



Faculty of Engineering and Technology
Master of Software Engineering (SWEN)
Detailed Proposal

Test Cases Prioritization for Component-Based Front End Technologies

تحديد الأولويات لحالات فحص منتج برمجي يستخدم تقنية واجهة مستخدم مبنية من خلال عمارة المكونات

By

Student Name: Hiba Ghannam

Student Number: 1165305

Supervised By:

Dr. Abdel Salam Sayyad

Dr. Sobhi Ahmad

**A thesis submitted in fulfillment of the requirements for the
degree of Master of Science in Software Engineering at Birzeit University, Palestine**

October 23, 2020

Approved by the thesis committee:

Dr. Abdel Salam Sayyad, Birzeit University

Dr. Sobhi Ahmad, Birzeit University

Dr. Samer Zein, Birzeit University

Dr. Majdi Mafarja, Birzeit University

Date approved:

Declaration of Authorship

I, Hiba Ghannam, declare that this thesis titled, “Test Cases Prioritization for Component-Based Front End Technologies Web Application” and the work presented in it are my own. I’m confirming the following:

- This work was done wholly or mainly while in candidature for a master degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

Test cases prioritization is an important action that has to be done during the testing phase within the software development life cycle. It helps to add more focus on test cases that have high priority. In addition, it helps to discover defects in early stages therefore cost and time will be managed in an effective way. Recently, for building web applications, component based architecture frontend frameworks are the most popular used technologies. Therefore new test cases prioritization could be connected with frontend components depending on components reusability and their business requirements. In previous research, several solutions were proposed for test cases prioritization. The gap here, that most of these solutions were built for regression testing. Few proposed solutions have been generated for new test cases, while these solutions are not considering the new frontend technologies like React or Angular.

This research presents a framework as an automated solution for prioritization of new test cases. Where those test cases represent a web application that is going to be developed using component based architecture frontend frameworks. The prioritization problem has been considered as a multi objective optimization problem where trade-off has to be done between different objectives. Therefore, the proposed solution in this research uses four genetic algorithms: NSGA-II, IBEA, MOCell and SPEA2. During this research, five datasets have been created since there are no available datasets. First one has been created manually, while the others have been created using a random approach. The randomly generated datasets have been created to generate different dataset sizes and this has provided an opportunity to study the impact of dataset size on results. This proposed random approach for creating datasets, can help researchers to create any dataset with any required size for testing any similar problem.

Several experiments have been done during this research and using the five datasets. Results for all datasets approved that 30 seconds as minimum execution time is enough to all mentioned algorithms. In addition the quality is close to all algorithms. The results also approved that having limited time for testing generates a high quality solution in less than 30 seconds as execution time for any mentioned algorithm. On the other hand, more available time for testing leads to a more complex problem that reduces the solutions quality.

المخلص

يعد تحديد أولويات حالات الاختبار إجراءً مهمًا يجب القيام به أثناء مرحلة الاختبار ضمن دورة حياة تطوير البرامج. يساعد ذلك على إضافة المزيد من التركيز على حالات الاختبار التي لها أولوية عالية. بالإضافة إلى ذلك ، فإنه يساعد على اكتشاف العيوب في المراحل المبكرة ، وبالتالي ستنم إدارة التكلفة والوقت بطريقة فعالة. في الأونة الأخيرة ، لبناء تطبيقات الويب ، تعد أطر الواجهة الأمامية للبنية القائمة على المكونات من أكثر التقنيات المستخدمة. لذلك يمكن ربط أولويات حالات الاختبار الجديدة بمكونات الواجهة الأمامية اعتمادًا على إمكانية إعادة استخدام المكونات ومتطلبات أعمالها. في الأبحاث السابقة ، تم اقتراح العديد من الحلول لتحديد أولويات حالات الاختبار. الفجوة هنا ، أن معظم هذه الحلول تم بناؤها لنوع من أنواع الاختبار يسمى الانحدار أو التراجع regression. تم إنشاء عدد قليل من الحلول المقترحة لحالات الاختبار الجديدة ، في حين أن هذه الحلول لا تأخذ في عين الاعتبار تقنيات الواجهة الأمامية الجديدة مثل React أو Angular.

قدم هذا البحث إطار عمل كحل آلي لتحديد أولويات حالات الاختبار الجديدة. حيث تمثل حالات الاختبار هذه تطبيق ويب سيتم تطويره باستخدام أطر عمل الواجهة الأمامية للبنية القائمة على المكونات. تم اعتبار مشكلة تحديد الأولويات على أنها مشكلة متعددة الاهداف حيث يجب إجراء المفاضلة بين الاهداف المختلفة. لذلك ، تم تطوير الحل المقترح باستخدام أربع خوارزميات جينية: NSGA-II و IBEA و MOCeII و SPEA2. خلال هذا البحث ، تم إنشاء خمس مجموعات بيانات نظرًا لعدم توفر مجموعات بيانات. تم إنشاء المجموعة الأولى يدويًا ، بينما تم إنشاء المجموعات الأخرى باستخدام نهج عشوائي. تم إنشاء مجموعات البيانات بالطريقة العشوائية للحصول على مجموعات بيانات بأحجام مختلفة ، وقد أتاح ذلك فرصة لدراسة تأثير حجم مجموعة البيانات على النتائج. يمكن أن يساعد هذا النهج العشوائي المقترح لإنشاء مجموعات البيانات الباحثين الآخرين على إنشاء أي مجموعة بيانات بأي حجم مطلوب لاختبار أي مشكلة مماثلة.

تم إجراء العديد من التجارب خلال هذا البحث وباستخدام مجموعات البيانات الخمس. تم الحصول على نتائج متقاربة لجميع مجموعات البيانات فقد كانت 30 ثانية كحد أدنى لوقت التنفيذ كافية لجميع الخوارزميات المذكورة. بالإضافة إلى أن الجودة متقاربة لجميع الخوارزميات التي تم تطبيقها على جميع مجموعات البيانات المذكورة في البحث. وأقرت النتائج أيضًا أن وجود وقت محدود للاختبار سيؤدي إلى حل عالي الجودة في أقل من 30 ثانية كوقت تنفيذ لأي خوارزمية مذكورة. من ناحية أخرى ، يؤدي المزيد من الوقت المتاح للاختبار إلى مشكلة أكثر تعقيدًا تقلل من جودة الحلول.

Acknowledgements

I would like to express my special thanks to my supervisor Dr.Abdelsalam Sayyad for all orientations and special guidance in this research field. I would also like to thank Dr.Yousef Hassouneh for the important courses, specifically design and architecture that I returned to during this research several times. Also I want to thank Dr.Samer Zain for the amazing mobile course, in that time I started to think about frontend technologies and that definitely helped me to discover the mentioned gap in this research thesis. Thanks also for all my instructors in the previous amazing experience.

Table of Contents

1. Introduction	1
1.1 Motivation	2
1.2 Problem Statement	3
1.3 Research Objectives	4
1.4 Research Questions	4
1.5 Research Contribution	5
2. Background	7
2.1 Test cases effectiveness and prioritization	7
2.2 Component Based Architecture Front End web applications	9
2.3 Multi Objective Optimization Problem and Solutions	12
2.3.1 Evolutionary Algorithms for Multi Objective Optimization	14
2.3.2 Genetic Algorithm (GA)	15
2.3.3 Nondominated Sorting Genetic Algorithm (NSGA and NSGA-II)	17
2.3.4 Strength pareto evolutionary algorithm (SPEA2)	19
2.3.5 Multi Objective Cellular Genetic algorithm (MOCeII)	20
2.3.6 Indicator- Based Evolutionary Algorithm (IBEA)	21
2.3.7 Solutions Evaluations by Hyper Volume	21
2.4 Optimization for Testing	22
3. Related Work	24
3.1 Prioritization Solutions for regression test cases	24
3.1.1 Structural Coverage Based Solutions	24
3.1.2 Fault Detection Based Solutions	26
3.1.3 History Based Solutions	28
3.1.4 Requirements Based Solution	30
3.1.5 Hybrid Approaches	33
3.2 Prioritization Techniques for New Test Cases	36
3.3 Related Work and the Research Gap Summary	38
4. Research Methodology	45
4.1 Facebook Dataset	45
4.2 Solution Design	51
4.2.1 Chromosome Representation	54
4.2.2 Objective Functions	55
4.3 Random Datasets	58
4.4 Development Environment	61
5. Experiment Setup and Run	62
5.1 General Design Structure	62
5.2 Problem definition	64
5.3 Algorithms Runners	70
5.4 Find Violation Algorithm	75
6. Experiment Results and Analysis	80
6.1 Finding the Minimum Execution Time	81
6.2 Available Time Impact on HV value	96

6.3 Impact of remaining time on Algorithm Execution time	102
6.4 Hypervolume (HV) Comparison	103
7. Threats to Validity and Conclusion	110
7.1 Threats to Validity	110
7.2 Conclusion	112
7.3 Future Work	113
Appendix A	121

List of Figures

Figure 2.1	Model View Controller (MVC) Architecture
Figure 2.2	Components reusability in component based frontend frameworks
Figure 2.3	Dominance Example
Figure 2.4	Pareto Front Example
Figure 2.5	General Procedure for finding solution in Multi Objective Optimization
Figure 2.6	Basic pseudocode for the genetic algorithms
Figure 2.7	Pseudo Code for MOCeII algorithm
Figure 4.1	Dataset Sample represents test cases and their properties
Figure 4.2	Facebook Component - New Post View
Figure 4.3	Facebook Component - Write Post
Figure 4.4	Facebook Component - View Post
Figure 4.5	High Priority Test Case Properties sample includes dependencies
Figure 4.6	Simple High Priority Test Case Properties sample that does not include dependencies
Figure 4.7	Phase 1 objectives and solution result
Figure 4.8	Chromosome Structure Representation
Figure 4.9	Sample from a random dataset with 400 test cases
Figure 4.10	Sample from a random dataset with 1000 test cases
Figure 4.11	Generating the random values using random function
Figure 5.1	TCP problem definition class
Figure 5.2	Algorithms Runners' Classes
Figure 5.3	Setting the number of variables and objectives
Figure 5.4	Dataset sample from the extracted input file
Figure 5.5	Signature of Read Problem method in TCP problem
Figure 5.6	Signature of Evaluate method in TCP problem
Figure 5.7	Evaluate Algorithm Flow
Figure 5.8	Solutions from several generations with single Run
Figure 5.9	Four solutions were resulted and printed from running an algorithm
Figure 5.10	Pareto Front for the generated solution in Figure 5.9
Figure 5.11	Summarization of Runners Classes Steps
Figure 5.12	Test cases dependencies sample
Figure 5.13	Test Case with three dependencies components from the original dataset
Figure 5.14	Six solutions from random run without marking violations
Figure 5.15	Resulted Solution with Violations
Figure 5.16	Test case doesn't have violation in another solution

Figure 6.1	SPEA2 implementation with time as a stopping condition
Figure 6.2	HV over time for NSGA-II with fcbk163-dataset
Figure 6.3	HV over time for NSGA-II with 400-dataset
Figure 6.4	HV over time for NSGA-II with 600-dataset
Figure 6.5	HV over time for NSGA-II with 800-dataset
Figure 6.6	HV over time for NSGA-II with 1000-dataset
Figure 6.7	HV over time for IBEA with fcbk163-dataset
Figure 6.8	HV over time for IBEA with 400-dataset
Figure 6.9	HV over time for IBEA with 600-dataset
Figure 6.10	HV over time for IBEA with 800-dataset
Figure 6.11	HV over time for IBEA with 1000-dataset
Figure 6.12	HV over time for MOCell with fcbk163-dataset
Figure 6.13	HV over time for MOCell with 400-dataset
Figure 6.14	HV over time for MOCell with 600-dataset
Figure 6.15	HV over time for MOCell with 600-dataset
Figure 6.16	HV over time for MOCell with 1000-dataset
Figure 6.17	HV over time for SPEA2 with fcbk163-dataset
Figure 6.18	HV over time for SPEA2 with 400-dataset
Figure 6.19	HV over time for SPEA2 with 600-dataset
Figure 6.20	HV over time for SPEA2 with 800-dataset
Figure 6.21	HV over time for SPEA2 with 1000-dataset
Figure 6.22	Assign the available time in problem definition
Figure 6.23	HV boxplot for small value of available time with small dataset size
Figure 6.24	HV boxplot for large value of available time with small dataset size
Figure 6.25	HV boxplot for small value of available time with large dataset size
Figure 6.26	HV boxplot for large value of available time with large dataset size
Figure 6.27	HV quality indicator for fcbk163-dataset
Figure 6.28	HV quality indicator for 400-dataset
Figure 6.29	HV quality indicator for 600-dataset
Figure 6.30	HV quality indicator for 800-dataset
Figure 6.31	HV quality indicator for 1000-dataset

List of Tables

Table 3.1	Summary of prioritization methods and the possibility of using them for this research
Table 5.1	Initial Runs Setup
Table 6.1	Datasets list and their test cases
Table 6.2	HV over time for NSGA-II with all the research datasets
Table 6.3	HV over Time for IBEA with all the research datasets
Table 6.4	HV over Time for MOCell with all the research datasets
Table 6.5	HV over Time for SPEA2 with all the research datasets
Table 6.6	HV mean for different values of available time
Table 6.7	Captured HV when the remaining time is a small value
Table 6.8	HV mean for 30 independent runs
Table 6.9	HV median for 30 independent runs
Table 6.10	Max and min value for each algorithm with all datasets

Chapter 1. Introduction

Software development life cycle has many stages, it starts from requirements gathering, design, development, testing and it ends up with deployment [2]. Testing is defined as a process which is used to discover the potential faults in software products [4]. The most important value for this phase is finding problems and fixing them before releasing the software to production. Early discovery for problems reduces the side effects before these problems are being larger [64]. For testing execution, test cases have to be prepared by software testers by selecting different data inputs under different conditions of the program. Each test case has an expected result which could be specified depending on the requirements [6]. In most cases there is a limited time for testing, so it's important to specify the test cases priorities that help to specify the order of test cases in the execution stage. This action helps to increase the coverage and to discover faults in early stages [1]. Different types of testing are available, for each type of testing it's important to know how the coverage of testing will be [66]. For example several coverage types are available to regression testing. Regression targets to verify other parts of the system when adding a new change[11]. On the other hand, not all types are supported with blackbox testing or with adding a new feature.

In this research a specific prioritization solution is proposed for a specific test cases prioritization problem that targets test cases prioritization for web application. Those web applications have to use a component based architecture technology for building the front end side. These technologies have an impact on test cases priorities and no previous solutions considered this impact. The test cases prioritization problem itself is based on requirements coverage and the solution has been built using search based software engineering. Search based software engineering (SBSE) term was first proposed by Harman in 2001 [52]. It refers to applying search based algorithms to solve problems and gaps in software engineering. For software engineering problems that have multiple objectives and when a tradeoff is required, the test cases prioritization problem is considered as a search based optimization. On the other

hand, optimization refers to finding a solution depending on the required objectives. In previous research, software optimization problems have been applied in different fields of software engineering like verification, project planning, cost estimation [54]. 59% of published research was done on software testing field testing [53].

Several solutions have been built in previous research for test cases prioritization, most of these solutions were created based on multi objective optimization algorithms. The reason behind considering the prioritization process as a multi objective optimization problem is the need for a trade-off several times between different objectives of test cases prioritization. Most of the proposed solutions have been created for doing prioritization of regression test cases that have historical data [9]. Therefore, prioritization for new test cases is a different problem.

This chapter illustrates the research motivation in section one. Then Problem statement in details in section two, it's generally about test cases prioritization for web applications that are going to be built using component based architecture front end frameworks applications. After that, three objectives are presented in section three and two research questions are specified in section four. Then the research contribution is presented in section five.

1.1 Motivation

Several frontend frameworks have been generated recently based on component based architecture. React and Angular are examples of these frameworks that are widely used for building web applications. Considering the main advantage of having reusable components may affect the test cases priorities. If the test case is connected with a reusable component that is required for a main business flow, then for sure it will have high priority. Any defects that might be discovered with that component will affect several test cases, it may block the testing process for a while. Hence re-prioritization might be required within the testing phase several times. In addition, for better requirements coverage and early discovery of defects, considering the frontend framework is important from the beginning.

In previous research, most available prioritization solutions were created to support regression testing [9]. While in this research problem, the test cases are going

to be created for the first time. These test cases are also connected with specific web applications technologies, hence, manual testing is the target. This means, using the available solutions that require code access or previous test cases history is not applicable. In addition, trying to apply any available solution from previous research to this test cases prioritization problem, will not consider the new frontend frameworks technologies. Therefore, this research was motivated by the lack of solutions for manual test cases prioritization that are going to be built using new frontend frameworks. Where these frameworks are based on component based architecture and there is a lack of research about them.

The test cases prioritization process has to increase the requirements coverage and it has to focus on high priority test cases. Trade-off is required here to choose between the two mentioned objectives. Therefore, the test cases prioritization problem is considered as multi objective optimization one and genetic algorithms could be used to generate the solutions for those kinds of problems..

1.2 Problem Statement

Different requirements changes are expected to be requested from the customers during the development cycle. This happens frequently when agile methodologies are being used. In this situation and with the large scale software applications, a long list of test cases is needed to be prioritized for multiple times. In modern web applications, component based architecture front end technologies are frequently used [48]. Different test cases should be designed to test micro small components, other test cases have to be designed for testing the complex flows. Any change on components or requirements priorities needs to update test cases priorities in order to increase the requirements coverage. When the feature of software under development is a new one, then there is no historical data about test cases or bugs [25]. This means no previous bugs that have severities or priorities that can be used in the new test cases prioritization [36]. Therefore, previous prioritization methods that are based on structural coverage or historical data are not useful for increasing the requirements coverage. In addition, even the component architecture itself is not new

[10], previous prioritization solutions are not considering the new front end technologies architecture.

For the mentioned gap, the test case itself may validate a single functionality by using only one component. With other more complex functionality, other test cases may validate a business requirement that needs to validate several components at the same flow. Some components are being reused more than others. That means, validation for complex test cases that have high priorities consumes more time. Therefore, increasing the requirements coverage for high priority test cases may reduce the requirements coverage in general. As a result, for this research test cases prioritization problem there is a kind of tradeoff that has to be done by decision makers to decide where they have to focus within the available testing time. Hence, the research problem is considered a multi objective optimization problem. Where the solution has to increase the requirements coverage for the high priority test cases and the total coverage at the same time.

1.3 Research Objectives

Two research objectives have been studied in this research for the mentioned test cases prioritization problem in the previous section:

1. Increasing the business requirements coverage for the new generated manual test cases with high priority.
2. Increasing the total test cases coverage for business requirements regardless the priority, within a specific available time for testing.

1.4 Research Questions

In order to achieve the previous objectives under the same condition for having component based architecture front end technologies, the research have to answer the following questions:

- RQ1: How to maximize the high priority test cases coverage for business requirements of new test cases?
- RQ2: How to maximize the total coverage for business requirements of new test cases?

1.5 Research Contribution

The research contributes basically by finding a prioritization solution for long lists of test cases and restricted or limited time. That has to work with any software development life cycle, regardless it's agile, waterfall or any other methodology. On the other hand, the research doesn't target the short time releases like one or two weeks. These short releases have a short list of test cases per a specific resource and without any need for changes. The prioritization solution also doesn't target applications that are not going to be built using frontend frameworks that are not component based. In addition, the target is not regression test cases because there are a lot of already existing solutions for regression. Even though the designed prioritization solution can work with regression testing. Implementing the prioritization solution in this research for the targeted application and environment has the following contributions:

1. Applying the existing genetic algorithms to find a solution for a new multi objective optimization problem. The jMetal framework has been used to implement a prioritization solution for new generated manual test cases.
2. Automated framework solution for new test cases prioritization of web applications that are going to be developed using Angular, React or any other component based architecture framework. The targeted test cases are the new test cases that were designed for new features or new software. This means these test cases are manual ones. At the same time, the targeted test cases prioritization problem of this research is a web application that is going to be built using component - based front end technologies.
3. The generated solution will be used to help decision makers like managers and product managers to have the best investing of the time under different circumstances. This could be done by different kinds of re-prioritization several times.
4. Generating several datasets for component based - front end technologies and finding a simple way to create datasets with any size for testing. Even the component based architecture is not new, but the front end technologies that have this architecture are new. As a result, there is no dataset that could be

used in the experiment, therefore a new one is needed to be created. A new dataset with a real application was generated for Facebook web app. Where Facebook was built using React, as one of the highly used front end technologies [57]. Additional datasets were created randomly with the main required fields for the experiments to validate the database size impact.

Chapter 2. Background

In this chapter, a background about the research topic has been presented. First section illustrates the effectiveness of the test case and the relation with prioritization. Section 2.2, presents component based architecture frameworks that this research test cases prioritization problem has been built on. While the next section illustrates the definition of multi objective optimization, problems and solutions. This includes the required algorithms that have been used for this research solution. The last section presents the optimization in software testing.

2.1 Test cases effectiveness and prioritization

Effectiveness of test cases could be measured by considering the number of bugs or defects that were discovered by executing the test case [14]. Once the number increases, the effectiveness of the test case increases. This could be considered as an indication that helps in discovering more failures and bugs, as an important result it helps in resolving them by executing the same test case or scenario. To improve test case effectiveness different tips were suggested from research. One important tip is understanding and making a kind of documentation for test cases. This action helps in adding more focus on analysis and reviewing and it helps to discover failures before execution starts [15]. Another important tip is thinking about the factors that may affect the test cases and different sources of errors and defects.

Different metrics are connected with test cases effectiveness and helps in detecting new bugs and defects, these metrics are called coverage metrics. The coverage gives the team an indication about their quality by the number of items that were covered in their testing. Definitely this helps in discovering some gaps or new areas that were not covered and that should be tested by the team. As a result, new bugs will be discovered and then early fixes will be done. This reduces the cost of bugs fixes instead of discovering them on production later, more trust and confidence about the quality could be achieved [18].

The focus of most research papers was done by coverage metrics that are directly connected with code, this is called structured or code coverage [18]. For example, statement coverage is used to verify that all statements in the code were covered in the testing phase. Other examples are conditions, branches and loops coverage [19]. All the mentioned types as noticed need to have the source code of the software and to follow that code. Another kind of test coverage which doesn't need the source code is requirements coverage [17], it is the best fit to work with black box testing [16]. In this case there is no need to know anything about internal code that should be built by the development team [7]. Another common name for black box testing is functional testing, where the tester needs to make sure that the software functions as expected [8]. Black box testing has main advantages, it's important for test cases design and execution because there is no need for code at all. In addition, it's easy to learn since there is no need for learning the programming language. On the other hand, it has clear limitations with code and path coverage.

Usually the testing team put an effort to order test cases depending on some criteria, this is called test cases prioritization. The target of prioritization is to increase the percent of error detection and failures in early stages. In addition, it's important for discovering the most important bugs and risks early [20]. This action definitely increases the effectiveness of testing in general. Start testing with test cases that have high coverage and error detection early, then missed areas, gaps and different kinds of errors will be detected early [21]. As a result of early discovery, this gives a better chance for debugging and fixing bugs and finding solutions, for sure, fixing cost will be reduced. Different techniques are available and were studied by researchers for test cases prioritization. Most of these techniques are applicable for regression testing since it consumes a lot of time and in a repeated way [22]. On the other hand, some studies focused on prioritization solutions for white box testing. In white box testing as mentioned before, the code must be available for testers and technical knowledge is needed. Less studies focused on manual testing and black box prioritization. More about all of these available solutions is discussed in the related work. Regardless of the testing type or even the coverage, in some cases the decision makers decide to release the software with some available known bugs or available areas without deep testing [65].

2.2 Component Based Architecture Front End web applications

Web application is generally considered as a client-server application that uses the browser as a client [46]. Clients send requests to the server, requests will be processed by the server. Later, response will be returned to client or browser as representative of client in web applications. Client-Server architecture style itself could be considered as two layers [47]. The Web application server might communicate with other servers or layers to do specific processing. As a result, the web application could be considered as a multi tiers architecture. It applies three tiers architecture design pattern. The first tier is the presentation layer, it is used to display the user interface and it's represented by the web browser. The second layer is the business layer, it is represented in web application by the web server and it's used to process browser requests from business logic perspective. The third one is the database layer. For more separation between business logic and user interface, model-view-controller architecture design pattern could be applied as represented in Figure 2.1 [46]. View is used for display user interface on browsers by encapsulation of display choices. While the model is used for business information encapsulation. Third part, which is the controller, is used for communication and to maintain the consistency between model and view.

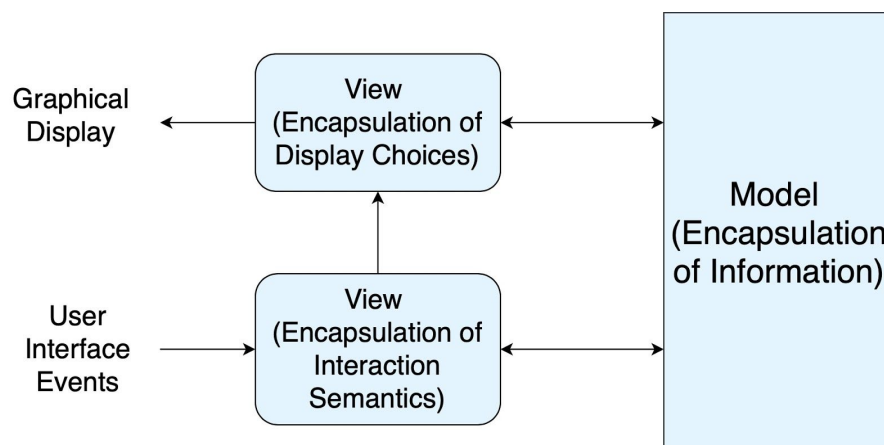


Figure 2.1 Model View Controller (MVC) Architecture

Development of the client side for the web application is known as frontend, while development of server side or business logic is known as backend. Different programming languages, frameworks and technologies are used for frontend and backend development. In modern web applications, the most web technology that is used for frontend development is Javascript [48][49]. Javascript is an interpreted language that provides object oriented capabilities [50]. It has different advantages such as performance, objects support, easy to learn and code reuse for both frontend and backend sides [51]. For different purposes for software developers such as performance and reusability, different frameworks and libraries were developed in recent years on top of Javascript. However, all these frameworks are working to support web applications development with model view controller (MVC) architecture.

Recently, for new large scale applications, there are different main and important challenges. A complex user interface that has a dataset which highly changes over time is one of them. In addition, high maintainability needs is a big challenge for owners and developers [57]. Justin Meyer who is the creator of JavaScriptMVC has a key solution for the mentioned challenges, it's represented by this quote "*The secret to building large apps is never build large apps. Break your applications into small pieces. Then, assemble those testable, bite-sized pieces into your big application*". The component based architecture is the best solution that represents that quote, therefore new solutions have to resolve problems separately. By component based architecture, the user interface is constructed by building different components. Components represent different business needs, each set of components are used together to perform full business flow. Raymond summarized the main concept about using the component based architecture in his book "The Art of Unix Programming" as the following: "*The only way to write complex software that won't fall on its face is to hold its global complexity down, to build it out of simple parts connected by well-defined interfaces, therefore most problems are local and you can have some hope of upgrading a part without breaking the whole.*"[57].

Angular2.0 and React are examples of the proposed frameworks as solutions that are highly used nowadays by developers. They are used for building complex

interfaces, scalable and maintainable web applications using MVC architecture. React is a JavaScript framework as described in some research, or it's a library as described in another research[59]. However, either it's a framework or a library, it was proposed by Facebook developers in 2013 [57] in order to resolve the mentioned problems. The needs specifically were raised when Facebook developers had a problem with managing the Ads that usually changes over the time. They also had a different need of keeping the syncing for user interface with business needs and application state [59]. One of the most proposed benefits of react and other component based technologies is reusability. So, building a new component may lead to breaking it into other smaller components and this will let you reuse all the micro components later. To illustrate this, an example from Facebook web app, comment is a totally separated component from comments list [59]. This means, when we see the whole comment UI we may think it's one component, but as described by Facebook developers [59], separated small components will give you the ability to reuse them everywhere in the system. In addition to reusability of small components, you also can reuse the large component itself which consists of smaller components. The result for you, is the ability to reuse everything. Figure 2.2 represents the components reusability in components based architecture, where a large component composes small or micro components and the same micro component is used in different components .

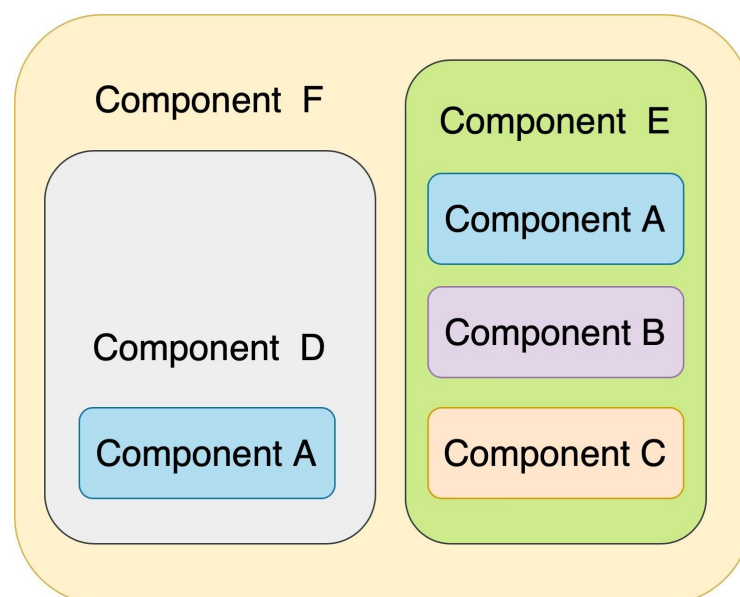


Figure 2.2 Components reusability in component based frontend frameworks

2.3 Multi Objective Optimization Problem and Solutions

Multi objective optimization problem refers to the case where different competing objectives are needed by decision makers. The process of finding solutions for those kinds of problems is called optimization [58]. By that solution, it is expected to find the optimal value for each objective with respecting all other objectives. The result will be a kind of tradeoff between all the competing or conflicting objectives. With single optimization the solution is evaluated by comparing the values of different functions, while in multi objective optimization, dominance is used [59]. When we have two solutions such as x_1 and x_2 , x_1 dominates x_2 when [57]:

- x_1 is not worse than x_2 for all objectives.
- x_1 is better than x_2 at least for one objective.

When the above two conditions are applied, we can say x_2 is dominated by x_1 . Figure 2.3 represents two competing functions or objectives, f_1 is maximizing while f_2 is minimizing [60]. There are several possible solutions where some of them are dominated by others. For example, solution 1 is better than solution 2 regarding f_2 because it has a value which is less than solution 2. Also solution 1 has maximum value than solution 2 if we compared them regarding f_1 . The result from the previous example is considering that solution 2 is dominated by solution 1. Same is applied to solution 5 and solution 1, even solution 5 and solution 1 are the same regarding f_2 , but solution 5 is better than 1 regarding f_1 . As a result, solution 5 dominates solution 1. Another important example from the diagram is comparison between solutions 1 and 4. Solution 1 is better regarding f_2 , while solution 4 is better regarding f_1 . Result: no one dominates the other.

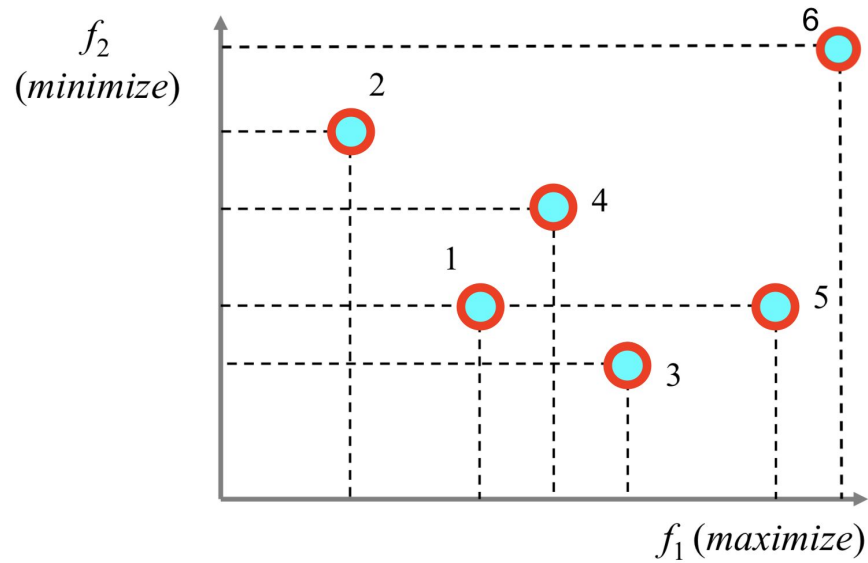


Figure 2.3 Dominance Example

The set of non dominated points of feasible decision space is called pareto optimal, while pareto front is the boundary. On the other hand, the set of non dominated solutions is called the solution set [60]. In Figure 2.4, we can see the non dominant points: 3, 5, 6 where their points are not dominated by any other points. The boundary is the pareto front [60].

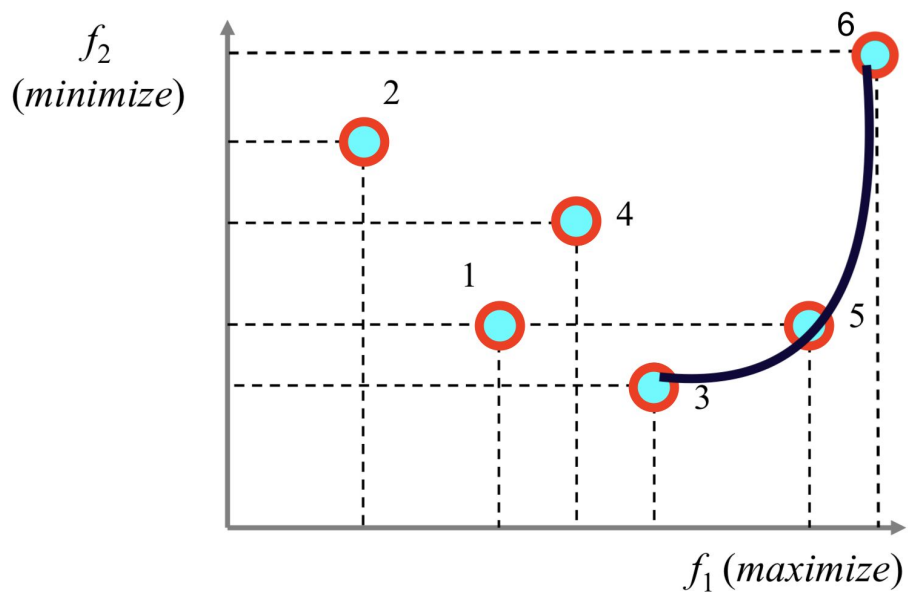


Figure 2.4 Pareto Front Example

Finding a diverse set of solutions that are close enough to the pareto optimal front, is the main target of multi objective optimization. To implement these solutions, evolutionary multi objective optimization (EMO) algorithms are used. After finding the pareto optimal, depending on a higher level of information a decision has to be taken about the required solution from the generated set of pareto optimal [60], Figure 2.5 represents the general procedure.

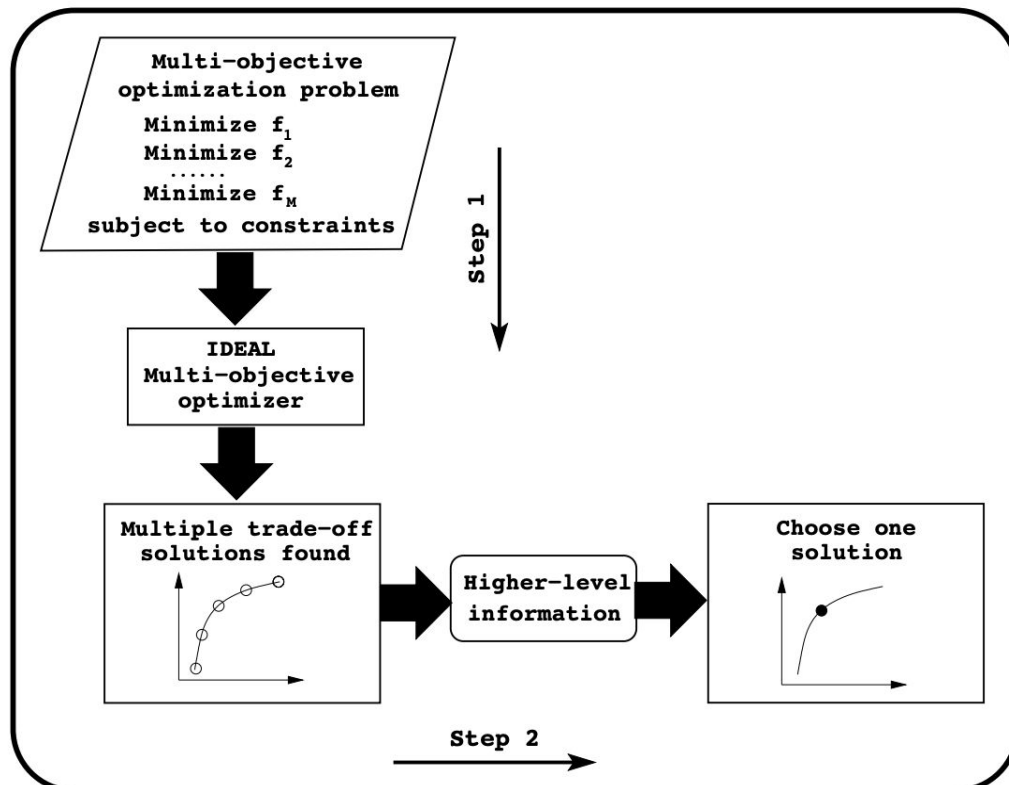


Figure 2.5 General Procedure for finding solution in multi objective optimization

2.3.1 Evolutionary Algorithms for Multi Objective Optimization

Evolutionary algorithms idea is based on natural evolution [61]. Two concepts are used from nature in these algorithms, the first one is selection and the second is variation [61]. Selection refers to the idea of competition between living beings on the available resources. This means some of these resources have a better chance to survive. The second concept refers to the ability of generating new living beings by principles of mutation and recombination. The importance of these algorithms in multiobjective optimization problems is finding more than one pareto optimal solution by a

single run. The whole idea of finding the solutions by EA has to be done by maintaining a set of solution candidates that were selected. After that, they have to be manipulated by recombination and mutation. Several methodologies were proposed under these algorithms, one of the most popular algorithms is the genetic algorithm. Under the genetic algorithm there is a variant group of algorithms [61].

2.3.2 Genetic Algorithm (GA)

Genetic algorithms were invented in the 1960s by John Holland in order to study how we can apply nature evolution and adaptation into computer science. After that genetic algorithms have been considered as most robust evolutionary algorithms. It was mainly inspired by the way species are being evolved by natural selection based on fittest individuals. The generated solution set by the genetic algorithm is called population, while each solution in the generated population is called chromosome or individual. A Single chromosome in nature contains several genes, the same concept is applied in genetic algorithms. Therefore each chromosome that represents a solution in the generated population contains discrete genes [5]. Mapping solutions with chromosomes requires an encoding process, the original genetic algorithm encoded genes using binary values [59]. Several encoding methods were used later like string, numeric and non-numeric values. During natural evolution, reproduction requires crossover between parents' chromosomes, therefore genes will be able to exchange. This process in nature may lead to a mutation in the generated offspring [3]. The same concepts are applied in genetic algorithms as being shown in Figure 2.6 which represents the basic pseudocode.

```

1. Initialize (Population)
2. Evaluate (Population)
3. While (StoppingConditionNotSatisfied){
4. Selection(EvaluatedPopulation)
5. Crossover(SelectedIndividuals)
6. Mutate(OffSpring)
7. }

```

Figure 2.6 Basic pseudocode for the genetic algorithms

From the above Figure 2.6 the following steps illustrates the genetic algorithm flow [4][3]:

- Initialize (population) : an initial population has to be created as a start point of the genetic algorithm, this population could be created randomly or manually.
- Evaluate (population) on this step, each chromosome that represents a single solution in the population has to be evaluated. The idea of evaluation is to calculate a fitness value for each individual therefore it could be used later for selection.
- Selection (population): after evaluating each individual, the selection operator has to choose a group of chromosomes from the evaluated set of individuals. The target is to specify good chromosomes that have better genes and to move them later to the next generation. Several methods are available for selection like elitist selection, it selects the chromosomes with highest fitnesses values..
- Crossover (population): in this step two individuals from the selected set in the previous step have to be selected to make a new change. This change has to target some genes on both individuals and to mix some of the first individual with the second one. New chromosomes with new genes will be generated from this step. The original two individuals are called parents. Several methods are available for doing

crossover, one type for example is swapping. By swapping, genes from first parents are replaced with another gene from the second parent. Determining the likelihood of applying the crossover to the selected chromosome, a crossover probability has to be defined. The crossover probability is connected with the fitness value in order to increase the probability of moving the good properties to the next generation after crossover.

- Mutation (population): it's a required change that has to be done for some genes on the generated offspring population. Choosing an individual from offspring population for mutation is based on a mutation probability. The objective from this process is to increase the diversity in the generated solution set. Even though this change has to generate a solution that has to solve the problem. If not, then the generated solution is not going to be considered in the solution set.

2.3.3 Nondominated Sorting Genetic Algorithm (NSGA and NSGA-II)

The NSGA algorithm was proposed by N. Srinivas and Kalyanmoy Deb [58][61]. The main idea about this algorithm is classifying all nondominated individuals into one category before selection. This algorithm was one of the first evolutionary algorithms, even though it has three main criticisms over the years:

1. The non dominated sort is expensive then the time cost is $O(MN^3)$ [61].
2. Lack of elitism [61], elitism refers to the important idea of keeping the good solutions alive for the next generations. In the context of multiobjective optimization, all non dominated solutions are considered as elitists [59]. Elitism could be done by keeping the elitists solutions in the populations or by sorting them in a secondary list, then reintroducing them again to the populations. This elitism process is not simple on multi objective optimization problems because of the large

number of non dominated solutions. On the other hand, supporting it within the algorithm can improve the algorithm performance and it will prevent the good solutions from being lost within different populations [61].

3. The sharing parameter is needed [61], to distribute the solutions over the pareto front, a diverse population is needed [59]. To do this, different methods were used based on generating a sharing parameter. However it may need additional cost to specify it and it may not generate the required diversity, so it's not desirable [59].

The same authors of NSGA developed another algorithm on top of NSGA, it's called NSGA-II. The above mentioned problems were resolved on the second version. By NSGA-II, each solution on the population compared with a partial population as the following:

- Keeping the first solution from original population on a new partial population \hat{P}
- Then each other solution p on P , must be compared with the partial population \hat{P}
- If there any solution q on the partial population is dominated by solution p , then remove the dominated solution q from the partial population.
- On the other hand, If the solution p is dominated by any other solution in the partial population, then ignore p .
- While if the solution p is not dominated by other solutions on the partial population, then add it to the partial population.

From the above description, we can see that for a second solution from the original population, one comparison is needed. Moreover, For the third solution, maximum two comparisons are needed. This means the cost for maximum checks is $O(N^2)$. After adding the number of functions M that are needed to be compared, the complexity is $O(MN^2)$.

2.3.4 Strength pareto evolutionary algorithm (SPEA2)

The SPEA2 algorithm was built on the basis of SPEA [68] in order to resolve some issues that were found in SPEA that could be used to find parallel multiple pareto solutions [68]. SPEA basically uses initial population and external one which is considered as archive, the archive initially is empty and it has to be updated per iteration. The non dominated members have to be stored in the external set, then this external set will be used to calculate the fitness for individuals. Briefly The basic SPEA algorithm will work as the following [67][68]:

- Create the initial population and initial empty external set which is considered as an archive.
- Store the non dominated set in the archive.
- Remove any duplicated solution in the archive.
- When archive size exceeds the limit, use clustering for pruning it.
- Calculate the fitness for both the current population and the archive.
- Select individuals from both sets by multiset union until they fill the mating pool.
- Apply the crossover and mutation, then stop if you reach the maximum number of generations. Otherwise return to step two.

Several weaknesses were specified for the SPEA algorithm, first about is related to fitness assignment where in some cases the SPEA works exactly like a random search algorithm. That happens when there is only a single member in the archive, while individuals have the same fitness value regardless if they dominate each other or not. The second issue is related to density estimation, where different individuals in the current generations don't dominate others. Therefore density is needed to collect information [68]. In this case clustering works only with the archive and doesn't work with the current generations. Therefore, search will not be effective. Third issue is

about archive truncation, where the clustering technique used in this algorithm may cause to lose some outer solutions.

SPEA2 resolved the fitness problem by assigning a strength value to each individual in both current population or the archive, that has to be done for dominating and dominated solutions in order to measure the number of solutions that each one dominates. In SPEA2 the second and third problems were resolved by the new update method [68], its operation prevents removing boundary solutions, also it keeps the archive content size as constant.

2.3.5 Multi Objective Cellular Genetic algorithm (MOCCell)

MOCCell is based on another algorithm which is called cGA, the main difference between them is using the pareto front in MOCCell for multi objectives. Density estimator is used for keeping the diversity during inserting solutions into pareto front, where pareto front is used as external or archive population. In addition, Density estimator is also used to remove solutions from the external when its size is full [69].

The algorithm is represented in Figure 2.7 , it starts by creating an empty external population which is represented by pareto front [69]. The genetic operators have to be applied successively to the individuals, while individuals have to be inserted first and arranged in a two dimensional toroidal. Two parents from each individual neighbor have to be selected. After that, in order to get the offspring, recombining has to be done, then mutating it. For the resulting individual, evaluation has to be done. After that, inserting it in both populations: auxiliary and the pareto front . Per each iteration, replace the old generation with the new one which is the auxiliary. Finally, a feedback procedure will replace a random fixed number of individuals in the populations from the external population which is the pareto front [69].

```

1. proc Steps Up(mocell) //Algorithm parameters in 'mocell'
2. Pareto front = Create Front() //Creates an empty Pareto front
3. while !TerminationCondition() do
4. for individual ← 1 to mocell.popSize do
5. n list←Get Neighborhood(mocell,position(individual));
6. parents←Selection(n list);
7. offspring←Recombination(mocell.Pc,parents);
8. offspring←Mutation(mocell.Pm,offspring);
9. Evaluate Fitness(offspring);
10. Insert(position(individual),offspring,mocell,aux pop);
11. Insert Pareto Front(individual);
12. end for
13. mocell.pop←aux pop;
14. mocell.pop←Feedback(mocell,ParetoFront);
15. end while
16. end proc Steps Up;

```

Figure 2.7 Pseudo Code for MOCell algorithm

2.3.6 Indicator- Based Evolutionary Algorithm (IBEA)

The basic idea of this algorithm is to define a binary indicator that represents an optimization goal. IBEA algorithm based on doing tournaments for mating selection and environmental selection. It has to remove the worst individual iteratively. After that, it has to update the remaining individual fitness values in the populations. Regarding to algorithm performance, if the population size is α then the execution time of the algorithm is $O(\alpha^2)$.

2.3.7 Solutions Evaluations by Hyper Volume

Several quality metrics are available to evaluate the generated solutions from genetic algorithms, these metrics are being used to validate the goddess of Pareto solutions set in the objective space. HyperVolume (HV) is

one of the most used indicators for solutions evaluations. It's a quantitative value that represents the difference between the size of objective space that is dominated by the observed pareto solution set and the space that is dominated by a true Pareto solution set [12]. The true Pareto solution set dominates the entire solution space, therefore the observed solution set is evaluated by measuring how much it's worse by comparing it with the true Pareto solution set. In this thesis, hypervolume will be used for solutions evaluations in all experiments that will be proposed later.

2.4 Optimization for Testing

Structural Testing is one of the most testing types that was used in research of SBSE [55]. The idea here is to measure the quality of a program by measuring the coverage depending on the structure of the program. For example, measuring the branch coverage is a method where we could measure the quality depending on the structure of the code. This will definitely give us an indication about the coverage in general. The search based on structural testing was first applied on C programming language programs. After that it was applied on object oriented [53]. A second type of testing that was used in search optimization is a model based testing. Depending on the selected model such as finite state machine, data flow, control flow and others, we need to generate different unique sequences of data input output values. Search based algorithms also were applied on research for mutation testing, in this testing, test engineers seed several faults on the code. After that, they generate inputs and execute test cases with these inputs on the mutated version of the code. That has to be done in order to check if these inputs will be able to detect the fault. If the previous step succeeded, the mutant will be considered as killed by that input. This will give an indication where that input will successfully detect the faults on the real code that is not mutated with bugs. The search based on the previous testing was used to specify the best set of inputs that will kill the seeded mutant [53]. Exception testing also was used in SBSE in order to find the best inputs that could be used to produce exceptions [53]. Exception testing is a testing type that is used to test handled exceptions in the system. Another important research was done on regression, the number of cases

might be large and might increase the time. Sometimes if the wrong decision was taken to neglect some cases, this might be resulted in reducing the coverage. This important problem could be resolved by SBSE[53].

Testing phase sometimes includes nonfunctional requirements, one of these requirements is execution time. SBSE was used also with temporal testing, it refers to the type of testing in which the shortest or longest execution time is measured in order to maximize or minimize it. SBSE is applied here in order to find the cases where there is an objective to reduce or increase the execution time [53][55]. Another type of testing for nonfunctional requirements is stress testing. It's used to find the points in which the system will be broken because of degraded performance. This problem was also used to apply SBSE in order to find the test cases in which the performance might be degraded [53]. Different algorithms were used for resolving the mentioned optimization problems, NSGA-II is the most used one as a genetic algorithm [56].

Chapter 3. Related Work

Most work about test cases prioritization solutions in research was done for regression testing. This is related to the fact that was mentioned in the background about regression testing time. The time is connected with the number of test cases from previous releases. More features means more regression test cases, then more regression testing time. To see the gap for this research test cases prioritization problem in previous work, different solutions in previous work are listed and discussed here in this chapter. All these solutions are classified under regression testing or under new test cases that are going to be tested for the first time.

3.1 Prioritization Solutions for regression test cases

All regression prioritization solutions in previous research have different objectives to do prioritization. These solutions are also classified under main general solutions as the following:

3.1.1 Structural Coverage Based Solutions

These prioritization solutions under this category are connected with structural code coverage that was discussed in the background. The main objective for those solutions is enhancing the error and defects detection in early stages by code coverage [23][24].

Different studies were proposed under structural coverage, in this section different examples will be discussed. First example, an important research study, used branch coverage and statement coverage to do prioritization. The researchers resulted in increasing and enhancing the fault detection [28]. Similar prioritization solutions were proposed by other researchers in a different study, in this case two structural solutions were used in order to prioritize the test cases [25]. First is statement coverage, it prioritizes test cases depending on the number of code statements that could be

covered by each test case. The second one is function coverage, in this case prioritization for test cases was done depending on the number of code functions that could be covered from each test case. The study did an experiment that checked the impact of using different versions of the software. Versions were important in order to see the impact of new prioritization by collecting new data for each version. The study resulted in a better coverage when doing a new prioritization for each version. In addition, an important result from the previous study was done by having a comparison between function and statements coverage. The result approved that the statement is more costly than function. On the other hand, statement coverage is better for early detection of bugs. This means in case there is a critical impact for late discovery of bugs, statement is recommended to be used [25]. The two previous mentioned papers [28][25] have similar methodology, they both need source code in order to measure the coverage whatever it's. Also, it's not recommended to start using these solutions before finishing the system development phase. The reason behind this is the final numbers of statements and functions are not known in the beginning of the development phase. Moreover, technical skills and knowledge are required in order to return and understand the code during measuring the coverage.

Other structural solutions were proposed in another study that worked depending on the interaction or events coverage [26]. This study resulted in five prioritization techniques that prioritize test cases depending on how much each test case will cover events. First solution prioritizes test cases depending on how much test cases will cover unique events as early as possible. The second one prioritizes test cases depending on the interactions with events. It measures how the interaction with events can cover different parameters for the event. The third one prioritizes test cases depending on it's length during interacting with the event from shortest to longest. Fourth is similar to the third one, but test cases are prioritized from longest to shortest. Final one is random prioritization. All of these techniques are useful when there is an interaction coverage from test cases, otherwise fault detection will be low. These interaction prioritization techniques could be applied on regression

testing or functional testing which is kind of system testing. On the other hand, they need source code, technical knowledge and they will not be able to be applied before finishing the system development phase. The same reason behind finishing development code is mentioned for previous studies. There is no guarantee how the events will be before freezing the coding by the development team.

More structural solutions were proposed in another research by considering the new source code changes coverage and the related source code [27]. The study objective is to decrease the cost of regression testing by minimizing the number of regression test cases. As the previous structural studies, source code access is needed here in addition to technical knowledge.

From all mentioned studies, structural testing coverage solutions need source code and technical skills. The different results between these solutions is fault detection percentage and the system nature that needs specific techniques in some cases. Moreover, there is a need to have the final implementation for the system or the feature under testing. This will give the correct and accurate number of code coverage. As a result, applying these solutions is suitable for regression testing. Since the regression is usually done after finishing the new code changes. In regression testing you will have the opportunity to measure an accurate number for the code coverage regardless what the coverage solution is, this will give better prioritization.

3.1.2 Fault Detection Based Solutions

The proposed solutions under this category are working by making a classification for test cases depending on the probability of failures per test case. This probability is measured by returning to code statements and checking if the statement caused failure, will it lead to test case failure or not. In case it will cause test case failure, this will increase the fault probability value for the test case. A research study that was discussed under structural categories also proposed a solution under fault detection [28]. In addition to branch and statement coverages, the same research presented other solutions. These solutions added the probability of test case failure by measuring how

the statement or branch may lead the test case to fail. The research resulted in proving that fault detection will be improved using these solutions. Even though, they are considered as too expensive because you have to check each statement and its impact on all test cases.

Another fault prioritization solutions were proposed on a paper that also presented structural solutions [25]. This research added fault probability by statements and functions in parallel with code coverage. Different versions for software also were used, this is exactly as supported in the same paper for structural coverage. The solution for measuring the faults probability depends on the fact that some functions are a source of errors more than other functions. These functions should be given an index when they are expected to cause errors. This could be done by comparing the current version to previous versions, checking the new changes and then expecting the errors. If that function has to be executed during the test cases execution, then this will lead to increasing the test case priority. Similar work was done in another research paper [29] which was presented by the same authors of previous research for test cases prioritization [28]. They invested in the versions concept by a controlled experiment in order to study the fault detection. The research had a similar result to the previous one, fault detection will be improved by using different versions of software with prioritization solutions. Another important result from the same study is about the cost of failure probability based solution. While the statement fault detection is too expensive, function or branch is less expensive. On the other hand, the statement will work better for early detection of errors with higher rate coverage.

As we can see from the previous examples, failure detection based prioritization solutions are actually structural solutions. Instead of code coverage only they added the fault probability. As a result, the fault based solutions need access to source code. Also, some kind of technical knowledge and skills are needed in order to understand the code. Moreover, code has to be finished by the development team in order to measure the final faults probability from code that has an impact on test cases. From this result, these

solutions are easily used with regression testing where development phase and functional testing work for the new feature are done.

3.1.3 History Based Solutions

Test cases prioritization is done here depending on the previous test execution results. Therefore the previous results are considered as a history which could be used to prioritize the current release regression test cases. The objective of those solutions is to help in increasing the testing effectiveness and reducing the regression cost by using the history results as data source [23][30].

Different previous prioritization solutions are able to be used with history based solutions. For example, code coverage history could be applied here separately or with fault history. A proposed solution prioritizes test cases depending on fault history of test cases from previous releases, it also used function coverage history [30]. This solution calculates the number of faults per test case from previous executions. After that, for the same test case, the number of functions that were covered from previous executions will be measured. Therefore, the priority of that test case will be calculated depending on faults detection and functions coverage from previous releases [30].

Another solution was proposed in another research by calculating a historical value of the test case [31]. The prioritization will be done later by the calculated historical value only. It also could be done by merging it with another regression test cases prioritization solution such as structural coverage solutions. The historical value itself was calculated in the mentioned research by basically returning to the historical cost of the test case. After finding the cost, the solution has to find historical data about faults severities that were detected by executing the test cases in previous releases [31]. The cost was considered as execution time for that test case. While the severity was considered as to how much the detected fault has critical impact on the system. These historical values could be applied in different ways. Considering the cost as monetary, human resources time or machine time are examples. Moreover, instead of considering the fault severity in historical value, code

coverage such as statement or function could be considered [31]. The decision behind this has to be done depending on what is available to the testing team.

Another prioritization solution was reviewed in related work and it is categorized under historical based solutions. It depends on performance as a historical data [32]. Performance in this study is related to the ratio of how much the test case detected total number of defects in total number of previous releases. This is connected with the fact about the nature of test cases that might not be executed in all previous regression tests. On the other hand, some test cases might detect a higher number of defects. Even though they were executed in less number of releases. The mentioned proposed solution takes into consideration another two measures in addition to performance. First one is the priority of test cases in previous releases. While the second one is the duration that the test case wasn't executed [32].

Other similar studies were proposed based on historical data with changes on research methodology, environments, goals and new contributions. For example a research was done for prioritizing regression test cases in software companies that follow continuous integration in their process. This research used historical faults per test case for deciding its priority. Since the continuous integration nature automatically tracks the faults per execution, it's easy to collect historical faults [33]. Another study proposed a solution for test cases prioritization depending on the cost [34], fault detection and severity of the fault. The proposed work here is similar to previous work in previous mentioned research which used the performance cost [31]. The difference for this research is using the genetic algorithms for implementation. However, In related work, there are a lot of studies that were proposed based on genetic algorithms for test cases prioritization using historical data.

As we can see, all the mentioned studies are talking about regression testing. This means that they are not able to be applied for any level of testing when the test target is a new feature or a new change. The reason behind this is the need of historical data that is the basic part for building prioritization. As a result, since there is no history for test cases of the new change or for the new feature then no historical data. All history based prioritization solutions

are not applicable here. In other words, in order to apply them at least two releases of the same test cases are required.

3.1.4 Requirements Based Solution

One important reason behind faults that might be discovered in the testing phase is requirements [35]. Missed requirements or misunderstanding will have a critical impact on customer satisfaction. During system testing there is a need to test each single scenario that was requested in requirements by the customer. In system testing, requirements documents are the reference point for testers. Therefore functional testing could be done without any need to return to the source code itself. Different test cases prioritization solutions were proposed for prioritization of test cases for system testing by only depending on requirements. This might be applied for a new feature which needs new test cases that will be executed for the first time. Also it might be applied for regression test cases that are needed to be re-tested per new release. This is needed and important when the testing team doesn't have automation testing. It is also important when there is no technical knowledge, then testing team work by black box testing for each new release.

First solution is able to be applied for new test cases that are related to a new feature or new change. In other words they are going to be executed for the first time [36]. Also the solution is applicable for prioritizing the regression testing depending on requirements only [36]. The proposed solution used different factors that were extracted from requirements to generate the prioritization. First factor is the customer's priority. The customer who needs to use the expected system and who is the main source for requirements has to assign priorities to the requirements list. Second factor is the requirements volatility. This refers to how much requirements are stable or continuing change during the development cycle. A lot of work might need to be re-done again after changing requirements. Sometimes the change is needed for design as an example. On the other hand, there is a need for several changes on the code by adding, removing or refactoring. Therefore this factor is a major cause for errors and it's important to be considered in the mentioned solution

that depends on requirements [36]. The third factor is implementation complexity. The developers who were assigned to develop the system have to give their expectations about the code complexity. This complexity has to be evaluated before starting development. This means it will depend on requirements that have to be implemented by the code. The last factor is fault proneness, this is about how the number of faults that are expected to be discovered by a specific requirement. In order to expect the faults, history is needed to be tracked here. Therefore, this factor is only considered when the prioritization is needed to be done for regression testing only. By contrast, in case of having new test cases that are going to be executed for the first time, there is no history from previous releases then this factor is not considered [36]. The solution approved that customer satisfaction will be improved by using this prioritization technique. This will help in early discovery of faults which will directly have an impact on customer satisfaction [36].

Other authors proposed similar two solutions under requirements based prioritization in two different research studies in 2008 and 2009 [38][39]. First study has the same four factors from the previous mentioned study [36]. Both proposed solutions [38][39] are able to be applied for prioritization of new test cases that are connected with new requirements. Also they are used for prioritization of regression testing. This is a similar point with the previous discussed research [36]. First new study [38] added two new factors that are different from [36], so prioritization is able to be done by six factors. Those factors are: customer priority for requirements, developer code complexity expectation, rate of requirements changes, fault severity, usability and the last one is application flow [38]. For new test cases, the first three factors are only used. On the other hand, the last three factors are only used when the target of prioritization is regression testing [38].

About the first new factor which is usability, this factor is used in order to measure how the system implementation was easy to be used by customers after releasing the feature. This factor is important for system quality evaluation from the customer perspective. Therefore each requirement should be rated from the usability perspective. Once the feedback is ready from the

customer about the released feature, it has to be considered for the prioritization of regression test cases in the coming release [38]. Since the feedback is only ready after the first release of the system or feature, it's a normal reason to use this factor only with regression testing. On the other hand, for new test cases the system or feature is not implemented yet and then the system is not released. Hence, there is no available feedback. The second new factor is application flow. It is important for measuring how the functionality that represents requirements was going from one release to another. So, another rate will be given for each requirement depending on its implementation behavior between different releases [38]. That means it's also a normal reason to consider this factor only for regression test cases prioritization. Since we need to see the functionality behavior from release to another, then we can't use it with new test cases. These new test cases or new requirements were not available in previous releases. Therefore, there are no previous releases to compare and evaluate the requirements depending on the behaviour which is not available in this case.

In the second new mentioned research, a similar solution was proposed in 2009, it's very close to the previous one and it's for the same authors [39]. Also they used the same mentioned factors that were used in the first discussed research under requirements based solutions [36]. In addition to the four factors it added another two factors. They are only valid and applied on regression testing, while the same first three factors are valid and applied for new test cases. The new added factors for regression testing are completeness and requirements traceability. About completeness, it refers to measuring how much the requirements that are going to be re-tested or reused for regression are complete from a customer perspective. Hence, when regression is needed to be done, each requirement has to be verified if it satisfies all customer needs under each specific condition with expected performance. After that, a requirement will be given a value or rate from customers depending on it's completeness measurement. Then it will be used for test cases prioritization [39]. About the traceability which is the second added factor, it refers to measuring how much the selected requirement life cycle was tracked and

traced from its first cycle or first release until last feedback from customer after releases. The reason behind this from the authors' perspective is increasing the quality by tracking the requirements in different releases [39]. Although this study is similar to one of the previous studies [36], it was not compared with it by authors. Even though their final result approved that the fault detection could be better by their new proposed solution [39].

Another prioritization solution was listed under requirements based solution was proposed with specific kinds of applications, these applications are web services [37]. Test cases prioritization is done here by depending on how much the test case will satisfy or cover the constraint of request quota. This quota is supported by the web services structure, the APIs calls and requests. It should be specified as non functional requirements in the requirements phase. The generated solution was applied in the research on regression testing. We can notice that from the application nature, the code should be ready in order to know the final version of quotas and web service. Moreover, it's similar to structural based solutions and failure based solutions that connect test cases with code statements. The difference here is connecting the test cases with quota or APIs requests. As a result, this kind of prioritization is not able to be applied without some kind of knowledge about source code and without stable code that will not change again. That means, it's more suitable to work well with regression testing.

3.1.5 Hybrid Approaches

Different solutions were proposed by combining different solutions from above categories in order to enhance the fault detection. For example, a prioritization solution was proposed depending on the fault detection and historical data for previous execution time [40]. The proposed solution was built using genetic algorithms in order to enhance fault detection for regression test cases. In that solution time is considered as a constraint for releases [40]. Therefore this solution is only applicable for regression testing since we need history data. Another example for these solutions was combined between requirements based and historical based solutions [41]. It applied the

genetic algorithm by considering how much the test case covered requirements. It also takes into consideration the historical data about the execution time of test cases. Because the mentioned solution needs historical data, then it's exactly like other history based solutions. It's only able to be applied for prioritization of regression test cases, then it will not work with new test cases. A greedy algorithm was also used to propose another solution that combined between code coverage and test cost. The Cost is considered as historical data about the test cases [42]. This solution is similar to the same previous structural and historical solutions. Therefore it will not be used with new test cases and it's only applicable with regression test cases.

A lot of solutions were proposed by mixing the above approaches .Some times as discussed previously by combining code coverage, fault based, history or requirements. As code, fault detection and history are concepts that are connected with stable code that will not change or with different releases. , then generally these methods are valid for regression testing. At least, there is a need for having and understanding the stable code and then measuring some values, or there is a need to collect data from execution history. Ofcourse, that is not suitable for new test cases prioritization since no available executions to be used as history. In addition, the source code is not stable for new features and different changes have the chance to be added to the code. That means, all coverage measures are not accurate and then it's not possible to use them in prioritization.

Other studies that are not connected with previous solutions and are not listed under the previous categories were also proposed. As an example, one study proposed a solution which depends on data flow in the program and specifically all program variables [43].This definitely needs a final stable version of program code. The concept for this study is similar to previous mentioned structural code based solutions. That means it could be applied only for regression regression test cases for the same reasons. Another solution was proposed specifically for COTS components regression testing [44].

COTS are third party components that are integrated and reused in different systems. In some cases these components have new changes and

different versions might be released. As a result, new changes for software systems that use them will happen. This means some faults may occur for different systems depending on their contexts. That needs a new regression testing to the system that is integrated with this component. The authors proposed a solution for regression test cases prioritization depending on measuring how much the test cases interact with the component [44]. The proposed solution needs to check classes or solutions in the source code that represents the test cases. It's actually a representation for the real interaction with the component. This solution is only applied for test cases that interact with components on a specific target. The case applied only when there is a new change on the component itself, while the system code has no change. Since the code is stable and has no change this is exactly the regression test cases. In addition, the normal system will have different test cases that may interact or may not interact with COTS. This approach is not able to be applied for testing that includes all test cases that might not interact with COTS components. So this approach is able to be used for prioritizing test cases that are connected with the COTS that have new change only. This also means that some kind of technical knowledge about source code is needed in order to check which test cases are interacting with COTS. In this case COTS components are not visible for the tester. Therefore the tester will not be able to know if the test case will interact with that component or not ,either for the new test case or regression.

Another proposed prioritization solution could be listed under more than one category was proposed for web applications regression testing [45]. It prioritizes test cases by depending on the number of requests that were covered by the test case from previous executions. Also test cases could be prioritized depending on the number of pages that were covered by the test cases in previous executions. In addition, function parameter coverage could be used by the proposed solution [45]. This definitely is not be able to be applied for new test cases since all the mentioned measures need previous executions. Therefore they have the same limitations of history based

solutions. They also combined the history with structural based solutions, so the same limitations also are valid here.

3.2 Prioritization Techniques for New Test Cases

When the feature or system under testing has a large number of test cases, then it's very important to prioritize these test cases for better and effective testing and early fault discovery. The importance of the prioritization process for new test cases also comes from the fact about limited resources and strict time. Without prioritization, the limited time might be spent with the test cases that are connected with low priority requirements. Moreover, some test cases might discover the important bugs. If the team didn't start with these test cases, this may lead to late discovery of bugs. It may need critical decisions on the whole system.

In the related work, a lot of work was done for prioritization of regression test cases as described in the previous section. While for new test cases that are needed to be tested using black box testing there is a very limited work in this area. The following terms were used during the this research to search about any related work for new test cases prioritization:

- "Prioritization for new test cases", since the problem in this research is related to new test cases. In the previous section, different regression prioritization solutions were listed and there is a need to specify the ones that are specifically proposed for the new test cases.
- "Black box testing prioritization", since the new test cases in the testing process are usually tested by black box testing where code is blinded for testers, some solutions for new test cases might be discovered under the mentioned expression.
- "Manual test cases prioritization", new test cases are usually included with black box testing that is done manually.
- "System test cases prioritization", system testing includes new test cases that are generated from the requirements specification document.

- “Functional testing prioritization”, because functional testing is a kind of system testing, it’s also related to new test cases that are generated from requirements.
- “GUI testing prioritization”, some coverage metrics were presented for GUI, and for each new feature or system, a new GUI will be developed. Therefore new test cases are needed with different combinations.
- “White box testing prioritization”, this expression was used in case there are solutions under white box, then it might be possible to apply them on other types of testing.
- “Component based frontend test cases prioritization”, since the test cases prioritization problem in this research is itself connected directly with these types of applications.
- “Component based frontend testing”, this is a general search about testing for the component based application in order to see the final research in this topic.

The result from all of the above expressions was only the mentioned solutions that were discovered and discussed in the regression testing section [36][38][39]. As was discussed before, these solutions are classified under requirements based solutions, where they are applied either for new test cases or for regression testing.

In all mentioned solutions, some factors were only used for new test cases and for regression test cases at the same time. Additional factors are applied only for regression test cases. The three previous solutions that used the same factors: customer priority, requirements change and development complexity. These factors are general and could be applied to all types of applications. On the other hand, what about specific types that have more factors that might have an impact on the testing phase and test cases prioritization? For example when the front end framework is considered as a component based architecture, these components are reused in the same feature in different contexts and integrations. Also what about the factor of development expectations when something is missed as usually from development, then no accurate expectations. Moreover, in most cases it’s very hard to measure how the requirements could be changed and when. Therefore it may be very important to find new techniques for new test cases prioritization.

3.3 Related Work and the Research Gap Summary

This research has found a solution for web applications that are going to be built using component based architecture front end frameworks. At the same time, the prioritization has to be done for new test cases that are going to be tested for the first time. Since this is a web application then the testing needs to be done by blackbox testing that is usually used manually for web applications that have a graphical user interface. From related work that was discussed in previous sections, the following table 3.1 provides a summary for all mentioned solutions. In addition, Table 3.1 illustrates if the solutions are able to be applied with this research gap which is related to the mentioned kinds of web applications.

Category	#	Solution	Is it applicable with new test cases for web application?	Weaknesses of applying it with new test cases for web applications that have component based frontend frameworks
Structural Coverage Based Solutions	1	Branch coverage [28]	No	<ul style="list-style-type: none"> - Can't applied with manual and black box testing because it needs source code - Needs stable source code to get accurate measures, so it's suitable for regression
	2	Statement Coverage [28]		
	3	Statement coverage [25]	No	
	4	Function Coverage [25]		
	5	Interaction coverage [26]	No	
	6	Events Coverage [26]		
	7	Coverage of new changes in the source code [27]	No	<ul style="list-style-type: none"> - Can't applied with manual and black box testing because it needs source code - It was designed for regression test

				cases that are related to previous versions of the code. While for new test cases there is no previous code or version
Fault Detection Based Solutions	8	Probability of test case failure by branches [28]	No	<ul style="list-style-type: none"> - Can't applied with manual and black box testing because it needs source code - Needs stable source code to get accurate measures, so it's suitable for regression - When using versions this might be considered as a history based solution, for new test cases there are no previous versions.
	9	Probability of test case failure by statements [28]	No	
	10	Probability of test case failure by statements [25]	No	
	11	Probability of test case failure by functions [25]	No	
	12	Probability of test case failure by branches and versions [29]	No	
	13	Probability of test case failure by statements and versions [29]	No	
History Based Solutions	14	Test cases faults from previous execution with function coverage [30]	No	No History for new test cases because the new feature is not released yet, this means there are no executions for test cases to collect data about test cases.
	15	Historical value from previous executions. Value = test case time + faults severities in previous executions [31]	No	
	16	Test case performance measures [32]: 1- The ratio of how	No	

		<p>much the test case detected faults in the total number of previous releases.</p> <p>2- The duration of the test case that wasn't executed</p> <p>3- Priority of test case in previous releases</p>		
	17	Historical faults for test case from continuous integration executions [33]	No	
	18	Test case cost by faults and severity with genetic algorithm [34]	No	
Requirements Based Solutions	19	Customer priority + Requirements volatility + implementation complexity [36]	Yes	<ul style="list-style-type: none"> - Can't expect the requirements volatility in most cases - General approach which doesn't take in consideration the UI components - Complexity might not be connected with priority
	20	Customer priority + Requirements volatility + implementation complexity + fault proneness [36]	No	New test cases have no values for faults, so this solution was designed on top of the previous one for regression testing that has history
	21	Customer priority + Code Complexity + Rate of Requirements Changes [38]	Yes	<ul style="list-style-type: none"> - Can't expect the rate of requirements change in most cases - General approach which doesn't take in consideration the

				<ul style="list-style-type: none"> - UI components - Complexity might not be connected with priority
22	Customer priority + Code Complexity + Rate of Requirements Changes + Faults severity + Usability + Application Flow [38]	No		<ul style="list-style-type: none"> - New test cases have no values for faults - Usability measure is considered as feedback after releasing the software, so it can be used in the next release for regression as history. - Application flow is also connected with different releases and for new test cases there is no release yet
23	Customer priority + Requirements volatility + implementation complexity + fault proneness + Completeness + Traceability [39]	No		<ul style="list-style-type: none"> - New test cases have no values for faults - Completeness of requirements has to be measured after releasing the software and this is not available for new test cases. - Traceability is measured also between different releases and it's not available for new test cases.
24	Request Quota Requirements coverage for Web services [37]	No		<ul style="list-style-type: none"> - It's used only for web service and it's connected with its architecture. - Requests Quota might be only a part of test cases

				<p>that are needed to be tested as nonfunctional requirements.</p> <ul style="list-style-type: none"> - Needs source code and will not work with blackbox testing.
Other Solutions	25	Combination of fault detection and execution time history [40]	No	No history for new test cases, so it's also used for regression test cases
	26	Combination of execution time history and requirements coverage with genetic algorithm [41]	No	No history for new test cases, so it's also used for regression
	27	Combination of code coverage and historical test cost [42]	No	No history for new test cases, so it's also used for regression
	28	Data Flow for program variables [43]	No	Needs source code and technical knowledge and this not available for manual testing of new test cases
	29	COTS changes by classes or solutions interactions [44]	No	<ul style="list-style-type: none"> - Works for minimizing the number of regression test cases, so it works only with test cases that are connected with components that have new change - Needs source code access to check classes that are connected with targeted components.

	30	Number of requests from previous executions for Web applications [45]	No	<ul style="list-style-type: none"> - Different test cases are not connected with requests. - No previous history and executions
	31	Pages coverage for web applications [45]	No	<ul style="list-style-type: none"> - Different test cases are not connected with more than one page
	32	Functions parameters coverages in web applications [45]	No	<ul style="list-style-type: none"> - Needs source code access and technical knowledge therefore this is not the case with new test cases and manual testing

Table 3.1 Summary of prioritization solutions and the possibility of using them for this research

Thirty two solutions were summarized in the above table, it's clear that most solutions are not able to be used for the test cases prioritization problem. Most solutions need source code in order to measure the coverage. This means the system type itself is not important if it's component based architecture or not. Doesn't matter if the application is a web application, mobile or desktop. Using the source code needs a level of technical knowledge and this in different cases is not available for the testing team who is working by blackbox testing. As a result, if the test cases prioritization problem needs to be resolved by any solution that needs source code, at the same time, the testing team doesn't have technical knowledge, these solutions are not applicable. Moreover, measuring the code coverage, regardless of what is the coverage type, needs to use a stable and frozen code. This is required in order to take the correct coverage per test case, and this is not the case for the new test cases. The code will not be stable, different bugs, fixes and changes are going to be added for the new code. So, all of these solutions are suitable for regression test cases that have stable code and tested from previous releases. Other available solutions need history

data from previous executions of previous releases, and this is not this research situation. This research target is a new testing that doesn't have any previous execution to collect data from, therefore there is no history.

From the summary table, there are only two solutions that might be used for the test cases prioritization problem, their numbers are 19 and 2 in the above table. These two solutions have clear weaknesses for applying them in the problem application which is connected with component based frameworks. Even if they don't work for a specific kind of application, they don't consider the impact of using component based architecture. Also, these two solutions consider an expectation about requirements changes, and this is not easy to be expected correctly in most cases.

As we can see, there is a clear gap in research with new test cases that do not have a stable source code or a previous history yet. Hence, This research solution will be designed specifically for web applications that use component based front end frameworks. This means the complexity and simplicity of this architecture will be considered in the prioritization. Source code or historical data are not important at all, and this means the solution will fit the new test cases without any wrong or missed data.

Chapter 4. Research Methodology

This chapter introduces the research methodology and experiment design. It provides the details for each stage of the experiment and the expected output from each one. This chapter also describes the datasets, how to create random datasets with any size for testing and what are the required fields to design the dataset. In addition, this chapter describes the tools and framework that has been used to implement the solution.

In this research there are two phases, the first one is solution design that includes the chromosome structure, fitness functions and datasets design. Different targets have to be achieved from the designed solution, while the final output has to be a prioritization framework that will increase the test cases coverage and the high priority test cases coverage. It has to be helpful for the resources that are going to execute test cases with no previous knowledge about the priorities and requirements. The solution has been implemented by four genetic algorithms, after that a violation algorithm has been applied to find and remove any test case that violates its dependencies. The whole implementation details are presented in chapter 5, while this chapter focuses more on the test cases prioritization problem and genetic algorithms design.

In the second phase of this research, several experiments have been presented to measure the solutions quality. In addition, the minimum execution time has been measured for each algorithm. Several datasets have been used to take these measurements later.

4.1 Facebook Dataset

Since there is no previous research about the research test cases prioritization problem, then there is no dataset for a website that was built using component based frontend frameworks. This research added a contribution by creating a new dataset for a website which was built using frontend component based framework. Facebook as a

web application which was built using react framework will be used for creating the test cases as a new dataset. In addition to the Facebook dataset, several datasets have been created for this research randomly. These datasets have the main and required values for doing experiments without having a real description for their test cases. Therefore, they could be considered as a simulation for real test cases in order to measure the impact of dataset size during the experiments phase.

Choosing the Facebook app is related to having different components that are reused in different contexts and areas in the same web app. Therefore we can find different components that are composed from other smaller or micro components. In the Facebook application, other components that are not reused also could be found, they are simple single components. Hence, there is a good diversity that helps in testing the implemented solution. The test cases have to be documented in a high level form by writing the scenario and the properties for each test case as represented in the attached dataset in appendix. The following samples were taken from the generated dataset:

A	B	C	D	E	F	G	H
ID	Test Case	TC Priority	Dependency	Time	Component	Component Priority	TC Weight
10	Verify that new post view on page has text area with page default value	5	-	15	New Post View	7	12
11	Verify that new post view on Group has text area with group default value	5	-	15	New Post View	7	12
12	Verify that new post view on friend profile has text area with friend default value	5	-	20	New Post View	7	12
13	Verify that logged in user image is added on on user profile	1	-	20	New Post View	7	8
14	Verify that logged in user image is added on on group	1	-	20	New Post View	7	8
15	Verify that logged in page image is added on page	1	-	20	New Post View	7	8
16	Verify that logged in page image is added on friend profile	1	-	20	New Post View	7	8
17	Verify that new post view component on profile has different types that text and they should be visible on specific order	4	-	20	New Post View	7	11
18	Verify that new post view component on Page has different types that text and they should be visible on specific order	4	-	20	New Post View	7	11
19	Verify that new post view component on Group has different types that text and they should be visible on specific order	4	-	20	New Post View	7	11
	Verify that new post view component on friend profile has different types that text and they should be visible on specific						

ID	Test Case	TC Priority	Dependency	Time	Component	Component Priority	TC Weight
100	Verify that sharing new post on your profile with tagged friends works well on profile and tagged friends profiles and notifications	4	1- New Post View 2- View Post 3- Write Post 4- Tag Friends	20	Notification	2	6
101	Verify that sharing new post with tagged group members works well on profile and notifications	4	1- New Post View 2- View Post 3- Write Post 4- Tag Friends	20	Notification	2	6
102	Verify that sharing new post on friend profile with tagged friends works well on profile and tagged friends profiles and notifications	4	1- New Post View 2- View Post 3- Write Post 4- Tag Friends	20	Notification	2	6

Figure 4.1 Dataset Sample represents test cases and their properties

From the above sample, the following properties were specified per test case:

- ID: it's just a unique integer for the test case.
- Expected time: it represents the execution time for the test case including preparing the input data for that test case.
- Test case priority: it's a description that represents how much this test case is important for the new feature to be released. This is also a representation from a test engineers perspective, usually it has a value from the following set: Highest, High, Medium, Low, Lowest. As a result, this may lead to different test cases with same priority even some of them may have higher priorities.
- Component priority: if the test case execution target is to test a new component behaviour, and this component has to be built by the frontend team, then the component itself should have a priority. This priority has to be evaluated by the frontend team which knows exactly how it will be used or reused later to implement the feature. Once the frontend team had more expectations to reuse the component in the feature, definitely this has to increase its priority. To represent this in the research dataset, a list of all components that are needed for the dataset was created, it represents a sprint or release. Then a value for each component was given to represent the priority, it's attached to the same link of test cases in the appendix. Numbers had been assigned starting from 1 which represents the lowest priority component. While the total number of components in the targeted release represents the highest component.
- Test Case Dependencies: some test cases need different steps for execution, moreover, some steps need different modules to be implemented. The execution steps or modules may need other components. Moreover, the test case may target to verify a behaviour of the component that consists of micro or other smaller components. On the other hand, the test case itself does not target to test these smaller components or other steps components behaviour, it needs to verify a behaviour for a special component. But as we can see, testing the special component depends on other steps or it has dependencies with other components. Therefore test cases might have dependencies or might not,

and this will have an impact on test case priority. This impact comes from the fact that dependencies actually represent an important kind of testing, it's integration between components.

In order to generate the properties, the following examples illustrate the situation:

- **Example1:** From Facebook app, let's say we have the following three components with given names just for building this research dataset:
 1. New Post View, from this component you can click to start writing a new post, after that you need another component to actually write the post. This component is just used for giving indication for users from where they can add a new post.

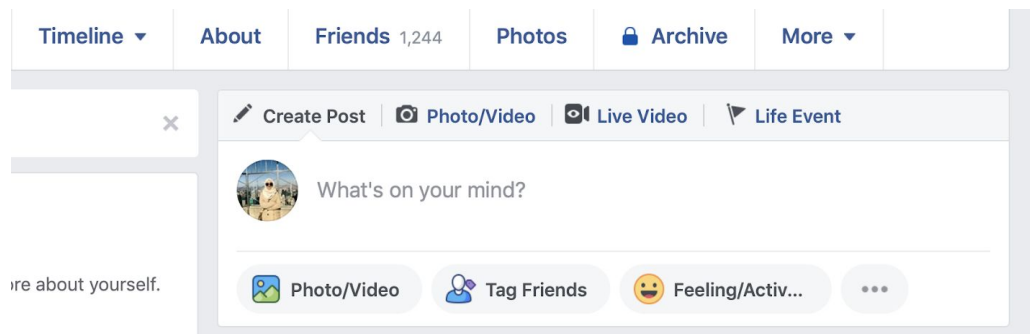


Figure 4.2 Facebook Component - New Post View

2. Write Post Component: it's used for start writing text, uploading photos or whatever the user wants to add in the new post.

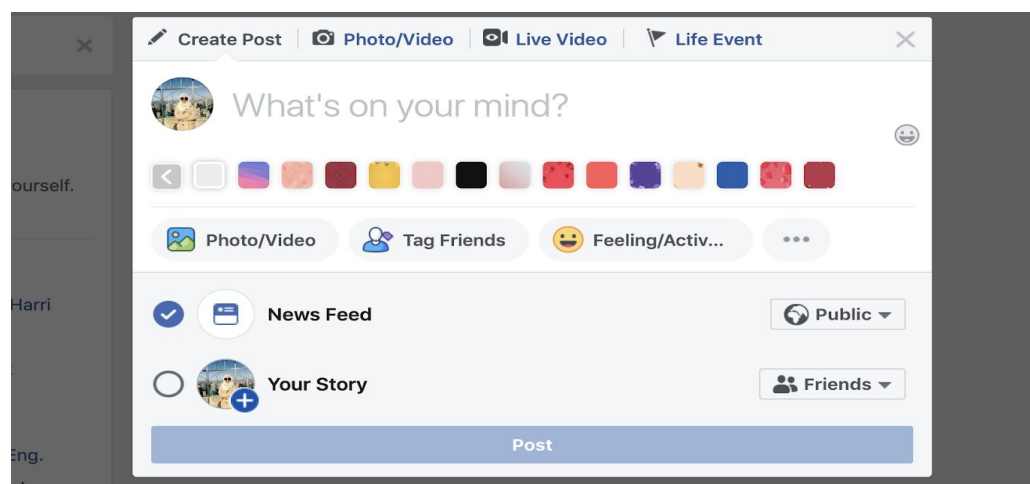


Figure 4.3 Facebook Component - Write Post

3. View Post Component: it's used to display any successfully created post.

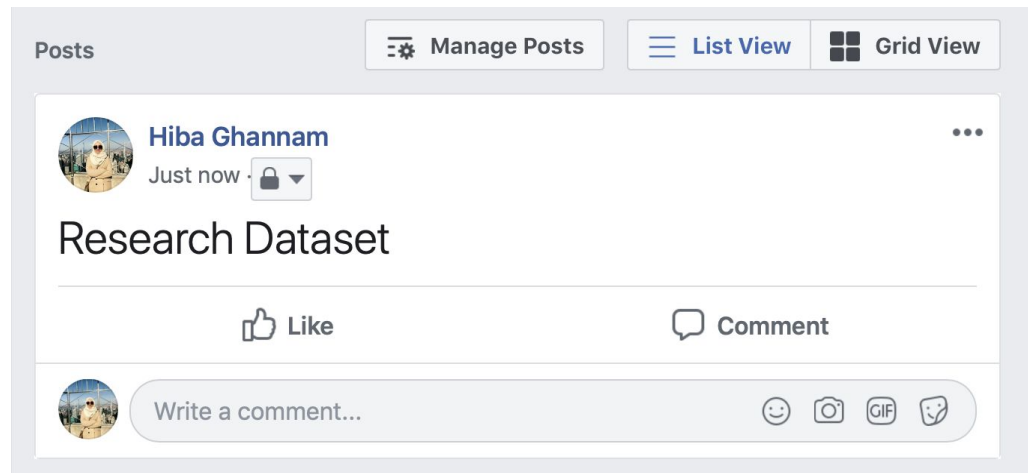


Figure 4.4 Facebook Component - View Post

From previous components there are several test cases that have different dependencies, priorities and execution time. In Figure 4.4, there is a test case to verify one behaviour of the “write post” component. In order to verify a text post which is actually a behaviour testing for the “write post” component, you need to click on “new post view” in the empty area. Then after writing the text and clicking on the “Post” button, the text has to be displayed in the “view post” component. Without the existence of these mentioned components, verifying the test case of writing a new text post will not work. Therefore “new pos view” and “view post” components are considered as dependencies for the test case of verifying writing new text post. While writing a new text post itself is connected with testing the behaviour of “write post component”.

The “write post” component itself has a priority which has to be specified by the frontend team as described before. This component is frequently reused in the Facebook app and it's connected with the base functionality for Facebook, so if the targeted release has 50 components, then it's priority is the top one and it's 50.

The test case priority has to be evaluated by the test engineer depending on the business needs for the test case. In the example since the test

case is a basic one for facebook business it was evaluated as highest. The test engineer also has to evaluate the execution time for the test case including test case input data, so in the example it's evaluated as 5 minutes.



Figure 4.5 High Priority Test Case Properties sample includes dependencies

- **Example 2:** another simple example for a high priority test case that doesn't have any dependencies, verifying the username displaying in the user profile. It's a too simple test case but it's an important one for business. If we consider the targeted release doesn't include the registration, then there are no components dependencies for this test case. Also, developers don't need to build any components for this test case, so it's not connected for any custom component, it's a simple label which is a builtin component and developers just need to use it. Figure 4.5, represents simple test case properties from the dataset.

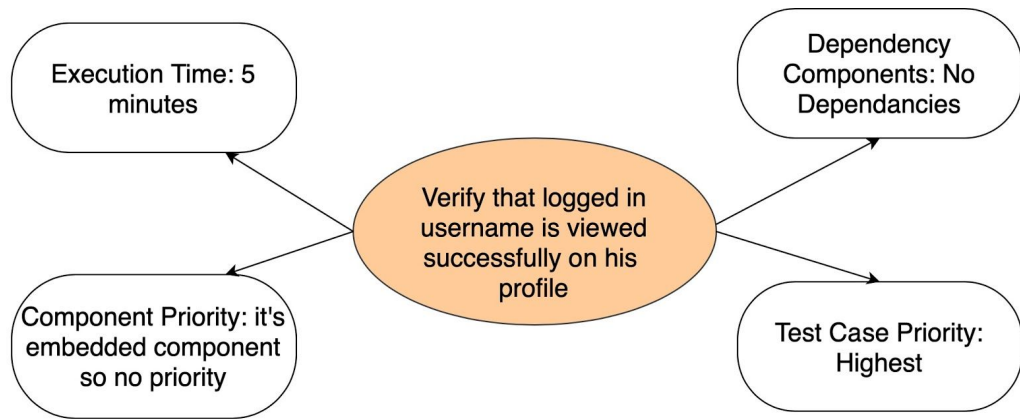


Figure 4.6 Simple High Priority Test Case Properties sample that does not include dependencies

4.2 Solution Design

The competition between the first two objectives or research questions have been resolved by genetic algorithms in this phase. The objectives are maximizing the coverage of test cases that have high priorities and at the same time increasing the coverage of the whole test case. These two objectives have to be done within a specific time, this means when the decision maker wants to increase the coverage of important test cases, this may consume more time. At the same time, this leads to having less time for doing testing of other test cases that have lower priorities. To achieve the previous objectives, the mentioned test cases properties could be used with genetic algorithms. In the generated chromosome that represents the solution, different test cases might have higher priorities than test cases of dependency components that are important for testing other test cases. Therefore, one important objective is to minimize the violation of these test cases that have higher order than important dependencies. After implementing the solution using the specified algorithms, the best execution time for the algorithm will be selected as the final solution. Figure 4.2, represents the work that has to be done in phase 1.

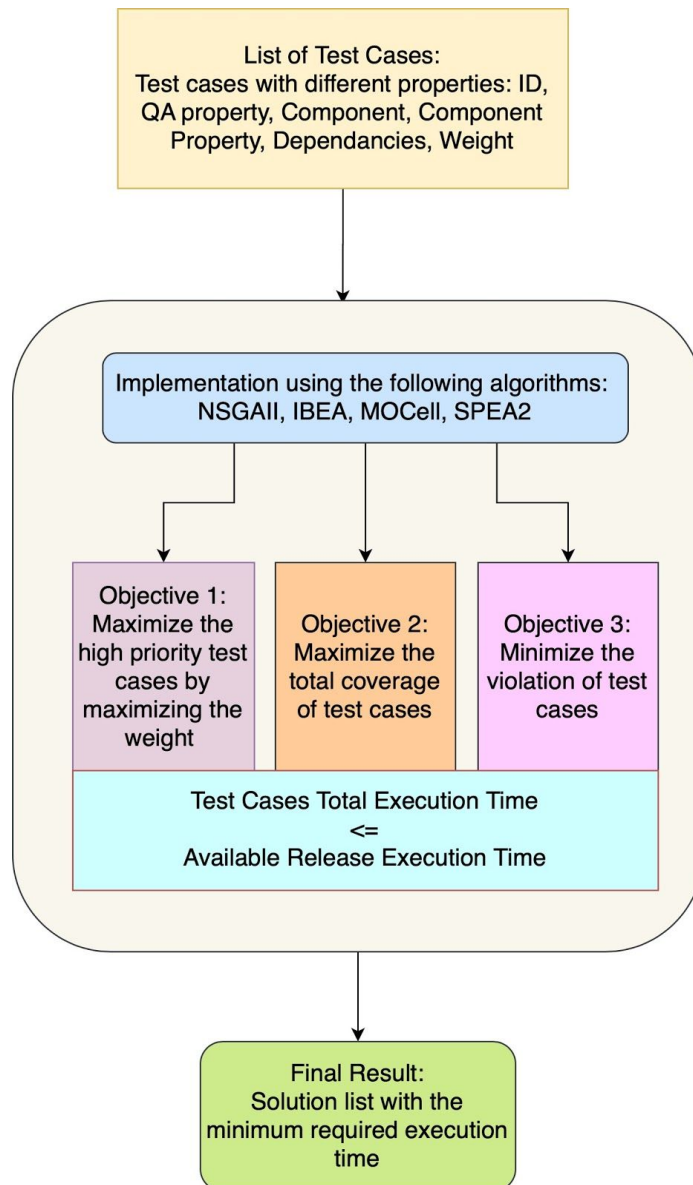


Figure 4.7 Phase 1 objectives and solution result

Since Each test case has components, dependencies and priorities, then a weight for each test case could be calculated by summation of the previous values. When the test case has more dependencies, this necessarily increases the test case weight and it's an important indication for the final test case priority. As a result, the high priority test case has high weight, while the low priority test case has low weight. Therefore a new weight value was added to each test case in the dataset. To illustrate the weight value, by returning back to Figure 4.4 in example 1, the following properties were collected:

- Test case manual priority: highest

- Component priority: 50
- Dependencies: “New Post View” and “View Post” components

To calculate the weight, the manual priority component has to be converted to a numeric value. Therefore since there are five values for manual assigned priority for test cases, they have a numeric values as the following:

- Highest=5
- High=4
- Medium=3
- Low=2
- Lowest=1

By return back to example 1, if the assigned priority of “New Post View” is 49 and “View Post” Component has 48, then the weight value for this test case will be:

- Weight = Test Case manual Priority (TC P) + Component Priority (CP)
(4.1)

Where:

- Test Case manual Priority (TC P): is a priority given by a test engineer who analyzed the requirements and wrote the test case. It’s an indication of how much the test cases are important to business requirements. The test engineer here doesn’t have to know any technical knowledge to specify the priority.
- Component Priority (CP): it’s a priority that has to be specified by frontend engineers who analyze the requirements from UI perspectives. This priority is an indication of how much this component will be reused in the system and how much this will open different areas for requirements implementation.

And by using the previous values as examples it will be as the following:

- Weight= 5 + 50 = 55

On the other hand, for the test case in example two which was represented in Figure 4.5, there is no component so the weight will be as the following:

- Weight = 5 + 0 = 5

Answering the first question which is connected to increasing the high priority test cases, means increasing the total weight for a subset of test cases. So That, with a limited time of sprint or release, the ability of increasing the total weight reflects how the coverage of high priority test cases looks like. Since each test case has expected execution time, then increasing the total weight will be limited by the execution time of test cases. On the other hand, answering the second question could be done by increasing the total number of test cases that are going to be executed in the coming release. Increasing the number of test cases also will be restricted by the total expected execution time of the selected test case subset. As a result, a competition between increasing the total count of test cases and increasing the total weight of selected test cases in a specific time.

4.2.1 Chromosome Representation

The required solution for prioritization is a list of test cases that have an order that indicates the priority. This means, each test case ID is an identifier for the test case, while it's order is enough to represent the priority. Therefore, chromosome structure has been represented by a list of test cases with a specific order, while each test case has been considered as a gene as represented in Figure 4.6. Each gene in the chromosome has two properties that could be used to simulate test cases properties as the following:

- Content: it can be used to represent the test case ID that has to be assigned by the testing engineer to each test case in the test suite. Test case ID is an integer number and it's usually an index which has to be incremented by 1 for each new test case. So That in the generated dataset for this research, the first test case ID is 0, the second is 1 and so on.
- Index: it simply represents the gene location in the chromosome, therefore it is used to represent the test case order in the chromosome. As a result. This order is considered as the priority of the test case in the generated solution.

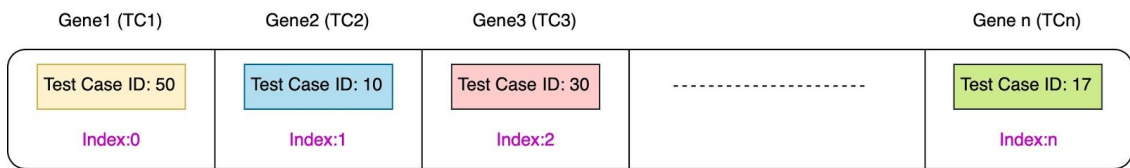


Figure 4.8 Chromosome Structure Representation

The final solution is a new sequence of test cases, they have to be ordered in a way that has the highest possible total weight of test cases and the highest total count of test cases within a limited release time.

4.2.2 Objective Functions

Since there is a competition between two clear objectives then the generated solution by any selected genetic algorithm have to achieve the following

1. Maximize the number of high priority test cases depending on the available release time and by calculating a test case weight. This means increasing the total weight has to include the test cases with high weights, more test cases from high weights means more test cases number from high priority test cases. And this answers the first research question.
2. Maximize the number of test cases in the final solution. The test cases are represented by genes, while the solution is a chromosome, therefore the algorithm has to increase the number of genes in the chromosome. Since test cases have not to be duplicated in execution, then the generated chromosome has to have unique genes or unique test cases depending on the available release time. Achieving this goal means increasing the total coverage of test cases, which is the second objective in this research and it answers the second research question.
3. Minimize the violation in test cases that have dependencies in the generated solution, therefore once finding a test case in the

chromosome that has dependencies with high priorities, and this test case order is before that priorities, remove that test case.

Each release of software has a list of new test cases that have different priorities and they have to be tested for the first time. Then each release r has n number of test cases TC , therefore test cases set per release is represented by the following:

$$TC_r = \{TC_1, TC_2, TC_3, \dots, TC_n\} \quad (4.2)$$

Where:

- r , it's the targeted release which has the prioritization problem.
- TC_r : it's the list of new test cases in the targeted release where each one has some properties.
- TC_1 : it's the first test case that has index 1, where the index is an ID which has to be assigned for each test case by the testing engineer.
- TC_n : it's the last test case in the targeted release r .

Each test case has an expected execution time TC_t , and the release itself has an expected deadline that adds some limitations and restrictions on the test cases total execution time. Achieving the first objective by increasing the high priority test cases coverage means increasing the total weight of test cases as mentioned in previous sections. The total weight for any subset of test cases is represented by the following function:

$$TCTW = \sum_{i=1}^k TCW_i = TCW_i + TCW_{i+1} + \dots + TCW_k \quad (4.3)$$

Where:

- $TCTW$: it's the summation or the total weight for a list of test cases that have to be executed.
- i : is the first test case in the list.
- k : is the last test case in the list.

Since Each test case also has an expected execution time, then the total execution time of all test cases is the summation of all test cases TCTT, it's represented in the following equation:

$$TCTT = \sum_{i=1}^k TCTi = TC1 + TC2 + + TCk \quad (4.4)$$

As a result, the first objective could be represented by the following:

$$TCTW = \sum_{i=1}^k TCWi, TCTT \succ Rt \quad (4.5)$$

Where:

Rt: is the release expected time.

Therefore the above function can be used to specify the list of test cases that have the maximum total weight without exceeding the release time.

On the other hand, Increasing the total number of test cases is the representation of increasing the test cases coverage. Regardless of the percentage of the high priority test cases in a selected subset of test cases, the total number of the count of test cases in any subset is represented as the following:

$$TCC = |TCs| \quad (4.6)$$

Where:

- **TCC**: it's the count or the total number of test cases in a test cases subset.
- **TCs**: is a test cases subset.

Now the second objective that answers the second question by increasing the coverage of total test cases is represented by the following by considering the release time:

$$TCC = |TCs|, TCTT \succ Rt \quad (4.7)$$

The above equation means that increasing the total number of test cases necessarily increases the coverage, but there is a limitation in release time. This means, decision makers may exclude some minor test cases in case there is some delay that might prevent them from being on time for release

date. Therefore increasing the coverage or the count of test cases, has not to exceed the release time.

Achieving the first objective by increasing the total weight of test cases, most likely means more execution time. The reason is that the total weight for each test case comes from business and technical components. Business priority is represented by testing priority, while the technical is given by considering the component priority. Therefore the total weight will be high and these test cases might need more time for execution, preparing data and using the system for the first time. Adding more steps for test cases means more dependencies have to be considered and this will also increase the time for high test cases. Therefore high priority test cases most likely will consume more time in execution. This will lead to a competition with the second objective. Because increasing the count of the total number of test cases in the selected subset, means picking the test more test cases with low execution time that will not break the release date or time. At that moment exactly, decision makers need to compromise or decide what to do. And this situation is repeated frequently in different situations when there is a long list of test cases, restricted time and different obstacles. All of this may lead to consuming more time in unexpected steps, then this will reduce the rest of time that will be needed for testing.

4.3 Random Datasets

Four datasets were created randomly in order to use them later in the experiments phase. The purpose of creating them is having different datasets with different sizes and values that may affect the algorithms. So creating random datasets was an easy way to have any size with different fields values. Dependencies, test case description, test case priority and component priority were not specified. Instead, only three fields were specified for each test case: ID, Weight and Expected execution time. The reason for creating those values only is related to how the fitness functions will be calculated later. Regardless how the weight came from, it will be required later for creating the solution.

Therefore all random datasets could be considered as the final format for the required dataset to run the algorithms. Considering the Facebook dataset, five datasets will be included with the following sizes: 163, 400, 600, 800 and 1000 test cases. All datasets are available on the attached link in appendix.

To illustrate the content of the random dataset, Figure 4.9 displays a screenshot from 400 test cases dataset, while Figure 4.10 displays it from 1000 test cases dataset.

f_x	A	B	C	D
1	0	87	15	
2	1	76	38	
3	2	176	21	
4	3	131	22	
5	4	220	38	
6	5	140	30	
7	6	349	23	
8	7	68	14	
9	8	279	11	
10	9	69	35	
11	10	375	33	
12	11	46	12	
13	12	317	36	

Figure 4.9 Sample from a random dataset with 400 test cases

	A	B	C	D
976	975	219	25	
977	976	313	40	
978	977	101	37	
979	978	322	17	
980	979	15	36	
981	980	599	16	
982	981	379	28	
983	982	510	31	
984	983	362	14	
985	984	148	24	
986	985	470	28	
987	986	280	32	
988	987	202	16	

Figure 4.10 Sample from a random dataset with 1000 test cases

As we can see from both Figures 4.9 and 4.10, the dataset contains only three columns. First one is A and it represents the test case ID, therefore you can see IDs from 0 to 12 in sample 4.9 for 4000 test cases datasets. While Figure 4.10 displayed a sample with IDs from 975 to 987 from the 1000 test cases dataset. The second column B represents a random number for the summation of the test case priority and the component priority. It was represented in the facebook dataset as Weight. Therefore, here instead of having separated columns for those values, a final random number is generated as simulation. This number was generated by the available random function in google sheets. For each dataset a min and max number was specified and a random number was generated between these edges. For example, to the 1000 test cases dataset, random values were created between 10 and 600 as described in the following screenshot.

fx		=RANDBETWEEN(10,600)	
	A	B	
970	969	48	
971	970	488	
972	971	333	

Figure 4.11 Generating the random values using random function

The same way was applied to generate weight values for other datasets, the difference is the min and max values for each dataset.

The third column in random datasets C is the expected execution time for each test case. It was also generated randomly using the same function of generating the weight. It just considers the min and max difference between weight and time, therefore it was selected to be between 10 and 40 for creating the time. This way will help to produce any required size for testing anytime. It also helps to increase diversity in the problems by generating different values of weight and expected time.

4.4 Development Environment

JMetal framework will be used for this research implementation, it's a java based framework that is used for solving optimization problems [63]. It supports different metaheuristics algorithms like NSGA-II, PEAS, MOCell and others [63]. Version 5.6 was selected to implement the solution. IntelliJ IDEA will be used as IDE and the following algorithms will be used to implement the solution in JMetal:

- NSGA-II
- IBEA
- MOCell
- SPEA2

Chapter 5. Experiment Setup and Run

This chapter illustrates the whole details about experiment setup by jMetal framework. It introduces the research test cases prioritization problem definition that has to support multi-objective optimization, and this means fitness functions that were illustrated in the previous chapter have to be defined in detail here. The Test cases prioritization Problem definition section also illustrates how to read the dataset with a specific form, where that form is required to build the test cases prioritization problem correctly.

The next important thing for experiment setup and run is calling the selected algorithms. This chapter illustrates how to use the required algorithms that are supported by jMetal framework, it illustrates the algorithms settings and how to connect them with problem definition. Calling the required algorithms needs to define a specific data type and to do some modifications to support this research test cases prioritization problem, all of these details have been introduced in this chapter.

As described in previous chapters, the third objective after doing optimization is minimizing the violation of test cases for the generated solutions. This means, if dependencies come after the required test case in a generated solution, there is a violation. An algorithm was built to do this and this chapter illustrates its implementation details.

5.1 General Design Structure

jMetal framework supports defining any new problem, regardless its single objective or multi objective optimization problem the same structure is used. For the test cases prioritization problem implementation, defining the problem is the first step to build the solution. To define a problem, a new class has to be created, it has to read the dataset in the correct format and then it has to convert it to a form that is needed to work with jMetal structure. It also defines the number of objectives and how to calculate the fitness functions, all

of these details are illustrated in the problem definition section. For this research experiment, a problem definition with class name “TCP” was created under multiobjective problems directory in jMetal project as described in Figure 5.1.

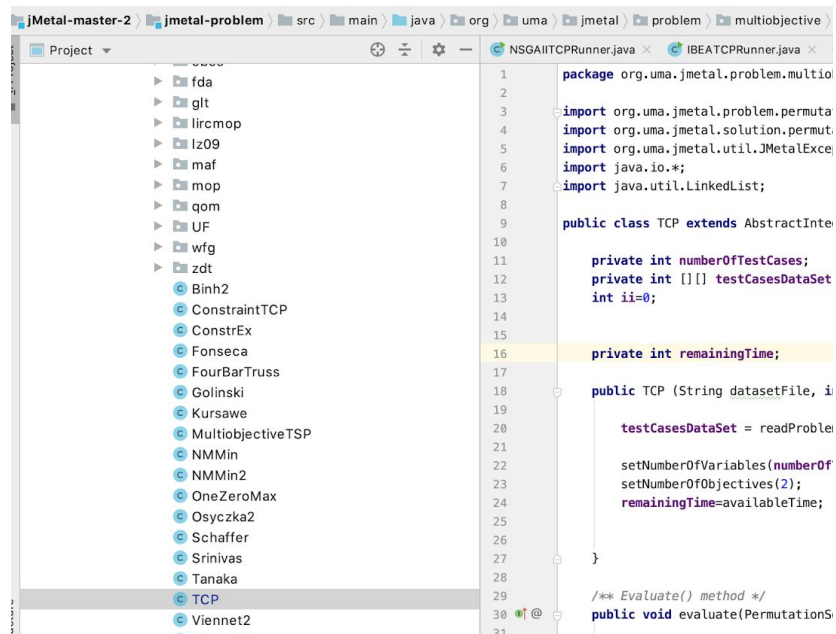


Figure 5.1 TCP problem definition class

After building the problem definition, new classes have to be created for each genetic algorithm that is needed to be supported. These classes are known as algorithm runners, each algorithm has its own runner, while each runner can call any problem and can use any genetic algorithm. In this research, six algorithms runners were built for six genetic algorithms, Figure 5.2 shows the runners' classes names. All runners call the same prioritization problem TCP.

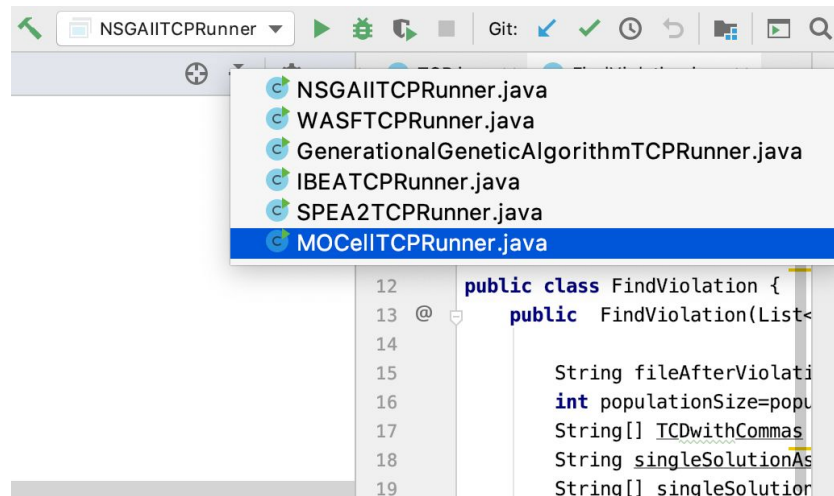


Figure 5.2 Algorithms Runners' Classes

For finding violations of test cases in the generated solutions, after running each genetic algorithm, a violation algorithm has to be called. Therefore a new class was created to implement this algorithm, this class was called Find violation and it's all related details have been illustrated in coming sections.

5.2 Problem definition

Since the research contributes to provide the best prioritization for decision makers, then all test cases have to be included at least once without repetition per solution. Therefore permutation is selected to define the test cases prioritization problem (TCP). As described in chapter 4, the chromosome content includes the test cases IDs, and in the generated dataset, test case ID is integer number. As a result, the TCP problem is considered as an Integer permutation problem. jMetal framework supports these kinds of problems definitions, so in this experiment design, the TCP problem definition class extended "AbstractIntegerPermutationProblem" abstraction class.

The problem definition has to define the number of variables and to set the number of objectives. Since the research problem has two competitive objectives then, the TCP class has to set two objectives. On the other hand, TCP has to read the dataset and evaluate the generated solutions per

generation. Reading the dataset has to be used to set the number of variables represented on a screenshot from TCP constructor in Figure 5.3.

```

18 public TCP (String datasetFile, int availableTime
19
20     testCasesDataSet = readProblem(datasetFile) ;
21
22     setNumberOfVariables(numberOfTestCases);
23
24     setNumberOfObjectives(2);

```

Figure 5.3 setting the number of variables and objectives

Two main methods were built to read the dataset and evaluate solutions inside TCP class as the following:

- A. Read Problem: it reads the Dataset and converts it to the required form for fitness functions later. As described in chapter 4, test cases fitness functions require their weight. On the other hand, expected execution time per test is required for stopping criteria. Therefore the experiment dataset was designed to include all mentioned values. For reading the TCP problem with the required data, a new csv file was extracted from the original dataset with three columns: Test Case ID, Test Case Weight and Test Case expected Execution Time, Figure 5.4 shows a dataset sample from the extracted file.

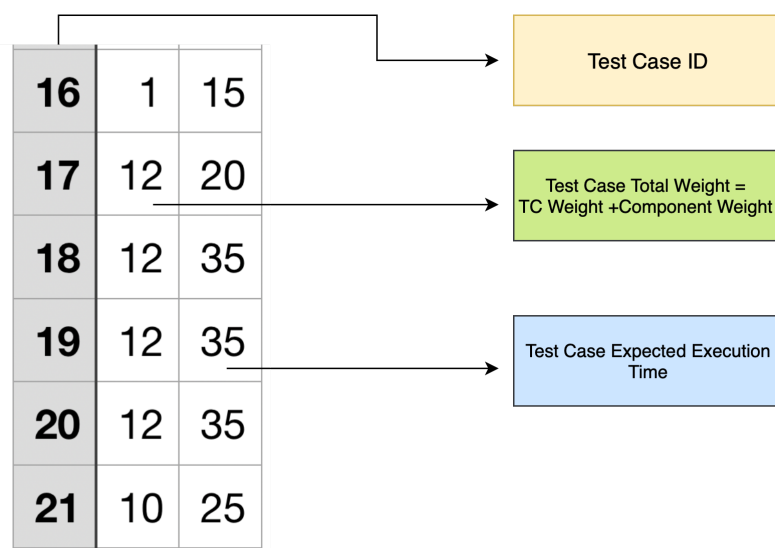


Figure 5.4 Dataset sample from the extracted input file

Once the Read Problem method, reads the file, it defines a multi dimensional array with three columns and dynamic number of rows. Three columns represent the three values from the Dataset csv file. The Array stores the same values with the same order from the csv file as the following: column 0 is used for Test Case ID, column 1 is used for Test Case weight and column 2 is used for expected Execution Time. On the other hand, the number of rows for the dataset tarray represents the number of test cases, therefore it's a dynamic number depending on the number of test cases that were added to the dataset file. The method at the end returns the dataset array which has been used for generating solutions and calculating the fitness functions later. Figure 5.5 shows a screenshot for that method form TCP problem class.

```
104 private int[][] readProblem(String file) {  
105
```

Figure 5.5 Signature of Read Problem method in TCP problem

The number of test cases that represents the number of rows on the returned array, was used as it's described in Figure 5.3. It sets the number of variables for genetic algorithms that are used later to find solutions.

- B. Evaluate: the second in TCP problem definition class is evaluation method. It's used to calculate the fitness functions for solutions per generation. The TCP problem was considered as "IntegerPermutationProblem" and solutions have to be declared "PermutationSolution<Integer>". This supports representing the solution as a chromosome with integers values that represent Test Cases IDs. As a result, the evaluate method was designed to accept the

“PermutationSolution<Integer>” solution as a parameter as described in method signature in Figure 5.6.


```
31  
32  public void evaluate(PermutationSolution<Integer> solution){  
33
```

Figure 5.6 Signature of Evaluate method in TCP problem

For the TCP problem, since there are two objectives and they were set in TCP class constructor, then two fitness functions have to be calculated per solution. Both optimization objectives were declared in chapter 4 with fitness functions. The first one was about increasing the total weight of high priority test cases within a specific period of time, while the second objective is to increase the number of test cases within the same specific time. The Evaluate method calculates the first objective by fitness1 and it uses fitness2 to calculate the objective 2. To illustrate the evaluation algorithm and how it calculates the fitness functions, Figure 5.7 represents the flow.

From Figure 5.7 the evaluation method could be represented by the following steps:

1. For each solution, start a for loop from the first test case. Each test case is represented by a gene, so each gene has the test case ID and it has an index which is represented by loop iteration.
2. Check the remaining time that represents the available release time for testing. If adding a new test case to calculate fitness breaks that remaining time, the test case will not be added to the fitness. For calculating the breaking condition, use the expected execution time for each test case. It was stored in the dataset matrix that was described in the Read Problem method.
3. In case there is no breaking for the remaining time, start updating the fitness functions.

4. For the first objective increase the fitness1 by adding a new test case weight. It was also stored through reading the problem in the dataset matrix, inside the second column.
5. Update the second objective, which is calculated by fitness2. To update fitness2, only increment the test cases number by one. This means more test cases are included in the solution with a specific priority.
6. Return to step 2.

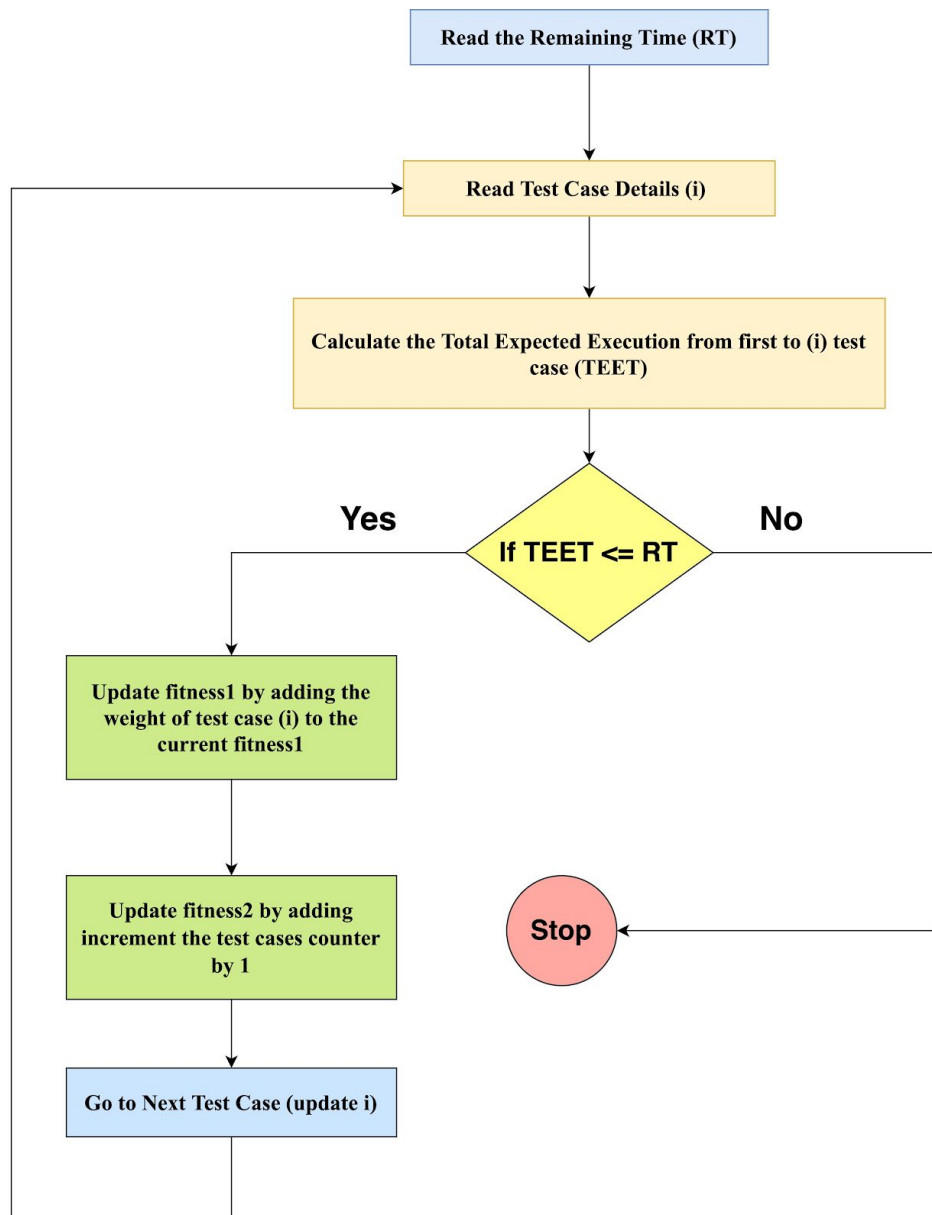


Figure 5.7 Evaluate Algorithm Flow

7. If there is a time breaking on step 2, leave the loop and get the final values for the fitness values.
8. After having the final value for fitness functions, set the solutions objectives.

To illustrate the above algorithm for the evaluation method, let's take an example. In Figure 5.8, NSGA-II was executed and all solutions from all generations were printed inside the evaluation method. The remaining time was set to 70, this means for each solution, the evaluation method keeps adding test cases to fitness functions until breaking the remaining time value. Exceeding 70, will not add the test case weight to fitness1 and will not increment the fitness2 that refers to test cases counter.

Referring to solution 0 in Figure 5.8, fitness1 is 200. This means the high priorities test cases that were included in evaluation have a total weight of 200. While for fitness2 value is 10, this means the first 10 test cases were included in the fitness value without breaking the remaining time. Therefore, if you check the variables on the same Figure 5.8, you will see that test cases from variable 68 to 128 were included in the fitness functions values. With variable 89 that represents test case ID 89 and which is next one to 128, the time was broke. Referring to execution times that were printed for solution 0, we can see that time before breaking the remaining time is 57. On the other hand, the specified execution time of test case 89 in that run was 17. Including test case 89 leads to 72 as total expected execution, and this number will break the remaining time which was set as 70. As a result, test case 89 was not included on fitness functions values.


```

Run: NSGAIITCRunner x
/Library/Java/JavaVirtualMachines/jdk-13.0.1.jdk/Contents/Home/bin/java .
* solution 0 *
Fitness1 - solutionIndex: 200
Fitness2 - solutionIndex: 10
Variables: 67 70 134 49 59 42 39 97 33 128 89 73 56 66 71 6 117 84 7 48 54

Expected Execution Time before breaking :57
Expected Execution Time at the moment of breaking the remaning time:17
Remaining time is broken by this Test Case: 89

-----

* solution 1 *
Fitness1 - solutionIndex: 120
Fitness2 - solutionIndex: 6
Variables: 48 104 111 121 78 133 138 89 97 67 11 0 45 5 32 18 28 50 140 87

Expected Execution Time before breaking :68
Expected Execution Time at the moment of breaking the remaning time:3
Remaining time is broken by this Test Case: 138

-----

* solution 2 *
Fitness1 - solutionIndex: 155

```

Figure 5.8 Solutions from several generations with single Run

Referring to solution 1 in Figure 5.8, it displays how the weight increased to 120 therefore fitness1 has higher than solution 0. On the other hand, fitness2 is 6 which is less than fitness2 of solution0. This means test cases execution time was not broken from variable 48 to 133, total execution time was 68 until test case 133. Once the expected execution time was considered with test case 138, remaining time was broken. That was related to adding 3 as expected execution time for test case 138 to the total expected execution time for all previous test cases. Total number will be 68+3 and 71 breaks the remaining time which was specified as 70. As a result, test case 138 was not included in evaluation.

5.3 Algorithms Runners

In the design structure section, the second main class that was defined is algorithm runner. By runner class, problem definition and genetic algorithm builders are called. In this research design, each algorithm has its own runner, therefore any genetic algorithm has its own settings and setup.

Related to the problem type that has to support integer dataset and to include all test cases in the resulted solution, then “<PermutationSolution<Integer>” was used for all genetic algorithm runners classes. Several parameters had been declared as a setup for each required genetic algorithm runner with the mentioned data type. For all mentioned algorithms in this research, several parameters have been initialized inside the runners classes as the following:

- Crossover: PMXCrossover was used to initialize the crossover parameter. The reason behind this choice is the problem type that needs integer permutation. As described before, the genetic algorithms for the TCP problem have to create a solution that includes the whole test cases in the dataset. At the same time, the genes in the generated solutions have to be represented by integer values that represent test cases IDs. PMXCrossover supports integer permutation therefore it was selected to fit the problem type.
- Mutation: PermutationSwapMutation was selected where two positions for two selected genes will be swapped. Permutation was selected to support the problem type that was clarified previously.
- Selection:
BinaryTournamentSelection<PermutationSolution<Integer>>(new RankingAndCrowdingDistanceComparator) was selected. It ranks the individuals with relative ranks and then it will select the one with best rank.

To run any genetic algorithm in jMetal, population size and max evaluations values have to be defined. In addition, some genetic algorithms need to define the archive size. Inside the runner class for each algorithm these values were initialized in addition to other algorithms parameters for initial runs. Table 5.1 represents the whole settings that were needed to all algorithms.

Algorithm Parameter	Value for initial run
Crossover Type	PMXCrossover
Crossover Probability	0.9
Mutation Type	PermutationSwapMutation
Mutation Probability	0.2
Selection Type	BinaryTournamentSelection
Population Size	100
Max Evaluations (Iterations)	500
Archive Size	100

Table 5.1 Initial Runs Setup

Using the above configuration, required genetic algorithms can be built using jMetal by algorithm builder class. So in each algorithm runner, the algorithm was defined depending on the algorithm builder signature. As NSGA-II works, it doesn't use an Archive, therefore no archive size parameter on its own builder as described. While for IBEA, it uses Archive therefore archive size was added to its builder as parameter. So each algorithm defines their own and required parameters.

All genetic algorithms that were selected for this research in jMetal support integer permutation that was selected for the TCP problem. On the other hand, not all builders are dynamic to accept any type, IBEA builder was one of them. Therefore, different modifications were done on IBEA builder to support integer permutation.

After running each genetic algorithm, the resulted solutions will be printed on a csv file Figure 5.9 shows.

129	104	35	43	38	19	37	16	79	27	26	15	51	100	0	123	85	20	108
50	51	150	57	143	111	79	147	34	120	13	127	25	118	44	124	72	100	84
146	64	32	27	79	107	44	144	34	55	135	143	106	128	31	50	61	92	149
3	89	105	130	93	148	44	10	85	117	111	107	53	147	139	4	74	14	125

Figure 5.9 Four solutions were resulted and printed from running an algorithm

In the above Figure 5.9, NSGA-IIRunner was executed with the initial parameters, the result was four chromosomes or solutions from that run. NSGA-IIRunner printed these solutions on a csv file, the screenshot includes only parts of the included test cases per chromosome or solution. Each cell in the csv file presents a test case id for the generated solution, it's also considered as a gene in the resulting chromosome.

After running any algorithm class, the generated solutions will be plotted to see the pareto front. For the previous run in Figure 5.9, the pareto front plot is shown in Figure 5.10.

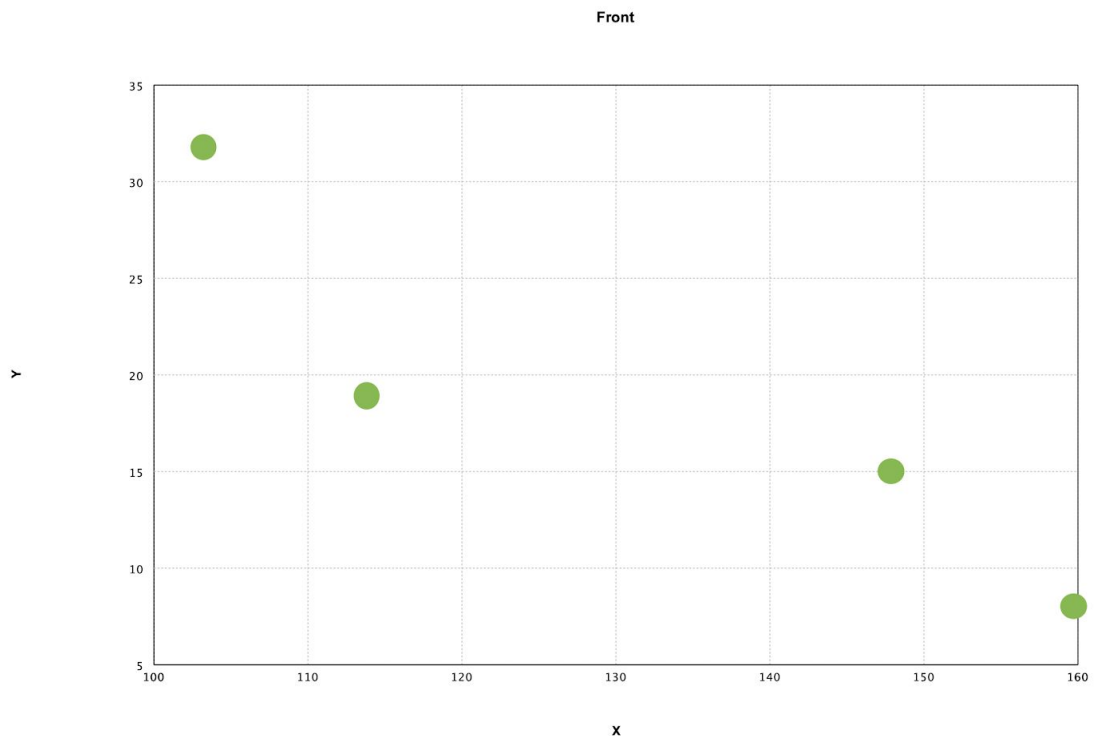


Figure 5.10 Pareto Front for the generated solution in Figure 5.9

Generated solutions from executing genetic algorithms have to be passed to the violation algorithm. The implemented Violation algorithm will give the chance to decision makers to take the suitable decision they want with the violated test cases. FindViolation algorithm will be discussed in detail in section 5.4. To briefly summarize the flow inside the runners classes, check Figure 5.11.

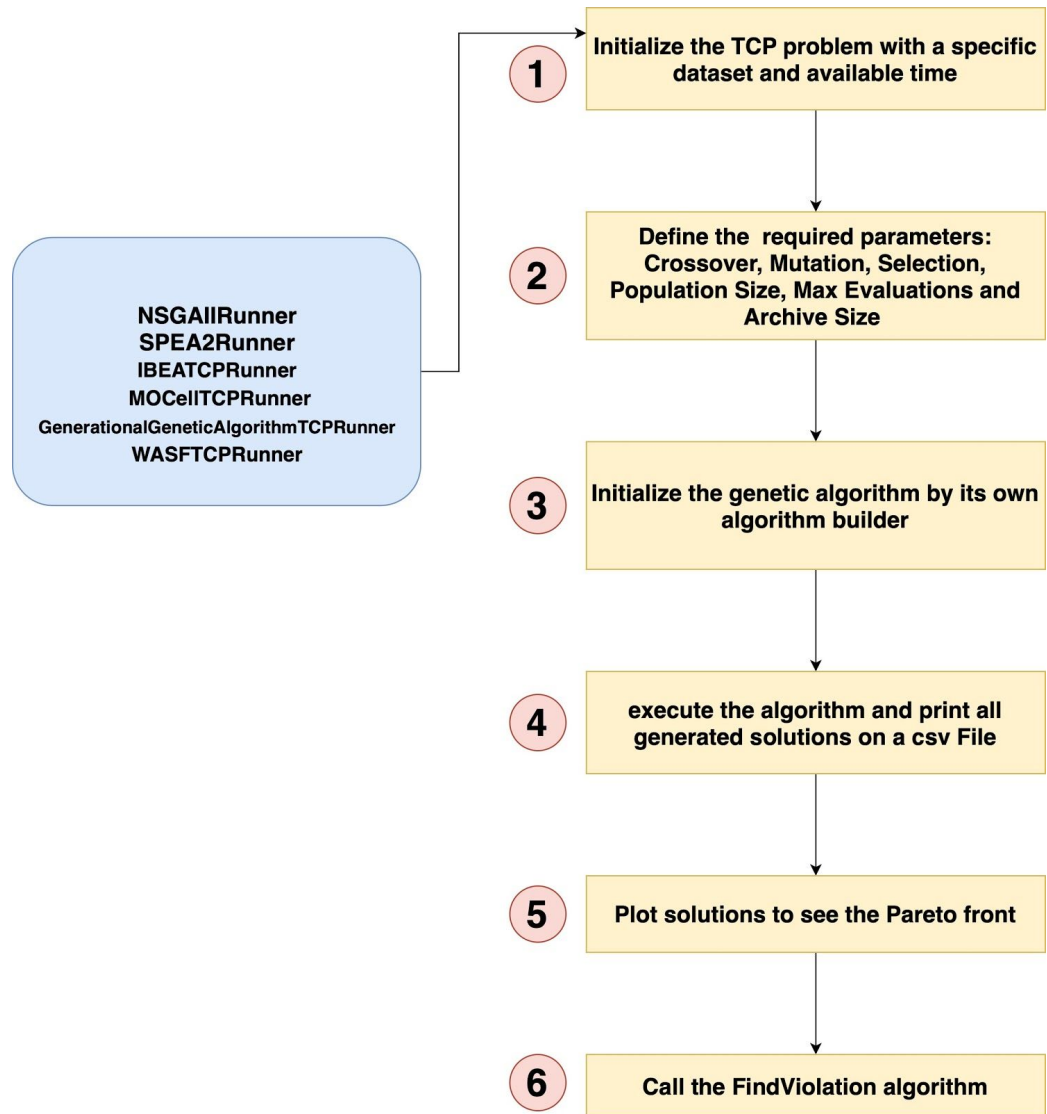


Figure 5.11 Summarization of Runners Classes Steps

All algorithms runners have the same steps, same parameters values for the initial and same dataset csv file. In addition the result has to be printed

on a csv file for each runner, also the result from violation will be printed on a separated file.

5.4 Find Violation Algorithm

Each test case has to be executed in an order that respects the dependencies test cases regardless of the priority. This means, test case TC [i] might have higher priority than it's dependencies from a business need perspective. At the same time, some dependencies have to be tested before the TC[i] to make sure all required components for testing TC[i] are working well. As a result, test case violation is defined when TC[i] has order in the generated chromosome before its dependencies. In other words, for each solution, finding TC[i] with priority that is higher than depanances priorities will cause violation for TC[i].

Find violation algorithm was designed to find violations for any test case in the generated solution. It receives any solution as a parameter regardless of the solutions how it was generated or what was the genetic algorithm. It has to know dependencies for any test case that decision makers have to find its violation. In this research after generating solutions by any runner class for each algorithm, the find violation algorithm was called. The whole generated population that contains all solutions has to be passed as a parameter to the algorithm. The Find Violation algorithm then will write the solutions on a new file with marking the test cases that have any violation. As a result, decision makers will have two files, one for all solutions regardless of the violation. While the second solutions file will contain all solutions with violations.

For this research experiment, and for running a find violation algorithm, a new dataset file was extracted from the original dataset. This file is a csv file that contains only two columns as shown in Figure 5.12. First one is for a list of test cases IDs and the second column contains all test case dependencies that are separated by commas.

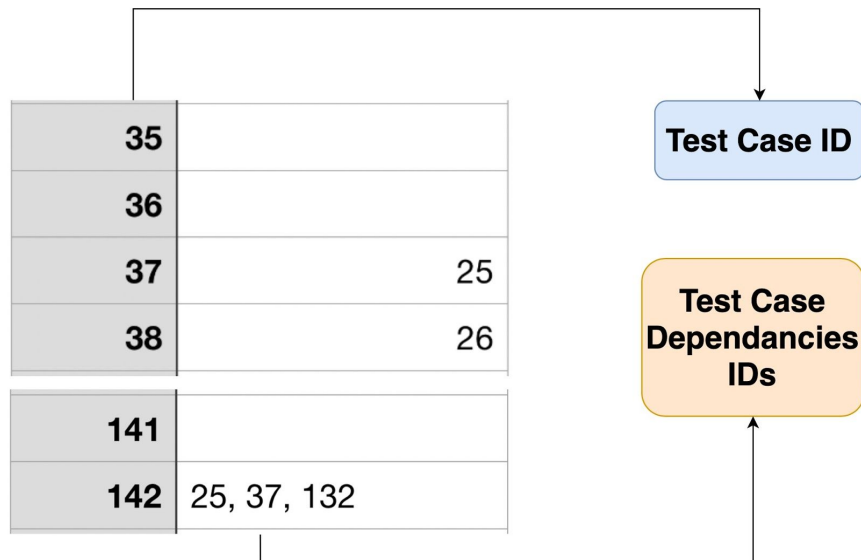


Figure 5.12 Test cases dependencies sample

As shown on previous Figure 5.12, on the first column, the test case with ID 35 doesn't have any dependencies. While for the test case with ID 37, it has only one dependency test case with ID 25. For Test Case with ID 142, it has three dependencies, 25, 37 and 132, they are separated by commas in order to process them later by the find violation algorithm.

To extract the dependencies file, each test case TC[i] from the original dataset has to be checked by its component. If the test case component has any other dependencies components, then dependencies test cases could be generated. By referring to Figure 5.13, test case 142 is related to the component with name "Comment". On the other hand, it has three dependencies components: "New Post View", "Write Post" and "View Post".

TC ID	Component	Dependencies	TC ID	Component
143	141	Verify that Comment component has text editable area, emoji, image	5	1- New Post View 2- Write Post 3- View Post
144	142	Verify that adding text comment with text only will be added with user name and image to comments list for user post	5	1- New Post View 2- Write Post 3- View Post

Figure 5.13 Test Case with three dependencies components from the original dataset

To generate the dependencies test cases, following the test cases that are connected with dependencies components is the key. This step takes more manual time than other steps. It also needs business knowledge from the human resource who is responsible to do it. Moreover, it might be done in case there is a critical need to respect dependencies, hence in some cases testing might be done in parallel from different resources. Therefore, for this research experiment, not all test cases were used to generate dependencies. Instead, the experiment for finding violation was selected to represent the case where there is a need to know only some test cases' dependencies. As a result, the extracted dataset file for finding violation has some test cases with dependencies, while the most test cases does not have dependencies.

After specifying test cases dependencies from the extracted file, the Find violation algorithm works by checking the index of test case TC[i] per solution and comparing it with dependencies indexes on the same solution. Finding any test case index before any dependency, will mark the dependency violation by adding the test case ID that has violation with.

For a specific test case in coming solution, if the test case dependency index (TCDIndex) is larger than test case index, then violation is found. To mark that discovered violation, “VF” text with dependency test case id will be added to the test case that has violation with its dependencies. That way, any test case has a violation with more than one dependency, all discovered violations will be appended to the test case. To illustrate the algorithm by example, for a random run of NSGA-II with default settings, the following 6 solutions in Figure 5.14 were generated without marking any violation.

83	56	53	63	129	48	96	46	138	30	6	55	133	94	58
30	93	111	108	38	76	77	97	39	50	69	29	34	64	27
131	15	55	116	61	91	115	68	130	5	7	37	97	13	32
45	127	120	76	149	150	91	59	92	19	71	117	46	98	27
41	26	94	11	105	76	92	27	104	45	50	87	33	90	111
80	149	39	145	107	60	71	112	99	50	104	137	72	136	73

Figure 20.14 Six solutions from random run without marking violations

For the above solutions, the Find Violation algorithm was executed with the same dataset that was described before. Test cases file with dependencies has test case with ID 39, and 39 has dependency test case with ID 27. For the second solution in the attached screenshot in Figure 20.15, test case 39 comes before test case 27 (which is the last one the attached sample). This means 27 has a larger index than 39, applying the algorithm will mark 39 as violation. The result from executing the Find Violation algorithm is shown in the following Figure 5.15.

83	56	53	63	129	48	96	46	138	30	6	55	133
30	93	111	108	38 VF: 26	76	77	97	39 VF: 27	50	69	29	34
131	15	55	116	61	91	115	68	130	5	7	37 VF: 25	97
45	127	120	76	149	150	91	59	92	19	71	117	46
41	26	94	11	105	76	92	27	104	45	50	87	33
80	149	39 VF: 27	145	107	60	71	112	99	50	104	137	72

Figure 5.15 Resulted Solution with Violations

From the above Figure, the test case with ID 39 was marked with “VF: 27”, this means for test case 39, a violation was found with test case 27. The result was discovered depending on 27 index in the resulting solution. In Figure 5.16, several violations also were discovered and marked, second violation on the same second solution, violations on solutions 3 and 6.

Referring to Figure 5.12, it shows that 38 has a dependency with 26 and this is the form for all other violations in the dataset file that contains dependencies. Following up for this test case in solution 2 that was shown on Figure 5.16, the test case 38 was marked as “VF:26” . In the same figure 5.15, test case 26 doesn’t appear on the screenshot because it has a larger index that was not able to be taken in the same screenshot. On the other hand, for solution 5, test case 26 is the second one in the generated solution as represented on Figure 5.15. While test case 38 comes later like Figure 5.16 shows.

109	146	131	11	28	35	1	37 VF: 25	22	106
131	114	14	73	98	13	82	78	32	109
99	84	127	107	111	74	144	42	60	53
55	90	95	82	66	116	35	70	42	110
108	24	140	39	122	59	132	38	141	88
110	17	61	62	126	41	43	24	96	135

Figure 5.16 Test case doesn't have violation in another solution

From Figure 5.16 we can see how the same test case might have violation on a solution while it doesn't have on another one. The whole idea as discussed before depends on a test case order in the generated solution, or it's a gene index.

Chapter 6. Experiment Results and Analysis

Quality indicators are available for multi objective optimization algorithms to evaluate the generated solutions quality using Preto Front [13]. In this chapter, several experiments have been done in order to measure the quality of each algorithm. Quality has been measured for the resulting solutions by hypervolume and execution time. For doing these experiments and taking measurements, the five mentioned datasets in chapter 4 have been used. Therefore a comparison was done for each algorithm and each dataset, then we can see the impact of increasing the datasets size.

jMetal framework supports calculating several quality indicators, therefore hypervolume values have been collected directly by the available jMetal methods. Other tools like RStudio have been used for data analysis like generating box plots.

In coming sections, comparison has been done for all datasets and algorithms by measuring the HV mean and median. All experiments have the same common setting in the same experiment. On the other hand, all experiments have been done without the violation algorithm. The reason is related to the fact that the violation is not algorithm dependent, in other words, the violation is a common algorithm. Therefore it doesn't make sense to do experiments for genetic algorithms with the violation, or excluding it doesn't affect the values that have to be measured. After measuring the HV for all algorithms and datasets, another experiment is presented for measuring the impact of available time. The target of the mentioned second experiment is to see how the available time which is considered as constraint affects the HV. The Last experiment has been done to measure the minimum execution time that is required to have a stable quality for each algorithm.

Each experiment was done using the five mentioned datasets in chapter 4. These datasets are described in table 6.1 with the number of test cases.

#	Number of test cases
Fcbk 163-dataset	163
400-dataset	400
600-dataset	600
800-dataset	800
1000-dataset	1000

Table 6.1 Datasets list and their test cases

From the above table, we can see that fcbk163-dataset represents the real dataset that was generated from the Facebook web app. While the other four datasets were randomly generated.

6.1 Finding the Minimum Execution Time

The target of this experiment is to find the minimum required execution time for each algorithm. The minimum algorithm execution time could be measured at the moment of starting having stable value of HV. Therefore experiment was designed to measure the HV values on several time stamps on a specific period of time. For each algorithm, time period was specified by milliseconds as a stopping time in jMetal for the algorithms, while the algorithm is working before reaching the time stopping condition, HV values were captured.

For NSGA-II algorithm, it has already two implementations in jMetal, one with max evaluations as stopping condition, and the second one is time as a stopping condition. On the other hands, other algorithms were only implemented with max evaluations as a stopping condition. Therefore, for this experiment, three new implementations have been created to support time as a

stopping condition. All algorithms have implemented a new method for stopping conditions that overrides the original one as described in Figure 6.1 .

```

@Override
protected boolean isStoppingConditionReached() {

```

Figure 6.1 stopping time condition

After implementing the new algorithms that support time as a stopping conditions, a new list of builds were executed to measure the time per each algorithm and per each dataset. For each algorithm, the same settings of previous experiments were used. An additional important common value for all algorithms is the remaining time in problem definition. For each dataset, the remaining time is fixed with all algorithm executions, and the value was selected to be something between a small and large value.

On previous experiments, max evaluations value was fixed to all algorithms by 100,000. While for this last experiment as the max evaluations is not used, the stopping time was fixed to all algorithms as 200 seconds. Therefore HV value was measured during that period on different timestamps as represented in the following list of charts.

1. NSGA-II Results

All HV values were captured in Table 6.2 for all datasets with NSGA-II algorithm. The table shows there is a minor difference between all captured numbers. If we added more focus to the captured values, we can see that the HV for fcbk163-dataset is stable from the first second. While for other datasets, by 30 seconds we can have stable values that are too close together.

Time (s)	Fcbk163-dataset	400-dataset	600-dataset	800-dataset	1000-dataset
1	0.5137	0.5738	0.2970	0.3030	0.3842
2	0.5171	0.6232	0.3332	0.3066	0.4015
3	0.5180	0.6625	0.3700	0.3394	0.4164
4	0.5180	0.6718	0.3954	0.3611	0.4301
5	0.5188	0.6761	0.4141	0.3743	0.4382

10	0.5188	0.6836	0.4371	0.4182	0.4717
20	0.5188	0.6903	0.4570	0.4417	0.4949
30	0.5188	0.6914	0.4622	0.4538	0.5110
40	0.5188	0.6914	0.4631	0.4573	0.5211
50	0.5188	0.6920	0.4636	0.4583	0.5293
60	0.5188	0.6921	0.4643	0.4590	0.5384
70	0.5188	0.6922	0.4649	0.4605	0.5418
80	0.5188	0.6922	0.4651	0.4606	0.5464
90	0.5188	0.6922	0.4655	0.4606	0.5491
100	0.5188	0.6922	0.4655	0.4612	0.5512
110	0.5188	0.6923	0.4656	0.4612	0.5551
120	0.5188	0.6923	0.4656	0.4612	0.5551
130	0.5200	0.6925	0.4660	0.4617	0.5579
140	0.5200	0.6930	0.4667	0.4617	0.5595
150	0.5200	0.6934	0.4667	0.4617	0.5613
160	0.5200	0.6934	0.4672	0.4617	0.5617
170	0.5200	0.6934	0.4672	0.4618	0.5617
180	0.5200	0.6934	0.4672	0.4618	0.5617
200	0.5200	0.6934	0.4672	0.4618	0.5617

Table 6.2 HV over time for NSGA-II with all the research datasets

The result from Table 6.2 and the relationship between time and HV for NSGA-II is displayed by the following figures: 6.2, 6.3, 6.4, 6,5 and 6.6.

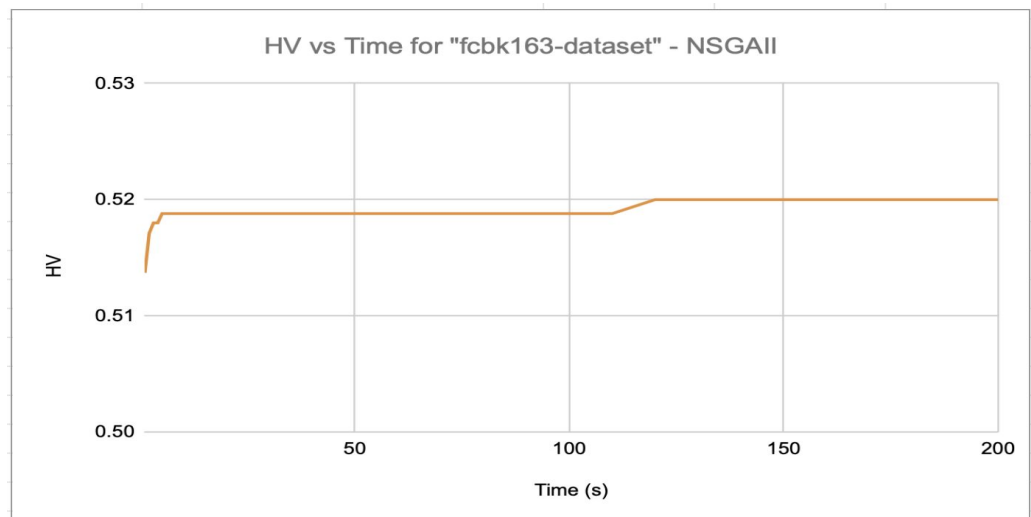


Figure 6.2 HV over time for NSGA-II with fcbk163-dataset

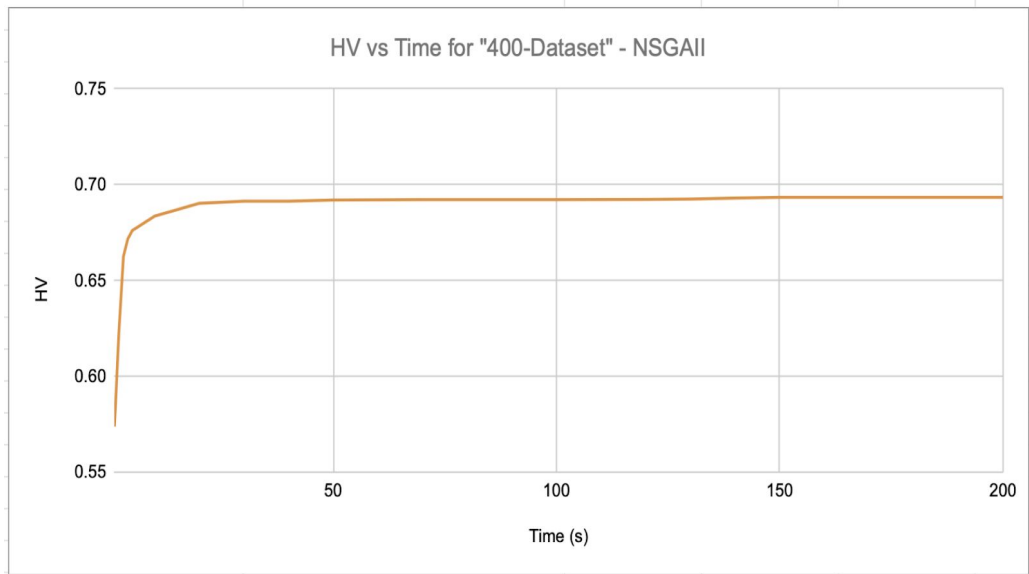


Figure 6.3 HV over time for NSGA-II with 400-dataset

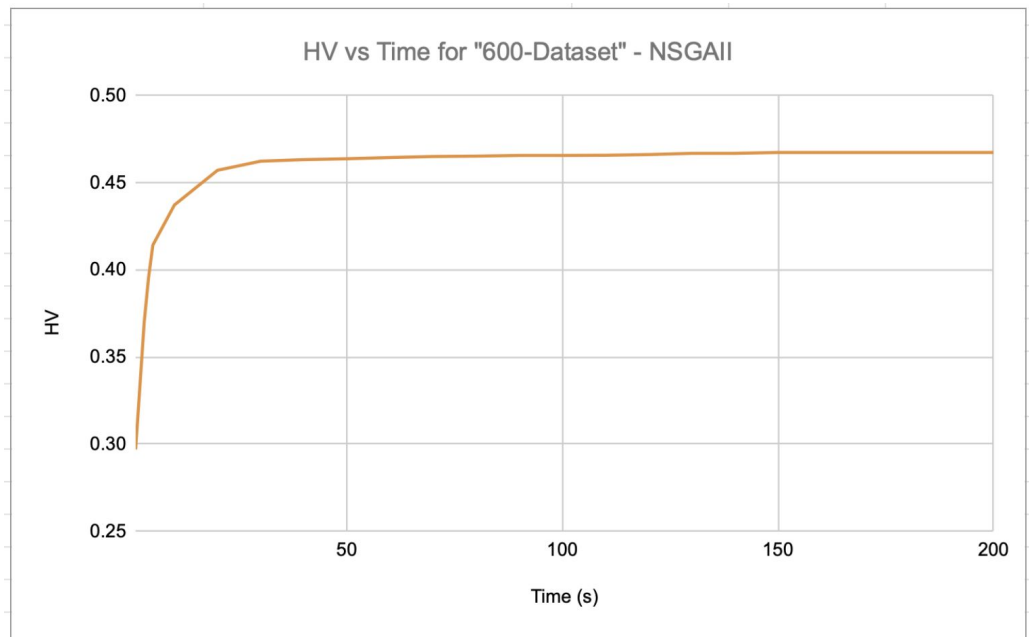


Figure 6.4 HV over time for NSGA-II with 600-dataset

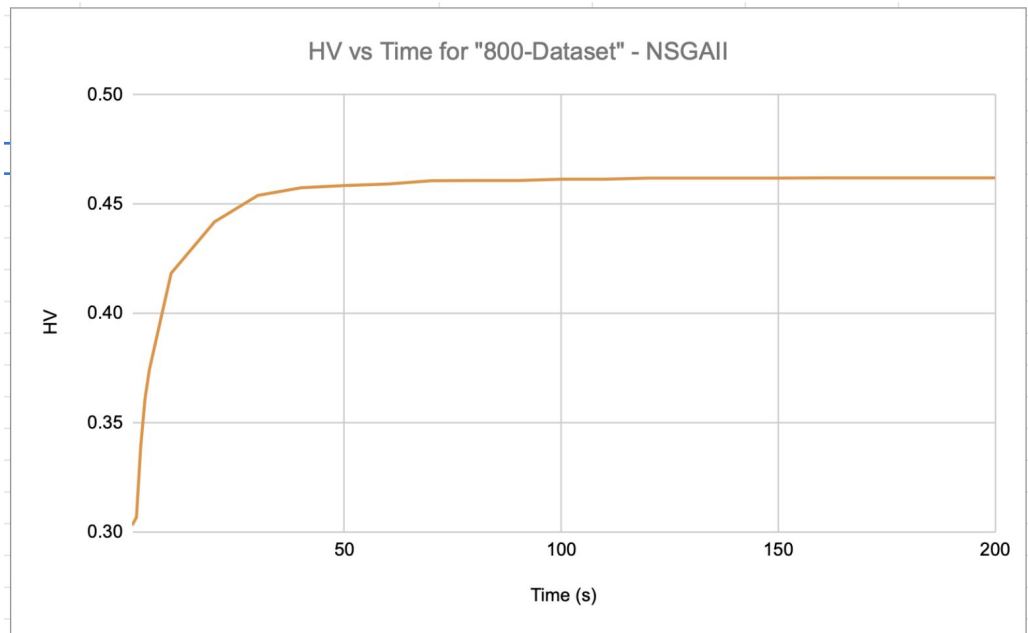


Figure 6.5 HV over time for NSGA-II with 800-dataset

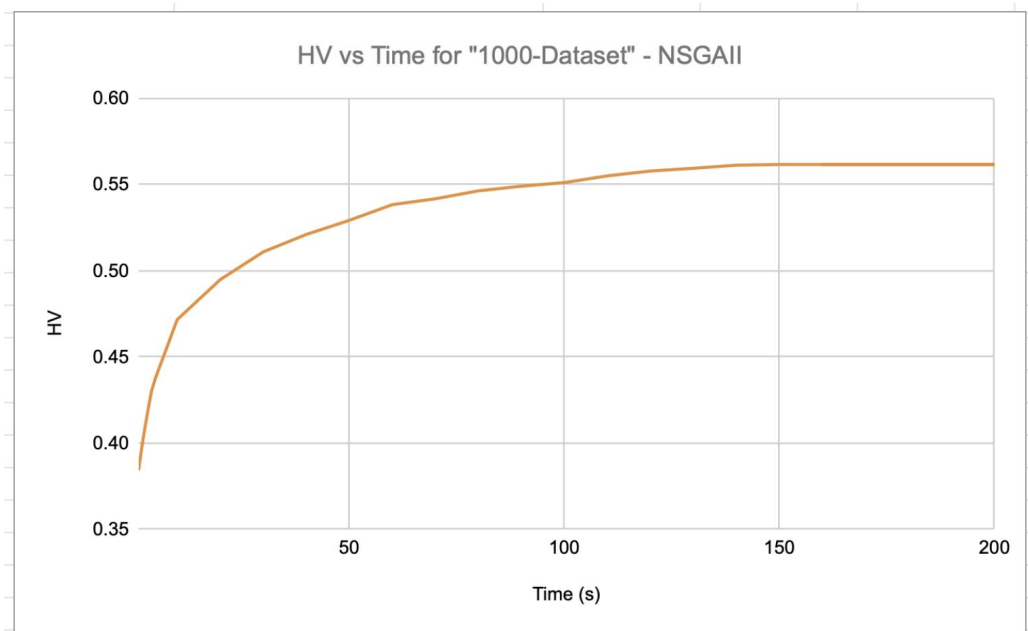


Figure 6.6 HV over time for NSGA-II with 1000-dataset

2. IBEA Results

All HV values for IBEA were captured in Table 6.3 for all datasets. There is also a small difference between values as applied to NSGA-II. Despite the fcbk-163 dataset HV being stable from the first second, the HV

values for all datasets are stable by 30 seconds. The result means that the minimum running time for NSGA-II and IBEA is the same.

Time (s)	Fcbk163-dataset	400-dataset	600-dataset	800-dataset	1000-dataset
1	0.5073	0.5631	0.2925	0.4457	0.3830
2	0.5155	0.5905	0.3229	0.4552	0.3960
3	0.5167	0.6306	0.3614	0.4695	0.4165
4	0.5167	0.6543	0.3851	0.4968	0.4220
5	0.5167	0.6666	0.3924	0.5164	0.4330
10	0.5167	0.6812	0.4278	0.5506	0.4789
20	0.5188	0.6880	0.4444	0.5712	0.5086
30	0.5188	0.6896	0.4494	0.5791	0.5234
40	0.5188	0.6910	0.4534	0.5811	0.5350
50	0.5188	0.6919	0.4581	0.5840	0.5451
60	0.5205	0.6919	0.4612	0.5856	0.5495
70	0.5205	0.6921	0.4620	0.5858	0.5529
80	0.5205	0.6925	0.4625	0.5858	0.5558
90	0.5205	0.6925	0.4630	0.5859	0.5581
100	0.5205	0.6933	0.4632	0.5860	0.5589
110	0.5205	0.6933	0.4637	0.5865	0.5599
120	0.5205	0.6933	0.4638	0.5871	0.5618
130	0.5205	0.6933	0.4641	0.5875	0.5621
140	0.5205	0.6933	0.4643	0.5875	0.5628
150	0.5205	0.6933	0.4643	0.5876	0.5634
160	0.5205	0.6933	0.4643	0.5876	0.5634
170	0.5205	0.6933	0.4646	0.5876	0.5634
180	0.5205	0.6933	0.4646	0.5876	0.5637
200	0.5205	0.6934	0.4646	0.5876	0.5643

Table 6.3 HV over Time for IBEA with all the research datasets

From the mentioned above table the following Figures were generated to describe the relation between time and HV: 6.7, 6.8, 6.9, 6,10 and 6.11

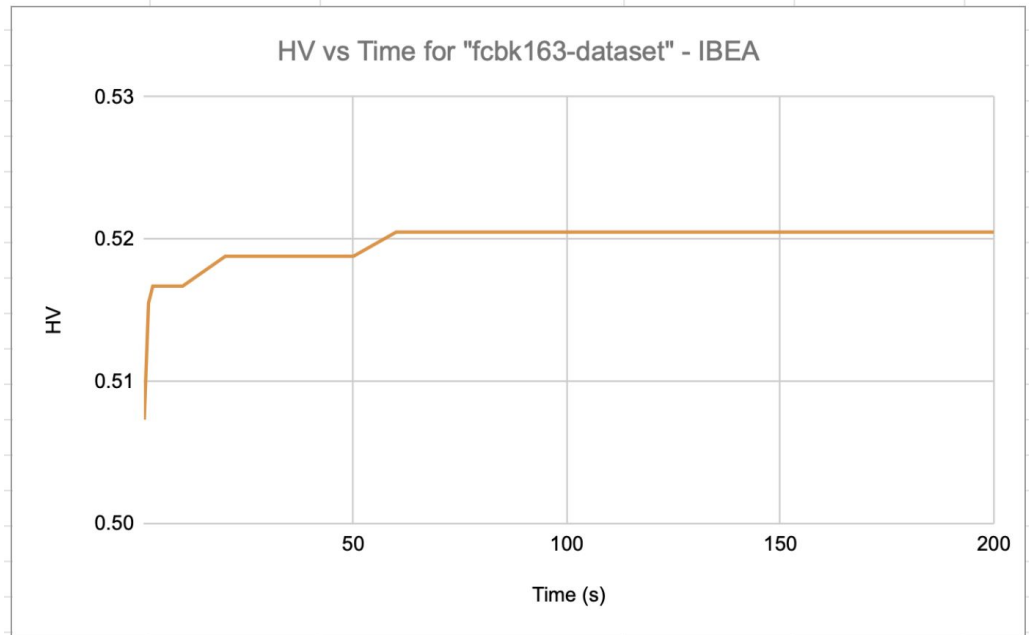


Figure 6.7 HV over time for IBEA with fcbk163-dataset

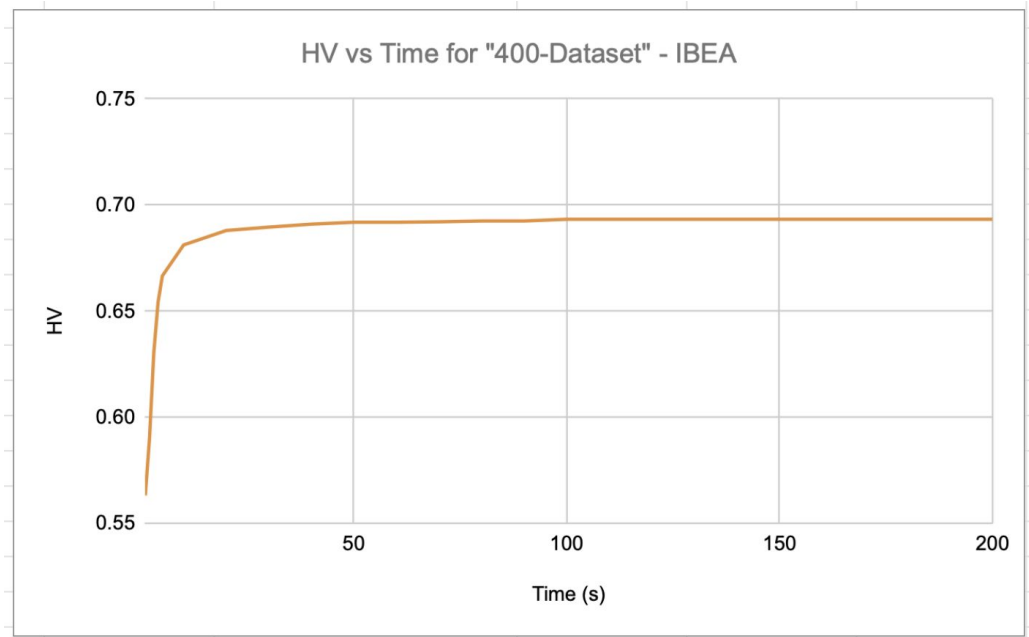


Figure 6.8 HV over time for IBEA with 400-dataset

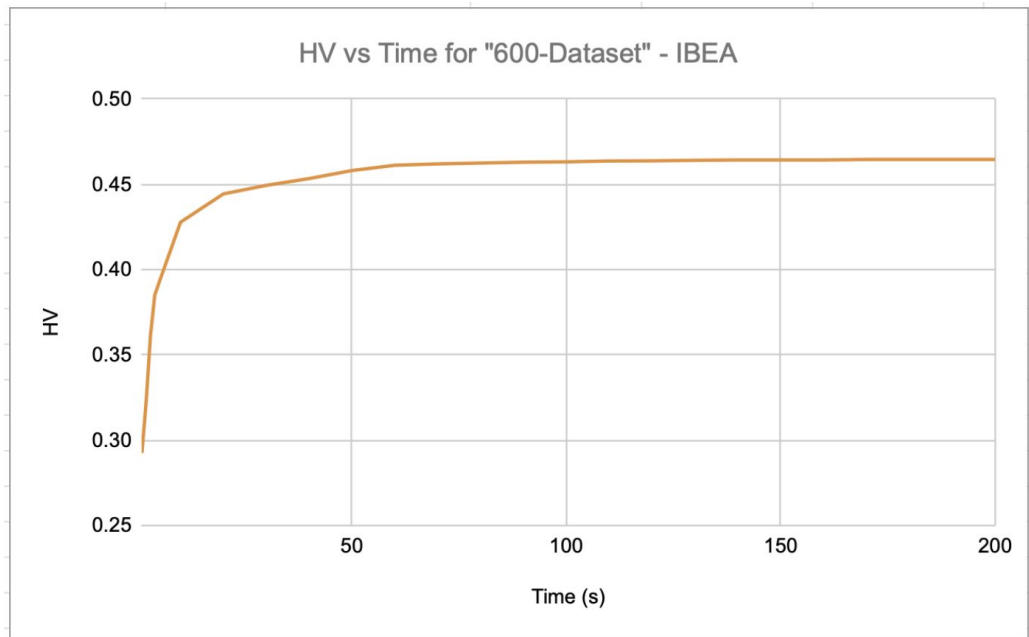


Figure 6.9 HV over time for IBEA with 600-dataset

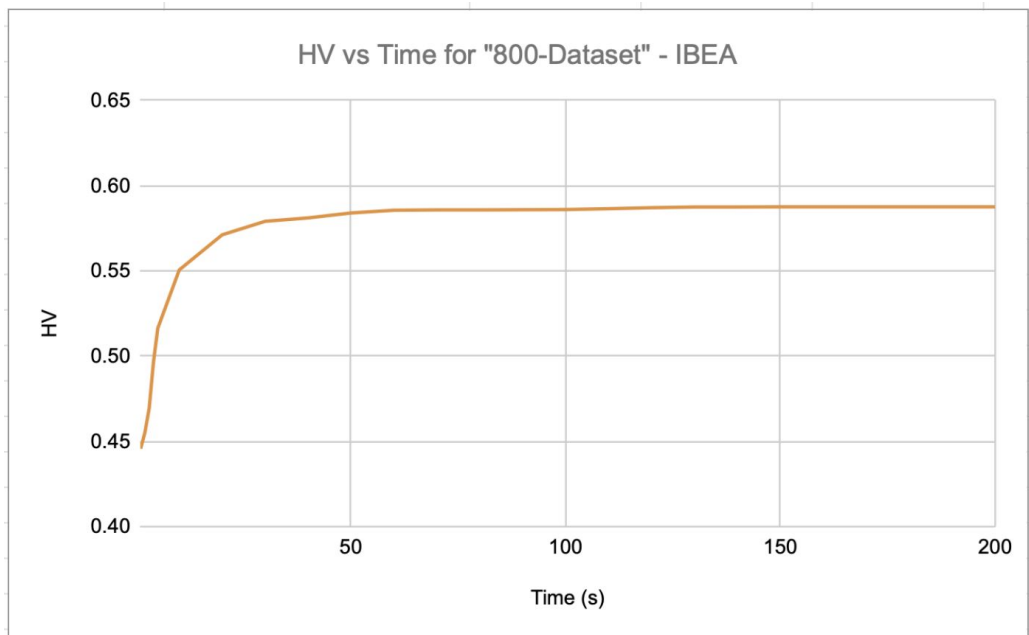


Figure 6.10 HV over time for IBEA with 800-dataset

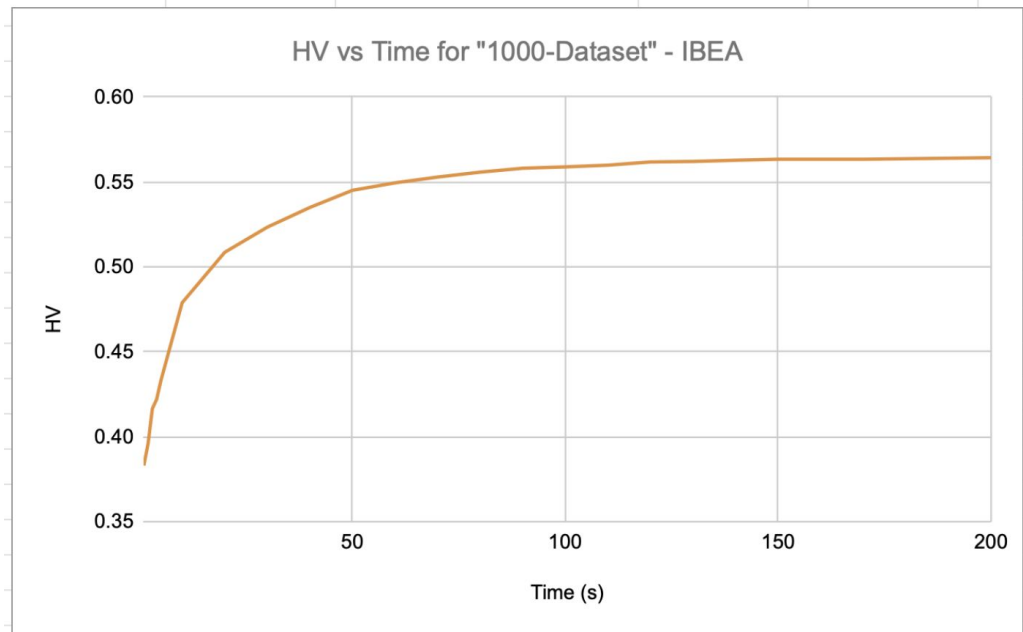


Figure 6.11 HV over time for IBEA with 1000-dataset

3. MOCeLL Results

The minimum execution time with a stable HV values for MOCeLL is similar to previous NSGA-II and IBEA algorithms. It's 1 second for the fcbk163-dataset and it's 30 for others. So far now, time is not better by comparing one algorithm to another. The results for MOCeLL were captured in Table 6.4 and the relation between time and HV is represented in the following figures: 6.12, 6.13, 6.14, 6.15 and 6.16

Time (s)	fcbk163-dataset	400-dataset	600-dataset	800-dataset	1000-dataset
1	0.4874	0.5640	0.3031	0.4667	0.3870
2	0.4874	0.5808	0.3222	0.478	0.4028
3	0.4874	0.6142	0.3425	0.488	0.4075
4	0.4886	0.6305	0.3561	0.4981	0.4213
5	0.4890	0.6382	0.3688	0.5127	0.4316
10	0.4894	0.6649	0.4003	0.5358	0.4624
20	0.4894	0.6769	0.4216	0.5654	0.4994
30	0.4906	0.6836	0.4335	0.5746	0.5226
40	0.4906	0.6845	0.4363	0.5769	0.5348
50	0.4906	0.6852	0.4368	0.5769	0.5398

60	0.4906	0.6852	0.4370	0.5782	0.5452
70	0.4906	0.6852	0.4379	0.579	0.5494
80	0.4906	0.6852	0.4383	0.5815	0.5505
90	0.4906	0.6861	0.4414	0.5815	0.5512
100	0.4906	0.6865	0.4423	0.5817	0.5522
110	0.4906	0.6870	0.4423	0.5817	0.5534
120	0.4906	0.6870	0.4423	0.5820	0.5539
130	0.4906	0.6870	0.4423	0.5820	0.5545
140	0.4906	0.6870	0.4428	0.5820	0.5554
150	0.4906	0.6879	0.4435	0.5830	0.5555
160	0.4906	0.6879	0.4435	0.5830	0.5555
170	0.4906	0.6879	0.4435	0.5830	0.5555
180	0.4906	0.6879	0.4435	0.5830	0.5555
200	0.4906	0.6934	0.4435	0.5830	0.5555

Table 6.4 HV over Time for MOCcell with all the research datasets

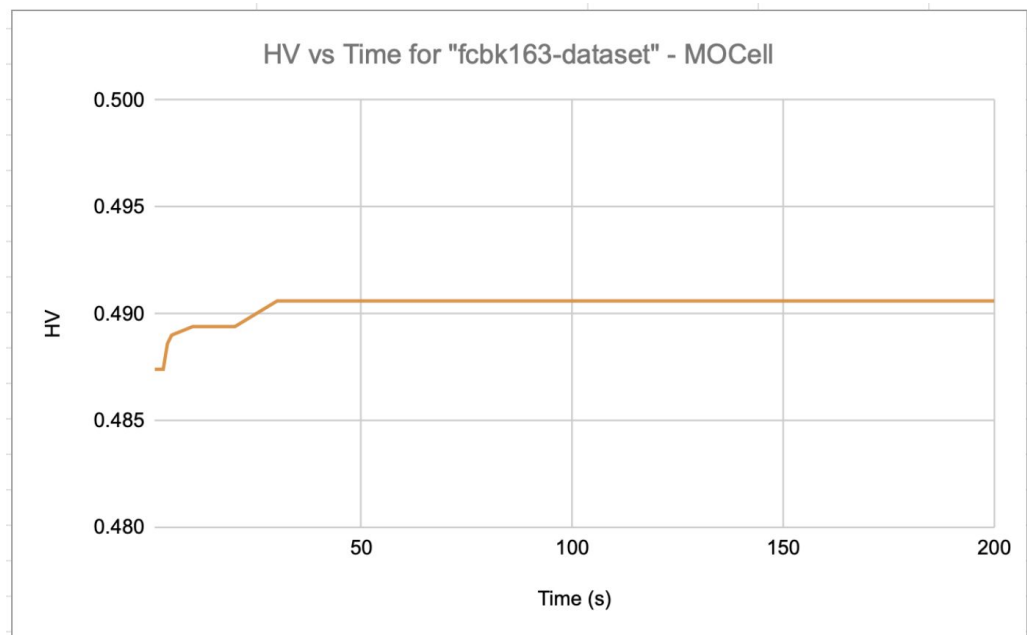


Figure 6.12 HV over time for MOCcell with fcbk163-dataset

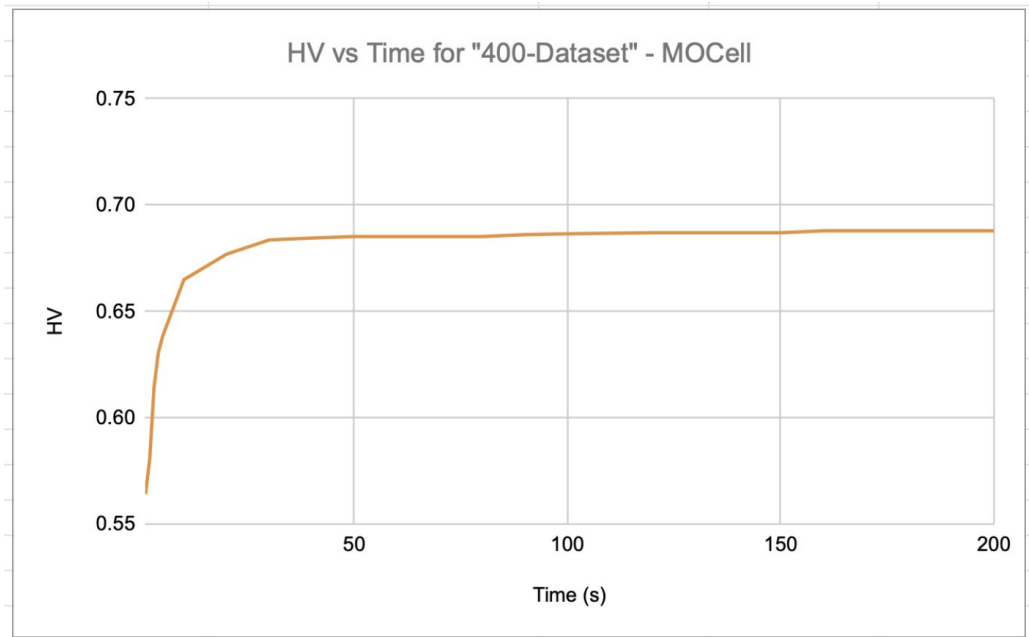


Figure 6.13 HV over time for MOCeCell with 400-dataset

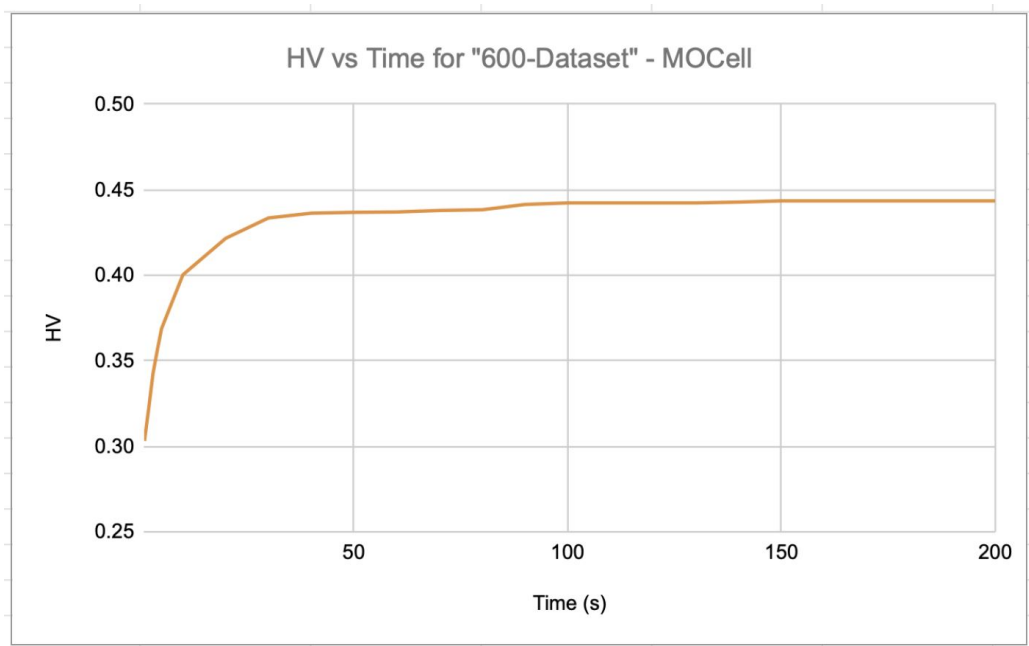


Figure 6.14 HV over time for MOCeCell with 600-dataset

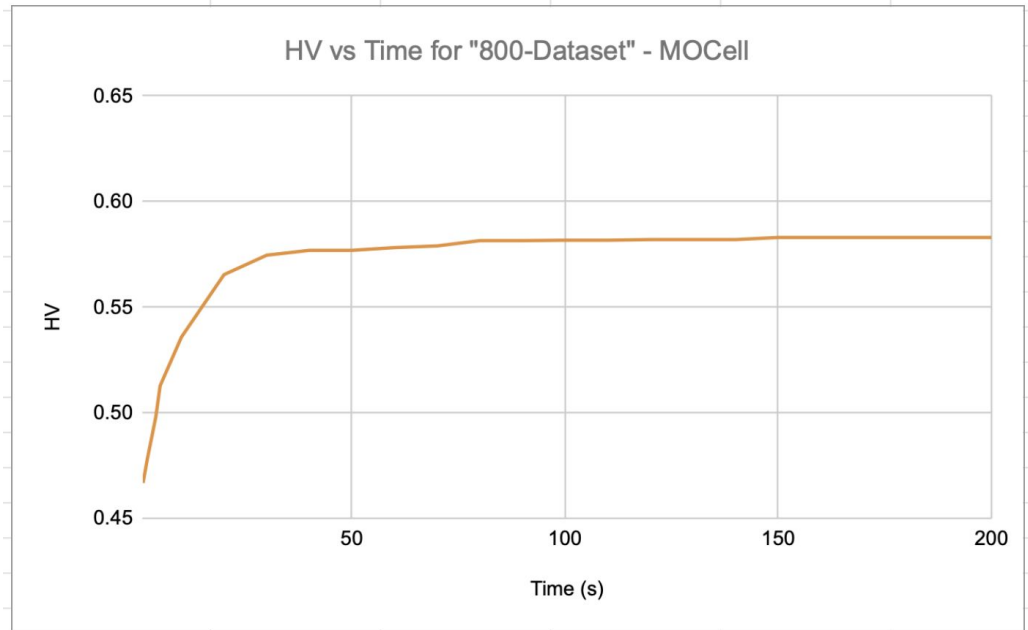


Figure 6.15 HV over time for MOCeCell with 600-dataset

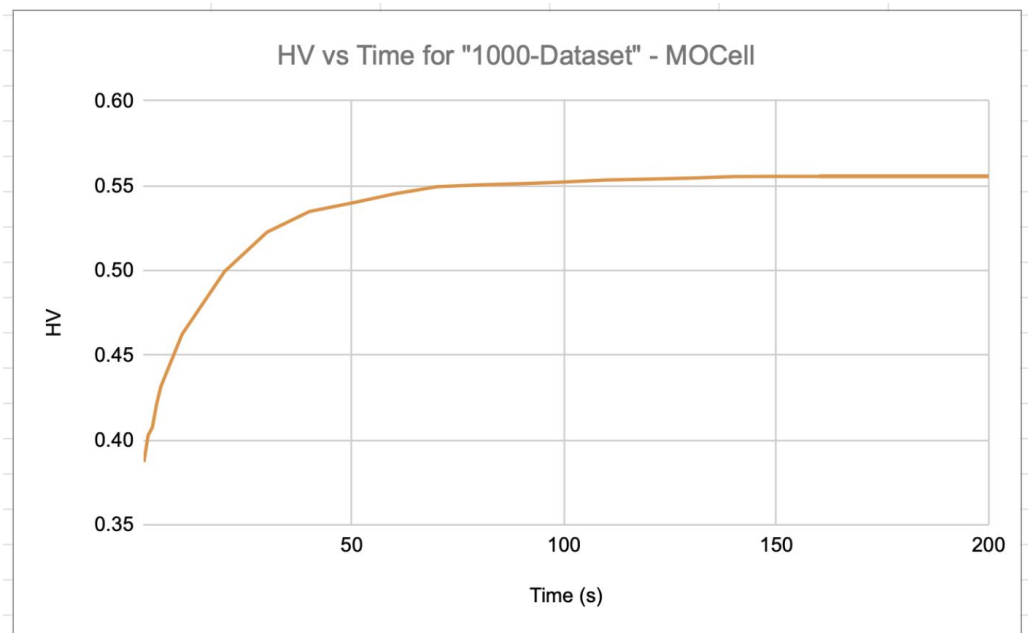


Figure 6.16 HV over time for MOCeCell with 1000-dataset

4. SPEA2 Results

Same result for minimum execution time of SPEA2, by 30 second, result is stable for all algorithms. It is represented in Table 6.5 and figures 6.17, 6.18, 6.19, 6.20 and 6.21.

Time (s)	Fcbk163-dataset	400-dataset	600-dataset	800-dataset	1000-dataset
1	0.5054	0.5643	0.2951	0.4417	0.3800
2	0.5127	0.6233	0.3287	0.4672	0.4058
3	0.5135	0.6518	0.3620	0.4979	0.4229
4	0.5136	0.6701	0.3860	0.5204	0.4421
5	0.5144	0.6721	0.3964	0.5300	0.4516
10	0.5157	0.6816	0.4191	0.5529	0.4915
20	0.5174	0.6886	0.4493	0.5705	0.5155
30	0.5182	0.6900	0.4579	0.5773	0.5292
40	0.5182	0.6907	0.4594	0.5797	0.5379
50	0.5182	0.6915	0.4606	0.5802	0.5487
60	0.5190	0.6915	0.4613	0.5821	0.5519
70	0.5195	0.6918	0.4621	0.5822	0.5570
80	0.5195	0.6924	0.4630	0.5822	0.5583
90	0.5195	0.6936	0.4633	0.5827	0.5594
100	0.5195	0.6940	0.4633	0.5833	0.5596
110	0.5195	0.6946	0.4642	0.5834	0.5607
120	0.5195	0.6962	0.4646	0.5834	0.5621
130	0.5195	0.6969	0.4651	0.584	0.5629
140	0.5195	0.6969	0.4653	0.5844	0.5636
150	0.5195	0.6972	0.4653	0.5844	0.5642
160	0.5195	0.6972	0.4653	0.5844	0.5648
170	0.5195	0.6972	0.4657	0.5845	0.5659
180	0.5195	0.6972	0.4668	0.5845	0.5662
200	0.5195	0.6934	0.467	0.5845	0.5670

Table 6.5 HV over Time for SPEA2 with all the research datasets

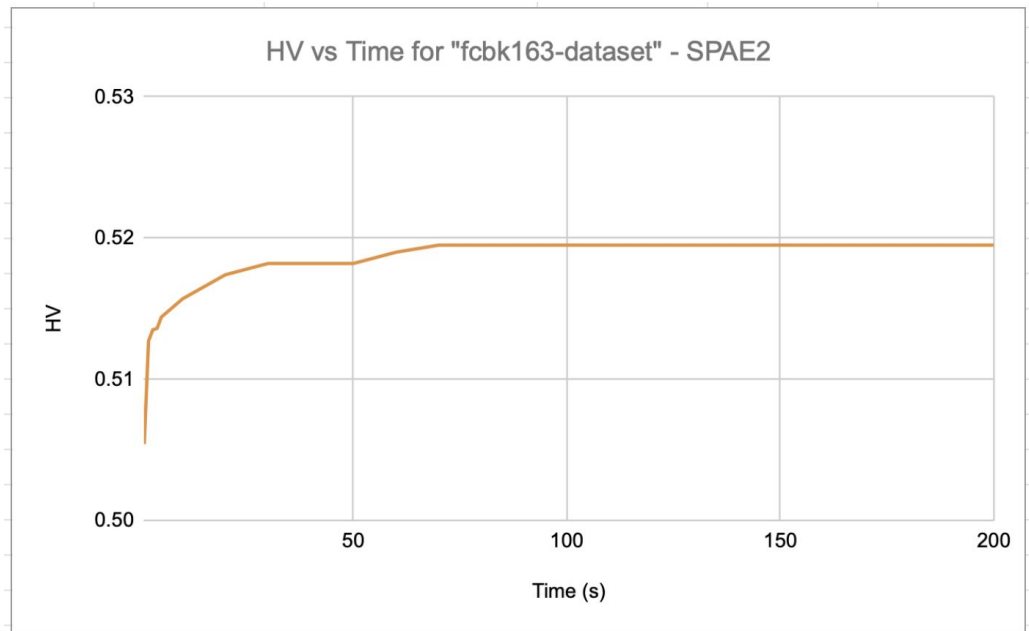


Figure 6.17 HV over time for SPEA2 with fcbk163-dataset

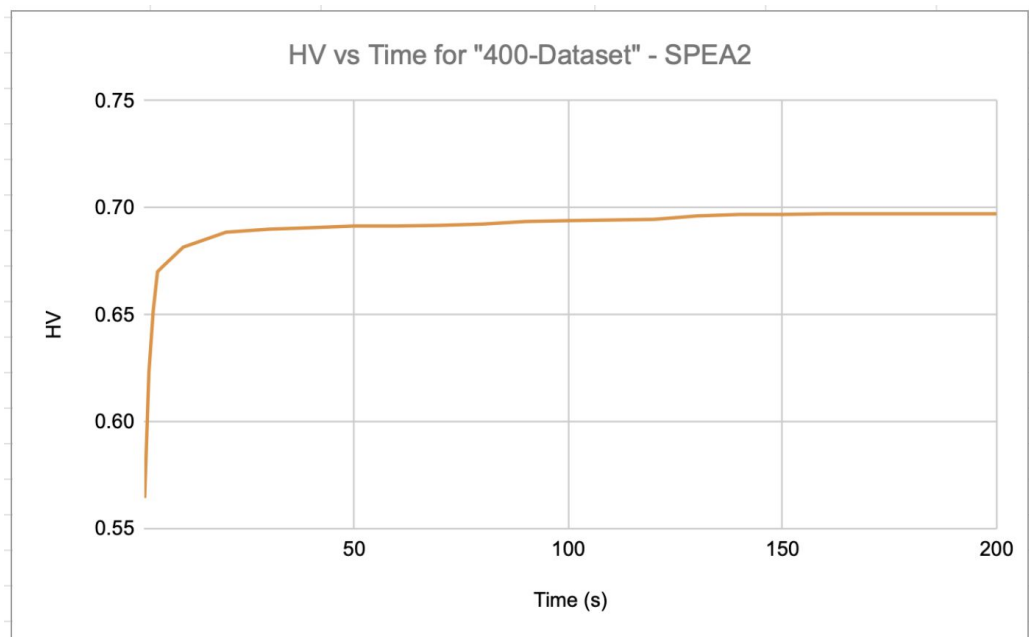


Figure 6.18 HV over time for SPEA2 with 400-dataset

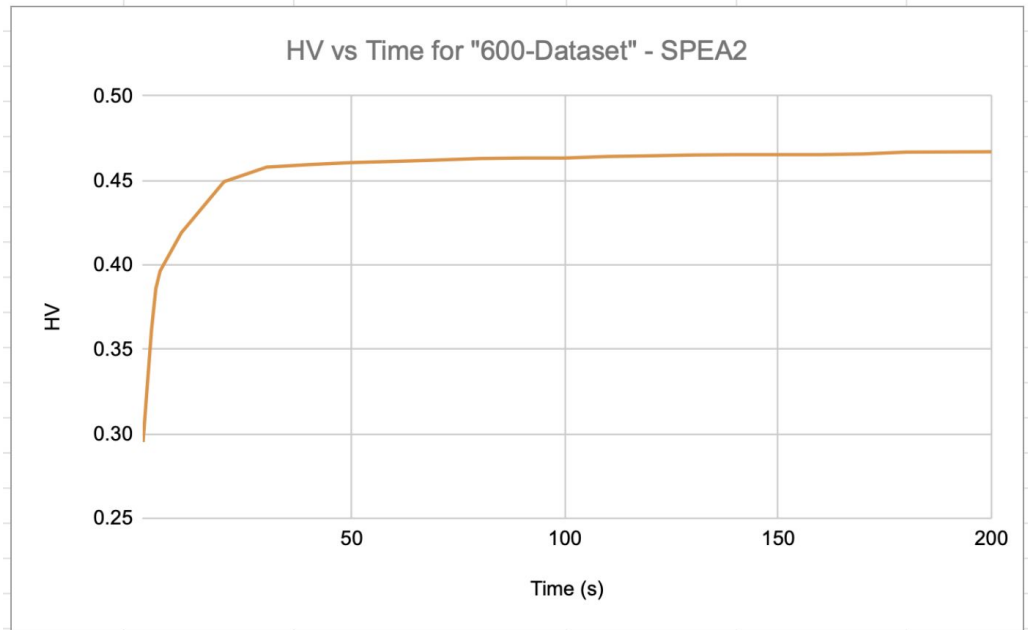


Figure 6.19 HV over time for SPEA2 with 600-dataset

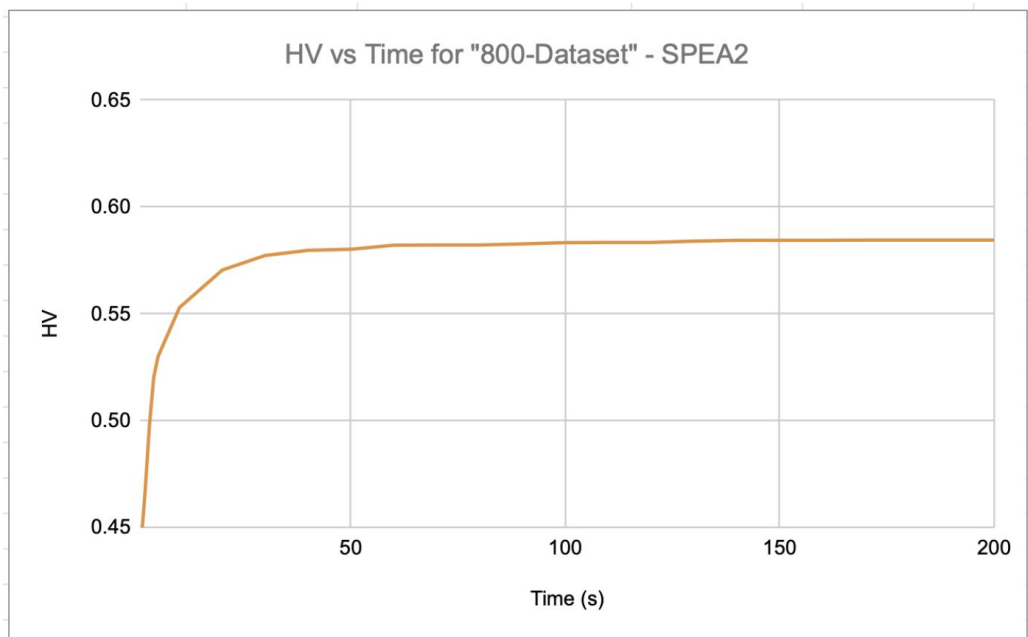


Figure 6.20 HV over time for SPEA2 with 800-dataset

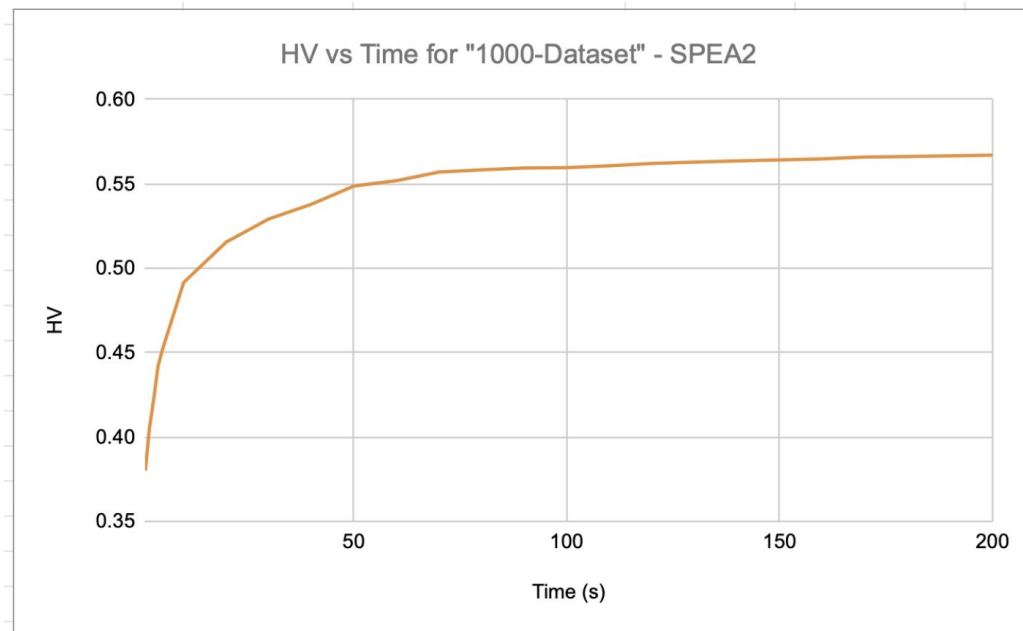


Figure 6.21 HV over time for SPEA2 with 1000-dataset

To summarize the result from all above tables and figures, the minimum required execution time for all algorithms is the same. It's 30 second with all datasets and algorithms.

6.2 Available Time Impact on HV value

The target of this experiment is to study the impact of the available time on the generated solutions quality. The available time could be defined as the available time for manual test cases execution. This value is connected to the problem definition itself, so once this value is changed for the same dataset, the problem is considered as a new one. To illustrate this by example, let say an available time for manual test cases execution is 160 hour. This value could be a total release available time, or it could be the remaining time at some point of the sprint. Hence, the prioritization has to respect that time. In this case the algorithms have to calculate the fitness functions depending on the available time value. The impact on fitness functions is considered as constraint, as example, if the test cases required execution time is more than the available time, then the coverage is different. As a result, the number of test cases that have to be included in the generated solution is different

depending on the available time. Moreover, this affects the high priorities test cases that might need more time. Therefore as mentioned previously, there is an obvious impact of available execution time on fitness functions.

This experiment was designed to measure the solution quality with two cases:

1. When the available time is a small value by comparing it to the required time for including all test cases.
2. When the available time is a large value that is close to the required time for including all test cases

The previous experiment in section 6.1 was done using an available time in the middle of the total expected execution time. Therefore HV value from the previous section is included for comparison purposes, then three cases are considered. In this experiment, the same settings that were mentioned in the previous experiment were used. As this second experiment measured the HV value depending on the available time for the same dataset, two datasets only were selected. Selection of datasets considered to study the results on a small and large datasets as in the following :

- Small dataset using fcbk163-dataset: for this dataset, the total expected execution time for all test cases is: 4365 (it doesn't matter if we consider it as hour or minutes for the experiment testing). So, for the first case, 500 is considered a small amount by comparing it with 4365, so it was used on problem definition. On the other hand, 4000 is considered as a close value to 4365, so it was used in the second case with the problem definition.
- Large Dataset using 1000-dataset: the total expected execution time is 25312. Therefore for the first case of selecting a small value for available time, 5000 was selected. While for the second case of selecting a large value of available time, 20000 was selected for the problem definition.

Two builds were executed per each mentioned dataset, available time was assigned to the problem definition each time as in the following screenshot:

```

19
20 public TCP() throws IOException {
21
22     // this("Dataset-163.csv",500);
23     // this("Dataset-163.csv",4000);
24
25     // this("Dataset-1000.csv",5000);
26     this( datasetFile: "Dataset-1000.csv", availableTime: 20000);
27
28

```

Figure 6.22 Assign the available time in problem definition

The screenshot was taken while doing the experiment of measuring HV for dataset-1000 with large available time, which was 20,000. Other problem definitions with comments are the other cases for this experiment. The results from all executions have been registered in Table 6.6. In addition to the results from the current experiment, Table 6.6 shows the HV values from the previous section when the selected value of the available time is between small and large selected values.

#	Time	NSGA II	IBEA	MOCcell	SPEA2
facebook163-dataset	500	0.8253	0.8241	0.8280	0.8449
facebook163-dataset	1,500	0.5130	0.5213	0.5031	0.5218
facebook163-dataset	4,000	0.0381	0.0384	0.0384	0.0384
1000-dataset	5,000	0.6737	0.6697	0.6895	0.7894
1000-dataset	13,000	0.4094	0.4140	0.4049	0.3832
1000-dataset	20,000	0.0468	0.0459	0.0515	0.1025

Table 6.6 HV mean for different values of available time

From the above table, it's clear that adding a smaller value of available time increased the HV values. While increasing the available time decreased the HV values. We can also notice that the result is applied on all algorithms and with both chosen datasets. The result is connected to the fact of using available time as constraint. Small value of available time limited the number of test cases that could be included in fitness functions. On the other hand, increasing it gives a better chance to add more test cases, this means higher complexity for the TCP problem.

The HV values for all builds of this experiment is represented on the following boxplots figures, 6.23, 6.24, 6.25 and 6.26. Form all figures and by considering the different scales on each boxplot, we can see that values are close together for all algorithms.

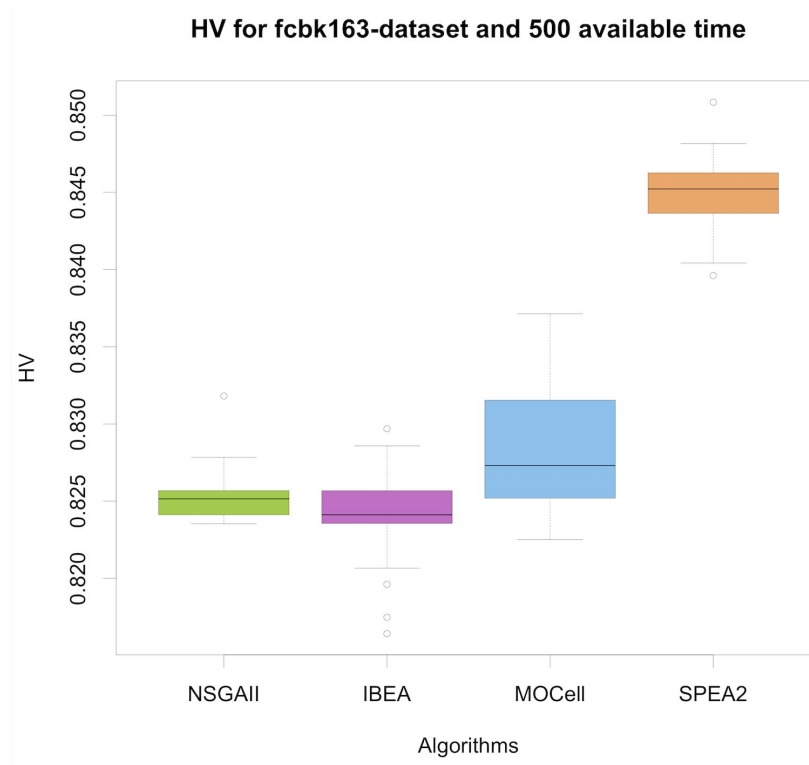


Figure 6.23 HV boxplot for small value of available time with small dataset size

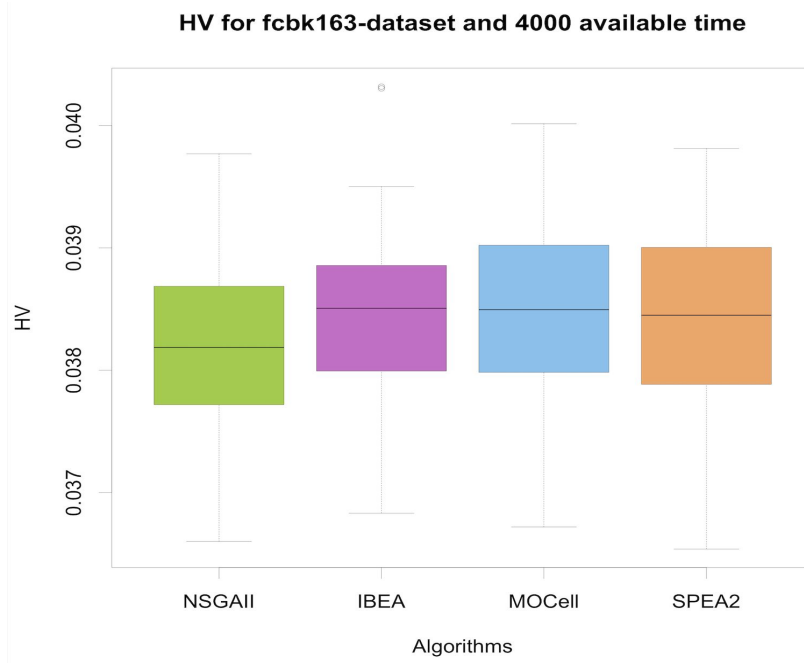


Figure 6.24 HV boxplot for large value of available time with small dataset size

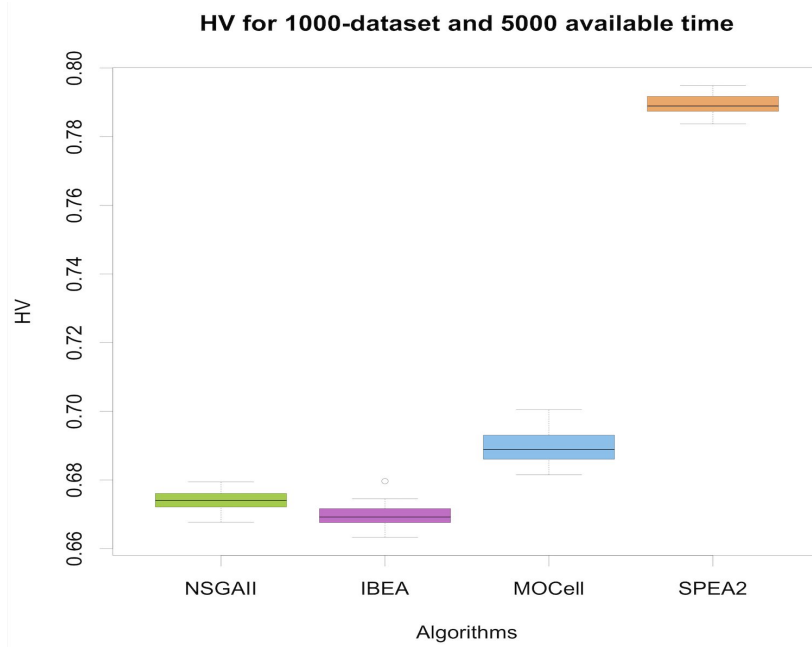


Figure 6.25 HV boxplot for small value of available time with large dataset size

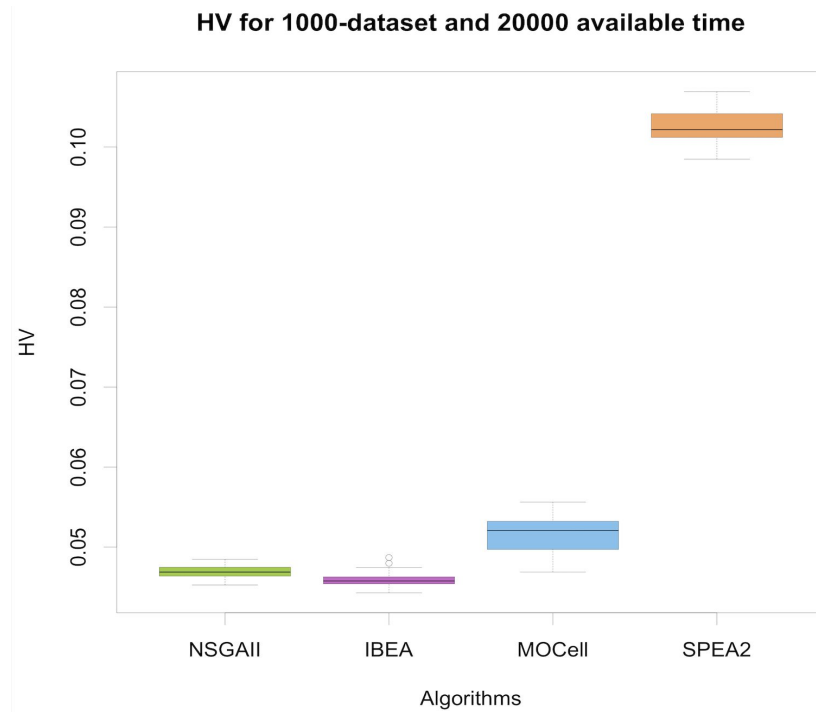


Figure 6.26 HV boxplot for large value of available time with large dataset size

For all algorithms and for both datasets, from boxplots, we can see that for 30 runs the HV values are too close together. In addition, by comparing each boxplot for the same dataset, we can see how increasing the available time decreased the HV values for all algorithms. This means, once the available time is close to the required time to all test cases, the HV is smaller. For example, if the required time for a set of test cases is 300 hour, choosing a small available time in the problem definition generates a high value of HV. While choosing a large value of available time like 280 hour, decreases the HV value.

To summarize the result from this experiment, increasing the remaining time decreases the value of HV itself. While decreasing the remaining time leads to large HV values.

6.3 Impact of remaining time on Algorithm Execution time

By returning back to experiment 2 in section 6.2, there was an impact of having a small or a large value of remaining time in the problem definition. Having a small value has to increase the HV value regardless of the dataset size. In this section a small value of remaining time was taken for one dataset and for one algorithm. The target is to see if the remaining time has any impact on the algorithm minimum execution time. As all algorithms have the same minimum execution time from previous experiment results, then only one algorithm is enough to do the study in the current experiment. The results for execution time in the previous section show that no impact of the dataset size on the minimum required execution time. Hence only one dataset has been chosen in the current experiment, it's the largest one 1000-dataset. Then for this experiment study the NSGA-II algorithm has been selected and 500 as a small value of available time was assigned to problem definition. The following table 6.7 captured the resulting values of HV in the first column with a timestamp of that value on the last column.

The time in the third column is printed in milliseconds, therefore we can see how the HV has a high value from the beginning of execution. For example, with the first record at 727 milliseconds, the HV was very high and it's close to 1. Therefore adding a high or a small stopping time as a stopping condition doesn't have an impact on HV. Since it started with a very high value, giving an opportunity to the algorithm to produce a better solution is not possible here. It reached the highest values from the beginning of execution and it was close to 1.

As a result, the algorithm has a stable value of HV, then the minimum required execution time of the algorithm is just milliseconds. On the other hand, the available time on the previous experiment was something between small and large values. Then the result for minimum execution time was 30 seconds for all algorithms and datasets. This means, when the algorithms have

more time as a constraint gives it better opportunity to find more solutions and HV was not a high one from the beginning. So, increasing the stopping time as was set to 30 second in the previous experiment affected the result. It gives the algorithms an opportunity to continue finding new solutions and to enhance the HV until having a stable value.

0.9705918502735312	Time:	727
0.9710775331864914	Time:	1292
0.9718478911626873	Time:	1516
0.9733145510754971	Time:	1786
0.9733145510754971	Time:	2096
0.9735867652729756	Time:	2414
0.9736927757697763	Time:	2633
0.9743284008392211	Time:	2800
0.9752612173935777	Time:	2965
0.9752612173935777	Time:	3107
0.9752614428852477	Time:	3279
0.9752614428852477	Time:	3411
0.9752633448584632	Time:	3568
0.9752633448584632	Time:	3697

Table 6.7 Captured HV when the remaining time is a small value

To summarize this section result, small amounts of remaining time leads to a stable value of HV values in early milliseconds of execution. Then a small value of execution time is enough to be selected as a stopping condition in this case. On the other hand, increasing the remaining time, requires more execution time to reach a stable value of HV.

6.4 Hypervolume (HV) Comparison

As all algorithms require the same minimum execution time regardless of the dataset size then no one is better than another from time perspective. A new experiment has been presented here to study the algorithms behaviour from HV quality perspective. Hence the minimum execution time from section 6.1 was used as a stopping condition. In addition to 30 ms as a stopping condition, 30 runs have been set to each execution with all datasets. This

allowed us to see if any algorithm outperformed the others from the solution quality view point. For this experiment design, all algorithms common settings were initialized with the same values. So the same default settings that were described in Table 5.1 were used. The remaining time for all datasets is fixed here to 2000. As a result, it was considered as a small value for some datasets. While it is a large one for others. This has been reflected on the HV values for different datasets, but it doesn't have to affect the experiment objective. Tables 6.8 and 6.9 represent the mean and median values for all builds with all algorithms and datasets.

#	NSGA II	IBEA	MOCcell	SPEA2
fcbk163-dataset	0.3761	0.3834	0.3595	0.3834
400-dataset	0.7960	0.7982	0.7957	0.8007
600-dataset	0.8488	0.8549	0.8528	0.8561
800-dataset	0.8799	0.8708	0.8739	0.8799
1000-dataset	0.8932	0.8951	0.8968	0.8898

Table 6.8 HV mean for 30 independent runs

#	NSGA II	IBEA	MOCcell	SPEA2
fcbk163-dataset	0.3754	0.3837	0.3591	0.3834
400-dataset	0.7968	0.7991	0.7939	0.7996
600-dataset	0.8496	0.8557	0.8510	0.8557
800-dataset	0.8812	0.8712	0.8737	0.8797
1000-dataset	0.8934	0.8942	0.8982	0.8893

Table 6.9 HV median for 30 independent runs

From the above table, we can see a minor variation from one algorithm to another with the same dataset. As a result we can't consider any algorithm

is outperforming the others. For example, with a 1000-dataset, check the first two decimal places for mean values, NSGA-II, IBEA and MOCell are 0.89. The difference between SPEA2 is only 0.01.

The HV values are small for the fcbk163-dataset because the 2000 as the remaining time is a large one. While the HV values for the 1000-dataset are large as 2000 is a small remaining time value. This result was presented previously on section 6.2.

The generated HV values by jMetal were used as input data to generate several boxplots by RStudio for all builds. The fcbk163-dataset, 400-dataset, 600-dataset, 800-dataset and 1000-dataset are represented by figures: 6.27, 6.28, 6.29, 6.30 and 6.31.

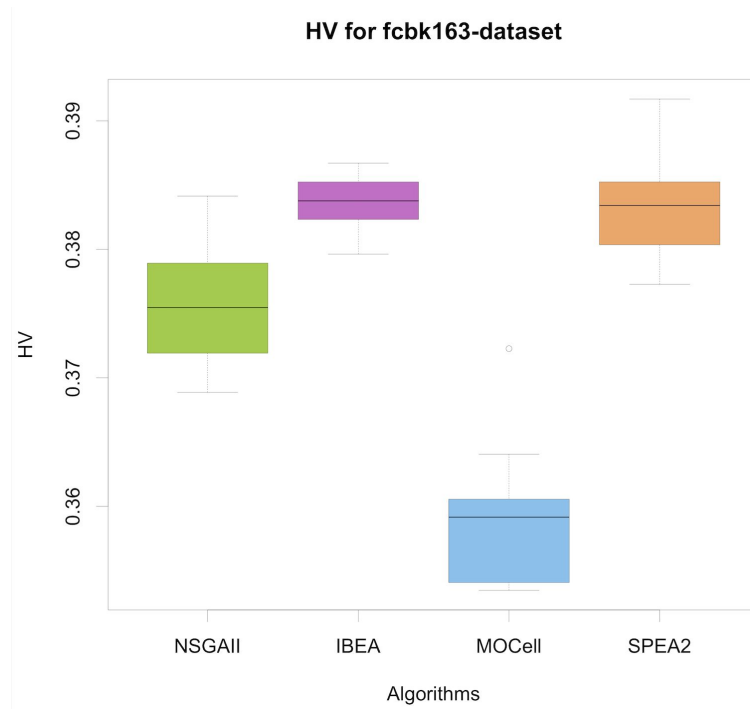


Figure 6.27 HV quality indicator for fcbk163-dataset

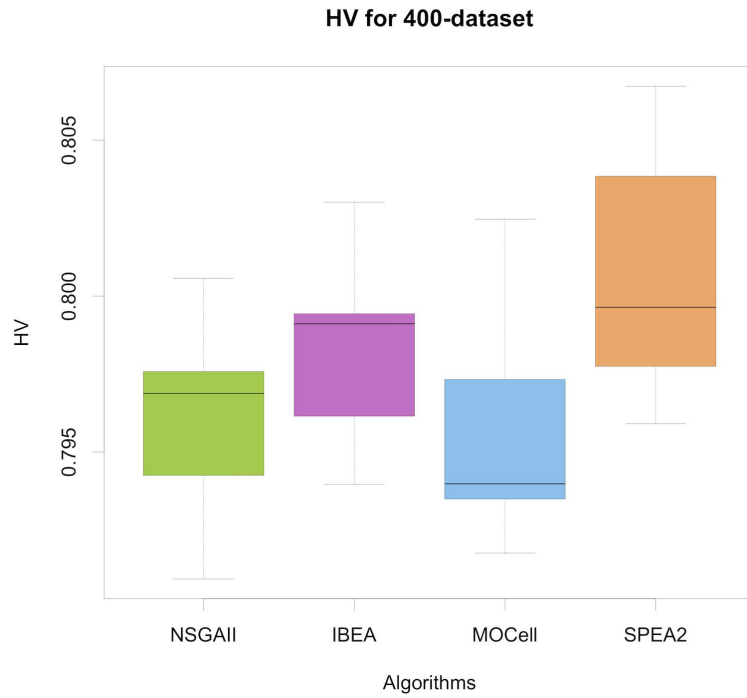


Figure 6.28 HV quality indicator for 400-dataset

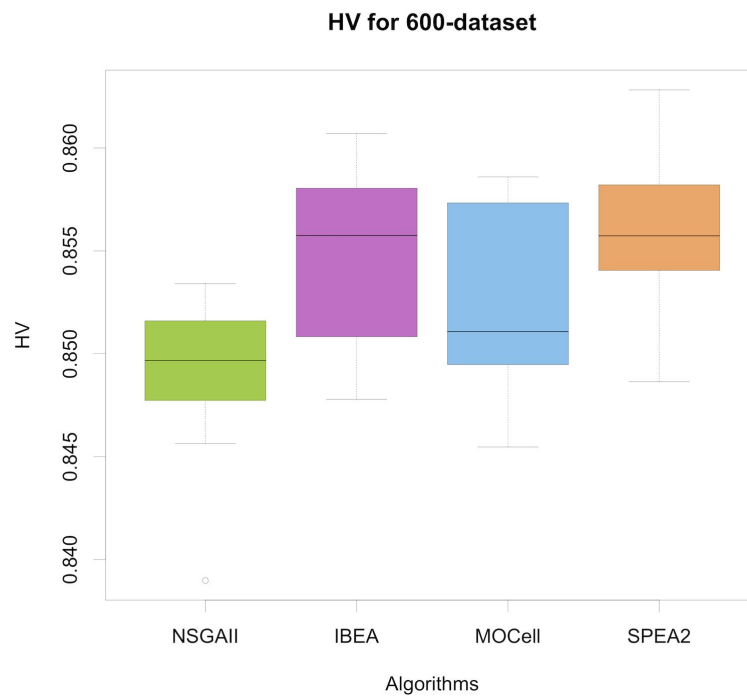


Figure 6.29 HV quality indicator for 600-dataset

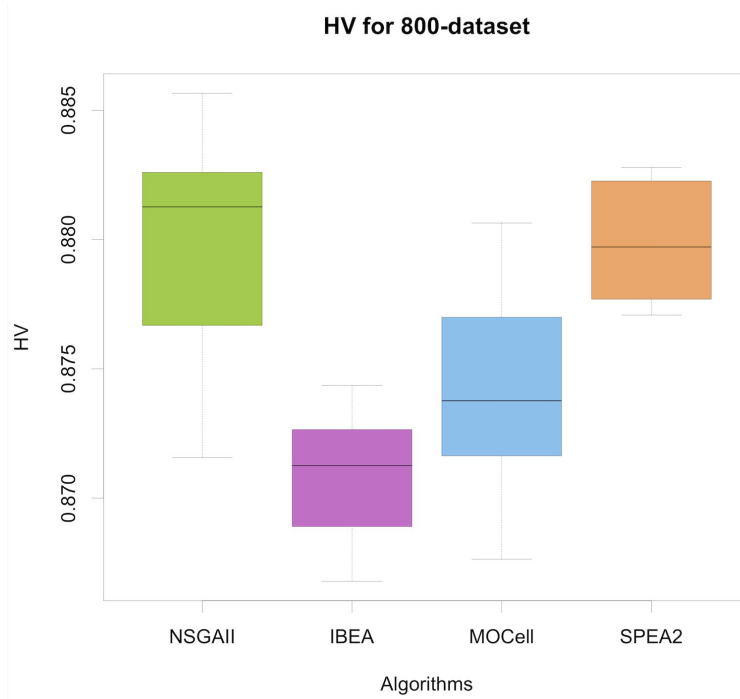


Figure 6.30 HV quality indicator for 800-dataset

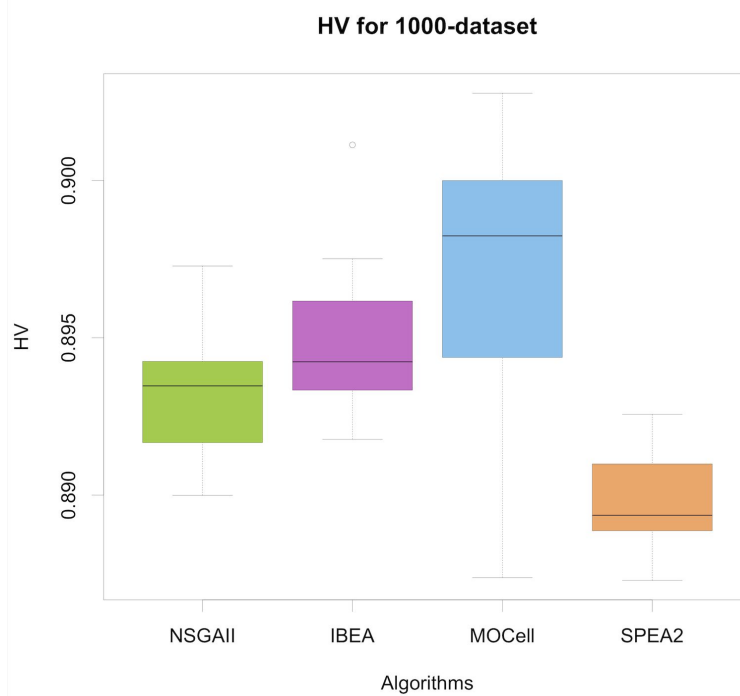


Figure 6.31 HV quality indicator for 1000-dataset

Same observations and results that were generated depending on table 6.8 are also clear by boxplots. Therefore, from boxplots, it's clear that values

are close together for all algorithms per dataset. The same algorithm has higher value with a specific dataset, while it has a lower with another one. All these differences don't make any sense since they are minor variations. From the generated results we also can compare the max and min values for each algorithm with the selected datasets, table 6.10 represents the results.

Dataset Name	Algorithms	Max HV	Min HV
fcbk163-dataset	NSGA-II	0.3841	0.3688
	IBEA	0.3867	0.3796
	MOCeII	0.3722	0.3534
	SPEA2	0.3916	0.3772
400-dataset	NSGA-II	0.8005	0.7909
	IBEA	0.8030	0.7939
	MOCeII	0.8024	0.7917
	SPEA2	0.8067	0.7959
600-dataset	NSGA-II	0.8534	0.8389
	IBEA	0.8607	0.8477
	MOCeII	0.8585	0.8454
	SPEA2	0.8628	0.8486
800-dataset	NSGA-II	0.8856	0.8715
	IBEA	0.8743	0.8667
	MOCeII	0.8806	0.8676
	SPEA2	0.8827	0.8770
1000-dataset	NSGA-II	0.8972	0.8899
	IBEA	0.9011	0.8917
	MOCeII	0.9027	0.8873
	SPEA2	0.8925	0.8872

Table 6.10 Max and min value for each algorithm with all datasets

Based on the above table, we can see the exact min and max values for each algorithm. For the first dataset which has the minimum number of test cases, the MOCell has the minimum value, while the SPEA2 has the largest. Same observation about the min and max values, the difference is small and values are close together for the same dataset with all algorithms.

To summarize the result of this section, we can say that all algorithms have similar values for HV with the same dataset. Hence no algorithm is better than another to produce a better solution.

Chapter 7. Threats to Validity and Conclusion

This chapter presents the validity threats for the conducted experiments in section 1. After that, the final conclusion about the research and results has been summarized in section 2. Finally, the possible future work has been presented in the last section.

7.1 Threats to Validity

Several types of validity threats are connected with search based software engineering problems and experiments. Therefore in this section several threats have been listed in order to see if there is an impact for those threats on the results and conclusion of this research. Some of these threats are connected to the research nature, others for experiment or datasets as in the following validity threats types:

- **Conclusion Validity Threats**

This threat type is connected with the relationship between treatment and results [13]. For this research experiment, all selected algorithms are genetic algorithms, therefore the four algorithms generate a random population as an initial point. This randomly generated population might lead to a bad start or a good one just by chance. At the same time we can't guarantee that the same randomly initial generated population has been created in the next run. Therefore, in order to avoid or at least to minimize the randomness effect on the results, several runs were used per mentioned experiment in chapter 6. Those runs have been chosen to be 30 independent runs, then the mean was calculated for the 30 runs for doing analysis.

- **Internal Validity Threats**

This type of threat is connected with the fact that some factors may have impact on the results while the research doesn't have awareness about them [13]. As an example in this research, the values of algorithms parameters are the defaults. Those defaults values were set by jMetal implementation, hence changing them may have an impact on the results. On the other hand, parameter tuning was not included in this experiment, therefore we can't guarantee if there are better or worst results in case parameter values were changed. Another threat may be considered here is the code design, the experiments were built on the current java implementation of jMetal. Therefore, if the same experiments have been done using the python version of jMetal or any other framework, different results might be generated.

- **Construct Validity Threats**

This is connected with the resulting outcome and the treatment, or in other words, by the relation between theory and observations [13]. In this research dataset was a challenge since there was no available dataset for the research test cases prioritization problem. At the same time, it was not possible to take actual datasets from available companies. The reason is this research was connected with manual testing, then those test cases are directly connected with the business. And business in most cases is connected with paid services that can't be shared for the public. The solution for the research test cases prioritization problem has built the dataset from available applications for the public. Therefore, the Facebook app was selected for building a dataset that is close to the real one. At the same time, specifying the frontend components was done by me as a researcher, but I don't have

access to the code. Hence, the frontend components might not be exactly as specified for building the dataset. Another connected issue with dataset is the need for having different datasets sizes and characteristics. Therefore, the solution was creating random datasets with random weights. Those weights also were created randomly and they represent the priority for unknown components and business. They are not actual datasets, but they have the same characteristics and values that will have impact on calculating the fitness functions.

- **External Validity Threats**

Generalization of results and approach is connected with external validity. In this research several datasets between 163 to 1000 test cases were generated. Then experiments included all generated datasets to see the impact of size change on results. At the same time, generalization of the approach is possible to several problems in software engineering, specifically those problems that are connected with weight or a need to create large datasets.

7.2 Conclusion

This research provided a framework for doing automated prioritization of manual test cases that are going to be tested for the first time. Those test cases are connected to component based frontend framework with web application, therefore building the problem and fitness functions considered this fact. Having this kind of prioritization being more important when the test cases are a long list and no previous history for them. Once there is any change in requirements or finding some blockers with limited and sensitive time, it's important to re-prioritize the test cases. Automated prioritization at any stage depending on the new needs and changes helps to include the high

priority test cases in early stages. Moreover, it helps to increase the coverage with the available time. The proposed framework helps to provide solutions that have the best values for the mentioned objectives. In addition, the proposed framework helps the decision makers to discover any violations on priorities and to exclude them from the generated priorities.

Four algorithms and five datasets were included in experiments of this research. They were all evaluated by HV as a quality indicator for each generated solution. It displayed a close quality for all algorithms with the same dataset. A small variance from one algorithm to another between captured mean values of HV was found. Hence, there is no algorithm that outperforms others from a quality perspective.

Another measurement was taken for evaluating the performance of each algorithm by execution time. Results have shown that all algorithms have the same minimum required execution time for having a stable HV value. The registered value for that execution time for all algorithms is 30 second. On the other hand, with a small dataset size, the stability needs only milliseconds. Another case that has milliseconds as minimum execution time is having a very limited or a small value of available time. This time is defined in the problem TCP definition and it has been considered as a constraint. In this case, the algorithms provide solutions with high HV value from the beginning of execution.

7.3 Future Work

Three main ideas are available now for future work of this research:

1. Building a UI for the generated framework, by this idea, a UI could be built to support the test cases documentation directly to the framework instead of adding them by csv files. The UI

will also provide a method to add the front end components and requirements priorities. This will help to automatically connect the documented manual test cases with frontend components. As a result, the weight will be automatically calculated instead of doing this manually. This means the user will be required only to add the test cases and to define components and their priorities. After that, for any prioritization or when components priorities change, all other required work for datasets will be prepared automatically.

The UI also will be used to present the generated solutions and the violations of test cases priorities. In addition, it will give the user several input choices to change the available time, select the algorithm or change the settings.

2. Building an API to integrate the generated framework with other test cases management systems. This will provide the chance to use the current systems with the generated frameworks at the same platform.
3. The same test cases prioritization could be studied and a new solution could be generated by Natural Language Processing. This could be done since the test cases are being documented as text for manual test cases. The new solution could be compared by the current research solution and then new results may be generated regarding quality or performance.

References

1. McLeod, Raymond. "Software Testing: Testing Across the Entire Software Development Life Cycle." (2007).
2. Ruparelia, Nayan B. "Software development lifecycle models." *ACM SIGSOFT Software Engineering Notes* 35.3 (2010): 8-13.
3. Rähkä, Outi. "Applying genetic algorithms in software architecture design." Master's thesis, 2008.
4. Barr, Earl T., Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey." *IEEE transactions on software engineering* 41, no. 5 (2014): 507-525.
5. Sharma, Chayanika, Sangeeta Sabharwal, and Ritu Sibal. "A survey on software testing techniques using genetic algorithm." *arXiv preprint arXiv:1411.1154* (2014).
6. Hooda, Itti, and Rajender Singh Chhillar. "Software test process, testing types and techniques." *International Journal of Computer Applications* 111.13 (2015).
7. Acharya, Shivani, and Vidhi Pandya. "Bridge between Black Box and White Box–Gray Box Testing Technique." *International Journal of Electronics and Computer Science Engineering* 2.1 (2012): 175-185.
8. Nidhra, Srinivas, and Jagruthi Dondeti. "Black box and white box testing techniques-a literature review." *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012): 29-50.
9. Tulasiraman, Megala, Nivethitha Vivekanandan, and Vivekanandan Kalimuthu. "Multi-objective Test Case Prioritization Using Improved Pareto-Optimal Clonal Selection Algorithm." *3D Research* 9.3 (2018): 32.
10. Watkins, John, and Simon Mills. *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2010.
11. Bass, Len, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.
12. Wu, Jin, and Shapour Azarm. "Metrics for quality assessment of a multiobjective design optimization solution set." *J. Mech. Des.* 123.1 (2001): 18-25.

13. de Oliveira Barros, Márcio, and Arilo Cláudio Dias-Neto. "0006/2011-threats to validity in search-based software engineering empirical studies." *RelaTe-DIA 5.1* (2011).
14. Gligoric, Milos, et al. "An empirical evaluation and comparison of manual and automated test selection." *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014.
15. Chernak, Yuri. "Validating and improving test-case effectiveness." *IEEE software* 18.1 (2001): 81-86.
16. Whalen, Michael W., et al. "Coverage metrics for requirements-based testing." *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006.
17. Utting, Mark, Alexander Pretschner, and Bruno Legard. "A taxonomy of model-based testing." (2006).
18. Shahid, Muhammad, Suhaimi Ibrahim, and Mohd Naz'ri Mahrin. "A study on test coverage in software testing." *Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia* (2011).
19. Askarunisa, MS A., MS L. Shanmugapriya, and DR N. Ramaraj. "Cost and coverage metrics for measuring the effectiveness of test case prioritization techniques." *INFOCOMP 9.1* (2010): 43-52.
20. Rothermel, Gregg, et al. "Prioritizing test cases for regression testing." *IEEE Transactions on software engineering* 27.10 (2001): 929-948.
21. Srikanth, Hema, Laurie Williams, and Jason Osborne. "System test case prioritization of new and regression test cases." *Empirical Software Engineering*, 2005. 2005 International Symposium on. IEEE, 2005.
22. Qu, Bo, et al. "Test case prioritization for black box testing." *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. Vol. 1. IEEE, 2007.
23. Singh, Yogesh, et al. "Systematic literature review on regression test prioritization techniques." *Informatica* 36.4 (2012).

24. Yoo, Shin, and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." *Software Testing, Verification and Reliability* 22.2 (2012): 67-120.
25. Elbaum, Sebastian, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. Vol. 25. No. 5. ACM, 2000.
26. Bryce, Renée C., and Atif M. Memon. "Test suite prioritization by interaction coverage." *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007.
27. Wong, W. Eric, et al. "A study of effective regression testing in practice." *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*. IEEE, 1997.
28. Rothermel, Gregg, et al. "Test case prioritization: An empirical study." *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99). 'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999.
29. Elbaum, Sebastian, Alexey G. Malishevsky, and Gregg Rothermel. "Test case prioritization: A family of empirical studies." *IEEE transactions on software engineering* 28.2 (2002): 159-182.
30. Kim, Jung-Min, and Adam Porter. "A history-based test prioritization technique for regression testing in resource constrained environments." *Proceedings of the 24th international conference on software engineering*. ACM, 2002.
31. Park, Hyuncheol, Hoyeon Ryu, and Jongmoon Baik. "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing." *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, 2008.
32. Fazlalizadeh, Yalda, et al. "Incorporating historical test case performance data and resource constraints into test case prioritization." *International Conference on Tests and Proofs*. Springer, Berlin, Heidelberg, 2009.
33. Marijan, Dusica, Arnaud Gotlieb, and Sagar Sen. "Test case prioritization for continuous regression testing: An industrial case study." *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013.

34. Huang, Yu-Chi, Kuan-Li Peng, and Chin-Yu Huang. "A history-based cost-cognizant test case prioritization technique in regression testing." *Journal of Systems and Software* 85.3 (2012): 626-637.
35. Mogyorodi, Gary. "Requirements-based testing: an overview." *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39.* IEEE, 2001.
36. Srikanth, Hema, Laurie Williams, and Jason Osborne. "System test case prioritization of new and regression test cases." *2005 International Symposium on Empirical Software Engineering, 2005..* IEEE, 2005.
37. Hou, Shan-Shan, et al. "Quota-constrained test-case prioritization for regression testing of service-centric systems." *2008 IEEE International Conference on Software Maintenance.* IEEE, 2008.
38. Ramasamy, Krishnamoorthi, and Sahaaya Arul Mary. "Incorporating varying requirement priorities and costs in test case prioritization for new and regression testing." *2008 International Conference on Computing, Communication and Networking.* IEEE, 2008.
39. Krishnamoorthi, R., and SA Sahaaya Arul Mary. "Factor oriented requirement coverage based system test case prioritization of new and regression test cases." *Information and Software Technology* 51.4 (2009): 799-808.
40. Walcott, Kristen R., et al. "Timeaware test suite prioritization." *Proceedings of the 2006 international symposium on Software testing and analysis.* ACM, 2006.
41. Conrad, Alexander P., Robert S. Roos, and Gregory M. Kapfhammer. "Empirically studying the role of selection operators during search-based test suite prioritization." *Proceedings of the 12th annual conference on Genetic and evolutionary computation.* ACM, 2010.
42. Smith, Adam M., and Gregory M. Kapfhammer. "An empirical study of incorporating cost into test suite reduction and prioritization." *Proceedings of the 2009 ACM symposium on Applied Computing.* ACM, 2009.
43. Rummel, Matthew J., Gregory M. Kapfhammer, and Andrew Thall. "Towards the prioritization of regression test suites with data flow information." *Proceedings of the 2005 ACM symposium on Applied computing.* ACM, 2005.

44. Henard, Christopher, et al. "Comparing white-box and black-box test prioritization." 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016.
45. Sampath, Sreedevi, et al. "Prioritizing user-session-based test cases for web applications testing." 2008 1st International Conference on Software Testing, Verification, and Validation. IEEE, 2008.
46. Rosen, L. S. R., and Leon Shklar. "Web Application Architecture: Principles, Protocol and Practices." (2009).
47. Medvidovic, Nenad, and Richard N. Taylor. "Software architecture: foundations, theory, and practice." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 2010. Comparative
48. Gizas, Andreas, Sotiris Christodoulou, and Theodore Papatheodorou. "Comparative evaluation of javascript frameworks." Proceedings of the 21st International Conference on World Wide Web. ACM, 2012.
49. Mariano, Carl Lawrence. "Benchmarking JavaScript Frameworks." (2017).
50. Flanagan, David, and Gregor M. Novak. "Java-Script: The Definitive Guide." (1998): 41-44.
51. Elliott, Eric. Programming JavaScript applications: Robust web architecture with node, HTML5, and modern JS libraries. " O'Reilly Media, Inc.", 2014.
52. Harman, Mark, and Bryan F. Jones. "Search-based software engineering." Information and software Technology 43.14 (2001): 833-839.
53. Harman, Mark, S. Afshin Mansouri, and Yuanyuan Zhang. "Search based software engineering: A comprehensive analysis and review of trends techniques and applications." Department of Computer Science, King's College London, Tech. Rep. TR-09-03 (2009): 23.
54. Harman, Mark, S. Afshin Mansouri, and Yuanyuan Zhang. "Search-based software engineering: Trends, techniques and applications." ACM Computing Surveys (CSUR) 45.1 (2012): 11.
55. McMinn, Phil. "Search-based software testing: Past, present and future." Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on. IEEE, 2011.

56. Harman, Mark. "The current state and future of search based software engineering." 2007 Future of Software Engineering. IEEE Computer Society, 2007.
57. Survey of multi-objective optimization methods for engineering
58. evolutionary algorithms for solving multi-objective problems
59. Multi-objective optimization using genetic algorithms tutorial
60. K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley & Sons, Inc., 2001
61. Evolutionary Algorithms for Multiobjective Optimization- Methods and Applications
62. Deb, Kalyanmoy, et al. "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II." International conference on parallel problem solving from nature. Springer, Berlin, Heidelberg, 2000.
63. Durillo, Juan J., and Antonio J. Nebro. "jMetal: A Java framework for multi-objective optimization." Advances in Engineering Software 42.10 (2011): 760-771.
64. Khan, Mohd Ehmer, and Farmeena Khan. "Importance of software testing in software development life cycle." International Journal of Computer Science Issues (IJCSI) 11.2 (2014): 120.
65. Saini, Gaurav, and Kestina Rai. "An analysis on objectives, importance and types of software testing." International Journal of Computer Science and Mobile Computing 2.9 (2013): 18-23.
66. Ahamed, S. S. "Studying the feasibility and importance of software testing: An Analysis." arXiv preprint arXiv:1001.4193(2010).
67. Zitzler, Eckart, and Lothar Thiele. "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach." IEEE transactions on Evolutionary Computation 3.4 (1999): 257-271.
68. Zitzler, Eckart, Marco Laumanns, and Lothar Thiele. "SPEA2: Improving the strength Pareto evolutionary algorithm." TIK-report 103 (2001).
69. Nebro, Antonio J., et al. "MOCcell: A cellular genetic algorithm for multiobjective optimization." International Journal of Intelligent Systems 24.7 (2009): 726-746.
70. Zitzler, Eckart, and Simon Künzli. "Indicator-based selection in multiobjective search." International Conference on Parallel Problem Solving from Nature. Springer, Berlin, Heidelberg, 2004.

Appendix A

- Datasets Google Drive Link:

https://drive.google.com/drive/u/3/folders/1zIqLoLD-H7gG_63oAfxT-PVnO0d83S

- Thesis Github Link:

<https://github.com/HibaMG/jMetal-master-2>

Note: the project code is a private repository on Github therefore please contact me to get access: hiba.mg.g@gmail.com