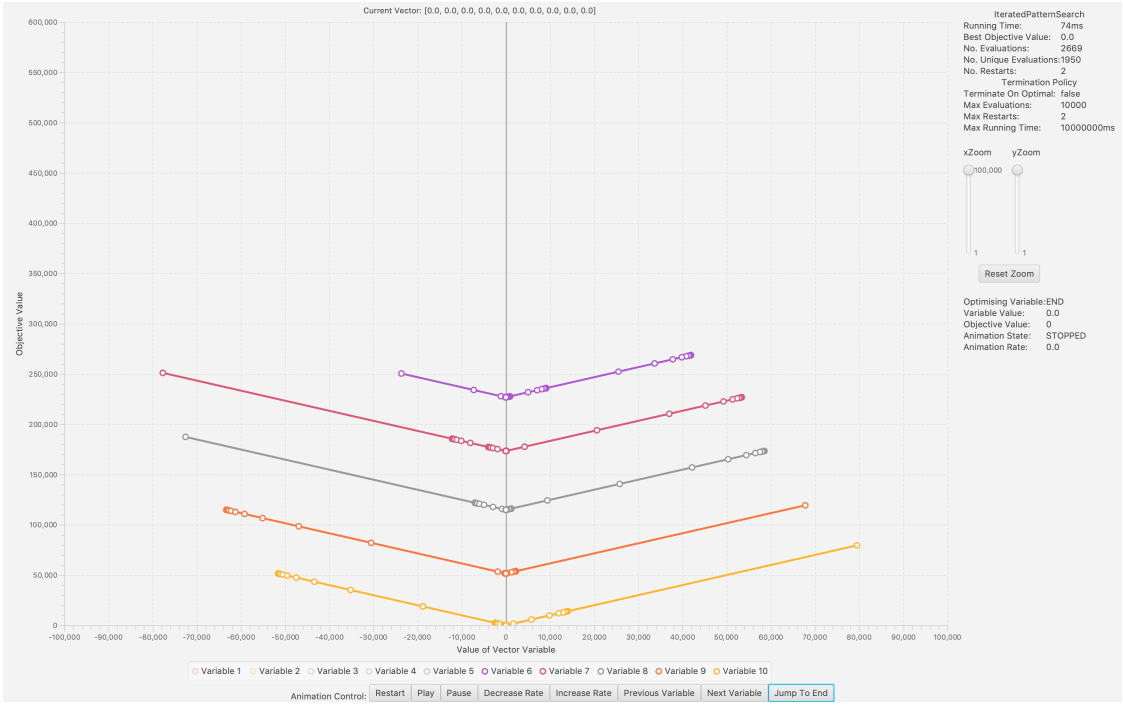

Figure 16: Screenshot showing the first 5 variables/series toggled off

Figure 17: Close up of the chart legend showing the first 5 series greyed out because they are turned off

5.5.2 Animation

Animation sequence design proposed in chapter 4 has been implemented. For each variable:

- The fitness landscape fades in.
- Each considered value for sequentially fades in and simultaneously shrinks into its position in the landscape. simultaneously the vector represented above the graph is updated to show the current values being considered. This paired animation design shows clearly the value is being is being considered by the AVM.

An area under the line chart is dedicated to controlling the animation using the features of the JavaFX animation API. Figure 18 shows this control area.

Figure 18: The Animation control section

The ‘Restart’ button is initially labelled ‘Start’ and changes to ‘Restart’ permanently after the first time it is clicked, and animation first begins.
The rate of animation has a range of 1.0 – 10.0, this number being a multiplier of the default rate. For program logic reasons it is not possible to change the rate while the animation is paused. ‘Previous Variable’, ‘Next Variable’ and ‘Jump To End’ set the animation sequence to their respective places using JavaFX’s animation API feature of Cue Points. After a Jump to end, animation must be restarted to use any of the animation controls.

5.5.3 Reporting

There are two main reporting sections. One shows the data from the header of the JSON file, this is all unchanging statistics about the AVMf run and configuration. See figure 19.

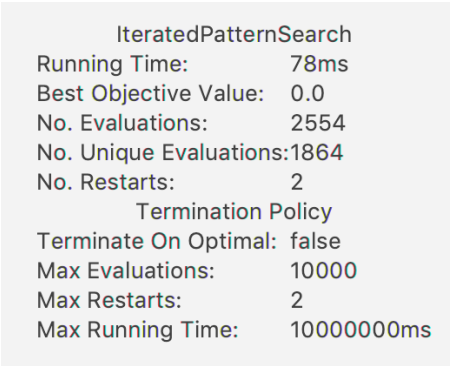

Figure 19: Static header reporting area

The other reporting area is live and updates with the progress of animation. It shows the current data point being considered and which variable these values belong to. It also shows the current the state of animation. See figure 20.

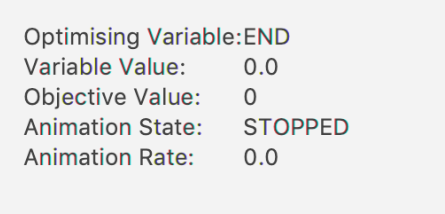

Figure 20: Live reporting area

5.5.4 Cursor Icon and Tooltips

The cursor icon changes when over different elements of the GUI giving the user hints about the functionality of that element. This aims to meets the intuitive GUI requirements. The icon is default over the empty space and reporting sections. When over any of the control buttons, legend icons or zoom sliders, it changes to a 'pointing hand' to tell the user they are clickable. When the graph zoom level is at default the icon is also the default. As soon as any zooming occurs, it changes to an 'open hand' to tell the user the graph is pannable. Moussing over any of the graph's data points
changes the cursor to a cross hair telling the user they can inspect the point. If the mouse hovers over a data point, a 'tooltip' displaying the points data will be displayed. Figure 21 illustrates how tooltips meet the precise data inspection requirements.

Figure 21: Tooltip shown on inspection of a data point on the graph

6. Results, Discussion and Evaluation

This chapter summarises the achievements of the project and the aims it failed to achieve. It also outlines suggestions of possible further work in and around the project scope.

This project has a fairly high success level in terms of design and implementation. The backend and software architecture designs were comprehensively built. They follow project requirements well and on the whole fully achieve project aims. It is lightweight and has good potential to be merged with and contributed to the original codebase in an opensource workflow.

Solid progress with visualisation and animation was made leaving the current implementation robust for normal use. The user can easily follow the algorithmic progress observing the line graph animations and live reporting elements. There is scope for more achievement, so this

requirement is considered partially achieved. Not every aspect of the AVM Algorithm was visualised and animated. For example, the wrap round the vector at the end of a run was left out of current animation sequence for time constraints on designing a solution for it. This is only an issue if in the AVMf termination policy `terminateOnOptimal` is set to false, as the algorithm terminates before the wrap round. The validity of the visualisation is not in question unless the objective value of one variable is dependent on the value of another. Better testing approaches would have discovered this issue earlier on. This section of the data currently is detected and left out of the graphs when series are constructed.

AVM restarts are correctly recorded as part of the JSON, but they are crudely handled. Only data for a single restart can be animated by the visualiser. This is currently hard coded to show only the first restart.

There is very limited support for the visualisation of vectors representing strings using the AVMf class `StringVariable`. These vectors are variable length and at present the visualiser can only handle vectors of constant length. A JSON file containing vectors of varying length can be imported into the visualiser but has unintended results. Seems to work reasonably well with a few errors but only if the vector is initialised with more variables than in its final optimised state. Unpredictable results are displayed if the vector is initialised with fewer variables. These visualisations are incorrect representations of the AVM progress along a fitness landscape

Animation requirements could only be considered fulfilled if visualisation support for every aspect of the AVM algorithm and every AVMf configuration was developed.

Inspection requirements are met using graph logarithmic zooming and panning to solve the issues of vast value ranges. Tooltips allow the user to see the values of every data point and there is powerful animation control to jump to particular sequence parts of interest.

This project suffered from time and project management issues; a challenge observed in the introduction. Time and effort resource estimations were heuristically thinking about a factor of 3 too low. Some valuable experience and knowledge has been gained in this area and the skill improved.

Fully contributing to original AVMf opensource project by pull request to original repository on GitHub did not happen. The visualiser code is not refined, robust and full featured enough to finish this process. It needs to be subjected to comprehensive unit and manual testing to be ready. Only manual testing has been done to date to gauge whether the visualiser works as expected.

There is some possible further work in some new areas opened up by this project. Using and documenting the visualiser tools application in a real SBSE project solution that uses the AVMf has benefits in evaluating its usefulness. It could help extend requirements.
A comprehensive user manual accompanying detailed documentation could also be created for this visualiser. Possibly incorporating a video documenting the animations and use tutorial.

There is also further possible work around features that have not yet made it into the implementation. Some of these are original requirements or designs that remain unmet. Others are new ideas for features that presented themselves along the during the project but were not incorporated due to time constraints.

The data recording harness could be improved to output JSON files in an appropriately named directory rather than the root. This would improve structure and tidiness.

Restarts could be properly handled in two possible ways. A basic approach that uses a argument (to a method or command line) specifying which restart to display in the GUI. Reporting in the GUI would be updated to show this number. A more advanced approach could adapt the GUI to show multiple restarts, possible through the use of tabs or pages. Extra statistics could be reported for each restart for comparison. For example, are the local solutions found the same?

More real-time statistics could be added to go along with animation such as:

- What the current best solution is.
- How many solutions have been considered so far.
- How many objective function evaluations have been made etc…

More precise animation controls could be added such as:

- A slider element to allow scrubbing through the animation sequence
- Buttons to move the animation from one variable to the next or previous, allowing pinpoint accuracy for inspection the user.

A menu system could be added to the GUI, showing options for loading a different file without having to restart the program. A help section could also be added, with an educational guide giving background information about the AVM and its paradigm.

Full support for visualising string vectors of variable lengths could be built in. Animation support for all aspects of the AVM and AVMf configurations could be implemented.

Code could be refactored, renaming classes, objects, variables and methods to be more semantically relevant be more self-documenting. Logic could also be refactored and made more modular as the GUI class is currently over 1000 lines of code. Comprehensive Javadoc coverage could be added, and unit testing implemented to test robustness.

If many of these changes were made the project would much better achieve the requirements set and a pull request could be made marking the main end objective fulfilled.

7. Conclusions

This report introduced a well-defined software engineering style design and build project in the field of search-based software engineering. SBSE uses local search-based optimisation algorithms to solve complex problems in reasonable amounts of time. The main project aim was to design and build a visualiser tool that animates the perceived real-time progress of such an algorithm called the Alternating Variable Method (AVM). The scope of implementation was constrained within an opensource Java library the AVMf, and so an opensource engineering workflow was used with the intent of creating a pull request to the original repository.

The literature Survey and Background analysis discussed the AVM and AVMf, their history, purposes and any recent developments/improvements to them. Various technologies were assessed for their suitability in this project. The JavaFX and GSON libraries in particular were looked at in detail and the conclusions lead to their inclusion in the project. The discipline of an opensource style development workflow was also examined with the intent of this inspiring and informing the workflow adopted in the project.

The Requirements and Analysis chapter broke down the main requirements and constraints of this project. There was analysis on opensource workflow and contribution and how this

informed decisions to keep all produced work within the Java paradigm. Problems facing visualisation and animation were considered and how to integrate the visualiser into the existing AVMf codebase.

The design chapter was chiefly concerned with giving an account of the software architecture design process and justifying why one choice was made over another. There is a detailed discussion of software architecture design concluding an offline design (where visualisation is detached from the AVM algorithm) is overwhelmingly advantageous over an online solution. The proposed offline architecture solution has three main modules. A data recording harness, which is the only part of the whole system altering existing AVMf code. This extracts data for visualisation from the AVMf and outputs it to a JSON file. The middle module is this JSON file acting as a data interface. The third module being the visualiser itself. Animation design is also discussed in this section

The Implementation and testing chapter gives a detailed description of the implementation of the design laid out in the previous chapter. Extensive use of diagrams and screenshots is made to help show how design requirements were met and manual testing carried out. It gives an overview of new classes and a deep analysis of the modifications made to any existing AVMf classes. This documentation is important to what exactly has been changed and added. It is of particular use the maintainers of the AVMf. Instructions on how to launch the visualiser are given here. An account of entry points and file handling logic is also provided.

The Results, Discussion and Evaluation chapter gives an account of the success and achievements of this project relative to its requirements. It points out that there has been a high level of success but there are still many areas that fall short of project requirements. The conclusion is; requirements have been fully met in some areas and partially in others but there are no areas of complete failure except to get a pull request accepted by the maintainers of the AVMf. It outlines future work that could be done to get the project ready for a pull request and fully satisfy project aims by contributing to the AVMf opensource project.

Read on for References, a Glossary of terms used in the project and the Appendix.

Bibliography

[1] P. McMinn, “Animating Search-Based Optimisation Algorithms in the AVMf,” 2018. [Online]. Available: https://mcminn.io/projects/animating-avm/. [Accessed 28 04 2019].

[2] “AVMf – An Open Source Framework for the Alternating Variable Method.,” [Online]. Available: http://avmframework.org. [Accessed 27 April 2019].

[3] “The AVMf GitHub Repository,” [Online]. Available: https://github.com/AVMf/avmf. [Accessed 30 04 2019].

[4] M. Harman and P. McMinn, “A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation,” 2007.

[5] P. McMinn and G. Kapfhammer, “AVMf: An Open Source Framework and Implementation of the Alternating Variable Method,” 2016.

[6] J. Kempka, P. McMinn and D. Sudholt, “A theoretical runtime and empirical analysis of different alternating variable searches for search-based testing.,” 2013.

[7] J. Kempka, P. McMinn and D. Sudholt, “Design and analysis of different alternating variable searches for search-based software testing.,” 2015.

[8] B. Korel, “Automated software test data generation.,” 1990.

[9] The Apache Software Foundation, “Welcome to Apache Maven,” The Apache Software Foundation, [Online]. Available: https://maven.apache.org. [Accessed 01 05 2019].

[10] S. Robinson, “What is Maven?,” Stack Abuse, 02 09 2013. [Online]. Available: https://stackabuse.com/what-is-maven/. [Accessed 01 05 2019].

[11] The Apache Software Foundation, “Intoduction to Build Lifecycle,” The Apache Software Foundation, 30 04 2019. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. [Accessed 30 04 2019].

[12] Oracle, “Java Client Roadmap Update,” 2018. [Online]. Available: https://www.oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf. [Accessed 01 12 2018].

[13] Oracle, “JavaFX Frequently Asked Questions,” [Online]. Available: https://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6 . [Accessed 01 12 2018].

[14] “OpenJFX wiki pages,” [Online]. Available: https://wiki.openjdk.java.net/display/OpenJFX/Main . [Accessed 01 12 2018].

[15] “The OpenJDK wiki pages,” [Online]. Available: https://openjdk.java.net. [Accessed 01 12 2018].

[16] G. Motroc, “JavaFX 11: What's changed, and what's stayed the same,” Jaxenter, 2018. [Online]. Available: https://jaxenter.com/javafx-interview-series-part-2-149921.html [. [Accessed 01 12 2018].

[17] J. Vos, “JavaFX as a seperate module: a look back and a leap forward.,” Jaxenter, 2018. [Online]. Available: https://jaxenter.com/javafx-separate-module-jdk-142253.html. [Accessed 01 12 2018].

[18] The JUnit Team, “JUnit,” The JUnit Team, 2019. [Online]. Available: https://junit.org/junit5/. [Accessed 01 05 2019].

[19] “OpenJFX open-source contribution log,” [Online]. Available: http://hg.openjdk.java.net/openjfx/jfx-dev/rt/. [Accessed 01 12 2018].

[20] G. Wielenga, “Developing NASA's misson software with Java,” Jaxenter, 2014. [Online]. Available: https://jaxenter.com/netbeans/developing-nasas-mission-software-with-java . [Accessed 01 12 2018].

[21] Google, “GSON GitHub Repository.,” Google, [Online]. Available: https://github.com/google/gson. [Accessed 30 04 2019].

[22] J. Dreyfuss, “he Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP,” OverOps, 2015 28 2015. [Online]. Available: https://blog.overops.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/. [Accessed 01 05 2019].

[23] Apache, “GSON Apache Licence 2.0,” Google, 2004. [Online]. Available: https://github.com/google/gson/blob/master/LICENSE. [Accessed 30 04 2019].

[24] R. Allen, “the beginners guide to contributing to a github project,” 22 09 2015. [Online]. Available: https://akrabat.com/the-beginners-guide-to-contributing-to-a-github-project/. [Accessed 29 04 2019].

[25] “Forking Projects,” GitHub, 30 11 2017. [Online]. Available: https://guides.github.com/activities/forking/. [Accessed 30 04 2019].

[26] V. Driessen, “A successful Git branching model,” 05 01 2010. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/. [Accessed 30 04 2019].

Glossary

SBO - Search based optimisation

SBSE – Search-Based Software Engineering

AVM – Alternating Variable Method. An effective hill climbing local search-based optimisation algorithm.

AVMf – AVM Framework. An open source Java library for the AVM algorithm an [2]

JSON – JavaScript Object Notation

GSON - A Java library that can be used to convert Java Objects into their JSON representation

GUI – Graphical User Interface

Online Software Architecture – In this project used to mean visualisation integrated into the mechanics of the algorithm.

Offline Software Architecture – In this project used to mean visualisation detached from the mechanics of the algorithm.

Objective Function – Synonymous with Fitness Function. a function that evaluates a vector and gives it a numeric score representing its fitness.

Fitness Landscape – The surface of a fitness function

Appendix

The project code is available in its forked GitHub repository at:
https://github.com/benscott2096/avmf

```
// Method takes in a value between x axis upper and lower bounds (value used
from slider) and returns its logarithmic zoom equivalent
private double xCalcLogValue(double value){
    double logValue = xALog * Math.exp(xBLog *value);
    return logValue;
}

// Method to calculate and store values needed for logarithmic x axis zoom
calculations
private void xSetupLogCalc(double x, double y){
    xBLog = Math.log(y/x)/(y-x);
    xALog = y / Math.exp(xBLog * y);
}

// Method takes in a value between y axis upper and lower bounds (value used
from slider) and returns its logarithmic zoom equivalent
private double yCalcLogValue(double value){
    double logValue = yALog * Math.exp(yBLog *value);
    return logValue;
}

// Method to calculate and store values needed for logarithmic y axis zoom
calculations
private void ySetupLogCalc(double x, double y){
    yBLog = Math.log(y/x)/(y-x);
    yALog = y / Math.exp(yBLog * y);
}
```

Figure 22: Methods for converting zoom slider values into logarithmic values: