UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Service and Information System Engineering
Campus Diagonal Nord
Edifici Ω
C. Jordi Girona, 1-3.
08034 Barcelona
Spain
+34 93 413 78 39
`www.essi.upc.edu`

ISIMA
INSTITUT SUPÉRIEUR D'INFORMATIQUE DE MODÉLISATION ET DE LEURS APPLICATIONS

Institut Supérieur d'Informatique, de
Modélisation et de leurs Applications
1 rue de la Chebarde
TSA 60125
CS 60026
63178 Aubière Cedex
France
`www.isima.fr`

Experimenting with Genetic Algorithms to resolve the Next Release Problem

Valentin Elvassore

July 2016

Master Thesis
Master in Innovation and Research in Informatics
Specialization Service Engineering

ESSI Supervisors: David Ameller, Xavier Franch Gutiérrez
ISIMA Supervisor: Vincent Barra

Acknowledgements

I would like to acknowledge all the people who helped me before and during my thesis.

First, I would like to thank Xavier Franch Gutiérrez and David Ameller who proposed the subject of the thesis and followed the progress of my work helping me and organising frequent meetings.

Then, I would also like to thank Vincent Barra and the administrative staff of ISIMA. In addition to helping me organise my internship at ESSI, they also ensured that everything was passing fine.

Finally, I am grateful for the assistance provided by the staff of both ESSI and FIB to facilitate my incorporation and my daily work.

Abstract

Nowadays, there are more and more software and applications which are often provided through a sequence of development cycles. In this context, a problem emerges: how to determine which features, among the ones requested by their clients, have to be developed during the next cycle. This is the Next Release Problem. The main issue of this problem is to maximise the value of the next release while minimising its development cost this is why this problem is considered as multi-objective. Due to its complexity this problem is classified as NP-hard, therefore it is unsolvable by exact techniques so an appropriate way to resolve it is using heuristics such as genetic algorithms.

In this thesis, the Next Release Problem is reformulated in order to better fit with current research challenges, considering the available development resources and assigning employees only to features they are skilled for. Indeed, the cost is considered as human hours instead of money and the value as priority instead of customer importance. This reformulation allows producing a precise planning of features to develop.

This thesis consists first in gathering knowledge about the Next Release Problem and about its resolution methods, especially on genetic algorithms. After that the aim is to implement this problem into a java software using the jMetal framework which provides all the necessary tools to solve multi-objective problems. This implementation has to consider the precedence constraints between features, the availability of the human resources and the skills they possess have to match with those requested by the features to be developed. Moreover, as the traditional Next Release Problem has to be on budget, our version has to respect the end date of the cycle development.

To attain these objectives, two programs are developed. The first is a data generator which creates features, employees and skills according to parameters in order to be processed by the Next Release Problem solver. The second is an interface that allows the user to execute a parametrised algorithm on a generated data set. These two programs reinforce the tests done previously and ensure that the solver works normally whatever the processed data.

Finally, as a universal better algorithm does not exist for solving all the multi-objective problems, the aim is to define an experiment method and to apply it in order to determine which genetic algorithm better solves the Next Release Problem as it is formalised in this thesis. To do this, it is necessary to be able to compare the results of two different algorithms on the same instance of the Next Release Problem.

Then, in order to match with real cases of the Next Release Problem, some real data coming from a company participating in the SUPERSEDE H2020 project was used as a reference to estimate the number of employees, the number of skills and the number of precedence constraints engaged for developing a certain amount of features.

Concerning the experiment, it was decided to realize three instances: one which considers the size of the problem, one which attaches importance to the number of employees, keeping constant the number of features and the last which varies the number of features with a constant number of resources.

This thesis considers the following genetic algorithms to solve the Next Release Problem: the Multi-Objective Cellular genetic algorithm (MOCell), the Non-dominated Sorting Genetic Algorithm II (NSGA-II), the Pareto Envelope-based Selection Algorithm II (PESA-II) and the Strength Pareto Evolutionary Algorithm II (SPEA-II).

These results figure out that MOCEll is the genetic algorithm which finds the better solutions in all the three experiments. It is also the faster one. On the other hand, PESA-II has shown the worst results of the genetic algorithms experimented. Between these two extremes, NSGA-II and SPEA-II can provide good results in reasonable times, especially when the size of the problem is high. Furthermore, it is observed that the rate of employees by feature does not influence the results quality with MOCell as it does for the PESA-II and SPEA-II algorithms.

Concerning the computing time, there are some variations depending on the algorithm: the faster is MOCell which can resolve the Next Release Problem in less than a second when SPEA-II, the more time-consumer, needs 10 seconds. About the realization of each experiment protocol, it lasts between 120 and 280 minutes.

Key words: next release problem; genetic algorithms; experimenting; jMetal

French Summary

Introduction

De nos jours, on fabrique de plus en plus de logiciels et d'applications. Les entreprises adoptent souvent un développement par cycle pour leur réalisation afin d'avoir un retour rapide sur ce qui a été fait et pouvoir s'adapter à de nouvelles demandes. La résolution du Next Release Problem (le problème de la prochaine version) permet de sélectionner, parmi toutes les fonctionnalités souhaitées par les clients, une liste de fonctionnalités à développer lors du prochain cycle de développement.

Ce problème met en avant deux objectifs en conflit à réaliser que sont la minimisation du coût de développement et la maximisation de la valeur apportée aux clients les plus importants. Ce problème a une trop grande complexité pour être résolu à l'aide de méthodes exactes et nécessite l'emploi d'heuristiques. Parmi ces méthodes, cette thèse se focalise sur les algorithmes génétiques et devra déterminer lequel fournit les meilleures performances pour la résolution du problème.

Lors de cette thèse, après une première étape d'apprentissage sur le problème et ses méthodes de résolution, il sera nécessaire d'adapter sa définition puis de créer un programme exécutant un algorithme choisi sur des instances du Next Release Problem. Ces instances seront créées par un second programme à développer qui devra utiliser des valeurs paramétrables pour une génération au plus près des cas réels. Finalement, une méthode d'expérimentation devra être trouvée et utilisée pour déterminer quel est l'algorithme génétique le plus adapté à notre version du Next Release Problem.

Contexte

État des lieux

Une partie importante de cette thèse a été consacrée à la recherche d'informations et d'articles afin d'en connaitre plus sur le Next Release Problem et ses méthodes de résolutions et plus spécifiquement sur les algorithmes génétiques.

Le Next Release Problem considère habituellement une liste de clients avec leur importance relative à l'entreprise ainsi que les fonctionnalités que chacun veut voir réalisées. Dans cette thèse, le problème a été reformulé pour s'adapter au projets de recherche, tenir compte des ressources disponibles et pouvoir produire un planning précis au lieu de la simple liste de fonctionnalités à développer.

La version utilisée dans cette thèse ne considèrera plus des clients mais un niveau de priorité affecté à chaque fonctionnalité et ce sera la somme des priorités des tâches planifiées qui devra être maximisée. En ce qui concerne le coût de développement d'une fonctionnalité, il a été remplacé par le nombre d'heures de travail nécessaire à sa réalisation. Le deuxième objectif consiste à minimiser le nombre d'heures nécessaire à la réalisation des fonctionnalités planifiées, tout en ne dépassant pas la date de fin du cycle de développement. C'est l'optimisation de ces deux objectifs contradictoires qui fait de ce problème un problème multi-objectifs.

Pour résoudre ce problème, cette thèse devra déterminer quel est l'algorithme génétique qui trouve les meilleures solutions. Les algorithmes génétiques font partie de la famille des algorithmes évolutionnistes et se comportent comme décrit sur la figure 1. Ils commencent par générer une population de base puis vont successivement sélectionner des individus, les modifier grâce à des mutations et des croisements puis les évaluer plusieurs fois jusqu'à ce qu'une condition soit atteinte (un nombre d'itérations, une qualité attendue, . . .).

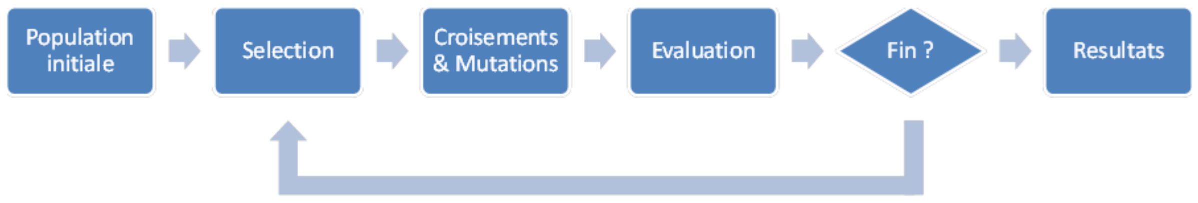


Figure 1: Fonctionnement des algorithmes génétiques

Planning

Après deux semaines de recherches et de lectures sur le contexte de la thèse, nous avons décidé de la découper en trois parties comme sur le diagramme de Gantt de la figure 2. La première de ces 3 étapes consiste à réaliser l'implémentation du problème et de sa résolution en un programme java. La seconde devra montrer que la première partie fonctionne correctement en permettant de générer des instances du problème et en les résolvant grâce à un algorithme choisi. Enfin, c'est lors de la dernière étape que sera définie et appliquée la méthode d'expérimentation qui déterminera quel algorithme génétique apporte les meilleures performances pour résoudre notre version du Next Release Problem.

À la fin de la thèse, le diagramme a été actualisé tel que sur la figure 3. Ce diagramme montre que la première des trois étapes a duré beaucoup plus longtemps que prévu. En effet, cette étape a nécessité des recherches et un temps de familiarisation des outils. Les deux étapes suivantes ont été raccourcies et non pas eu besoin du temps initialement

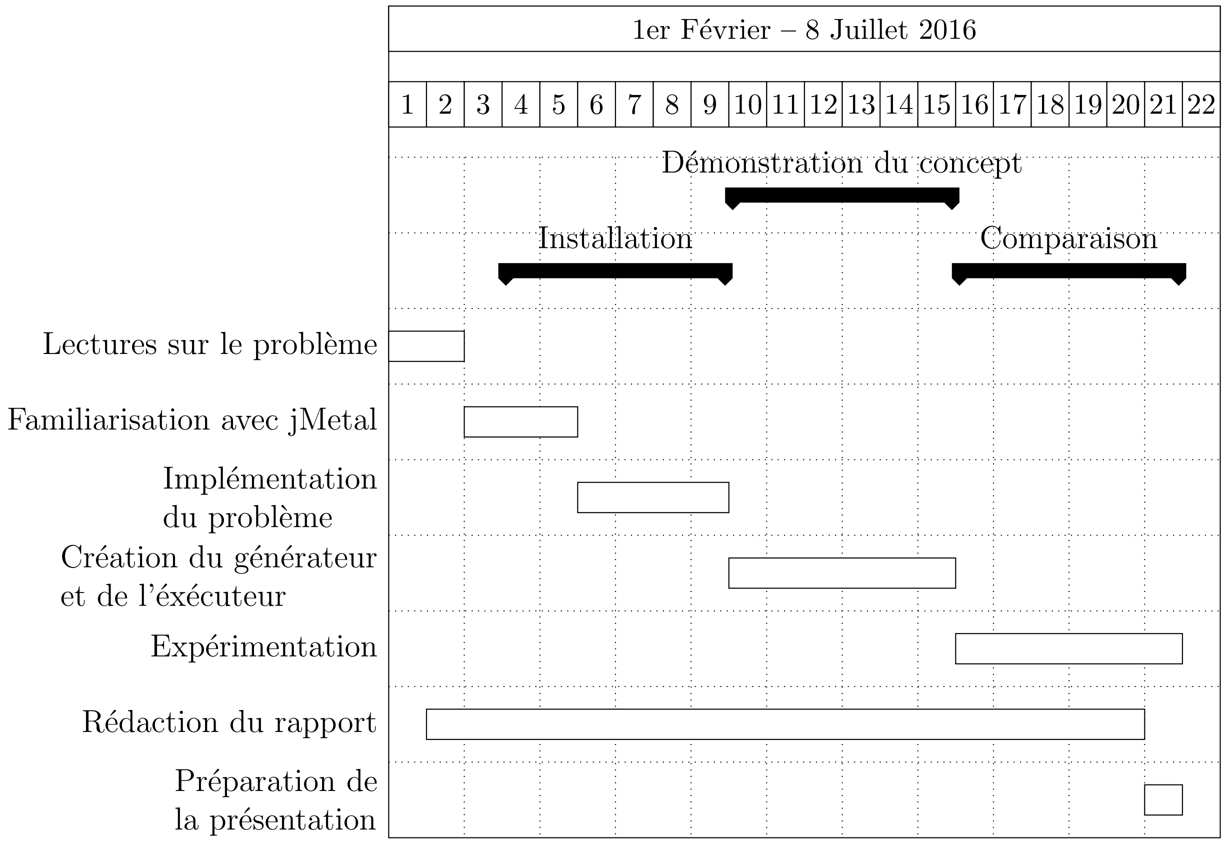


Figure 2: Diagramme de Gantt initial

planifié puisque la majorité des recherches avait déjà été faite.

Outils

Pour l'implémentation des programmes à fournir durant cette thèse, il a été imposé le langage java et l'utilisation de la bibliothèque jMetal. Cette bibliothèque fournit de nombreux outils pour résoudre des problèmes à l'aide de méta-heuristiques et propose déjà l'implémentation des algorithmes les plus connus.

De plus, certains outils supplémentaires ont été utilisés pour le suivi et la qualité du projet:

***Git*:** C'est un logiciel de controle de version très utilisé pour le développement informatique. Je l'ai utilisé afin de conserver un historique des changements, pour partager l'avancement avec mes superviseurs et pour conserver une sauvegarde sur un serveur.

***Trello*:** C'est une application web qui permet de suivre l'avancement d'un projet. Elle propose de définir des tâches sous formes de cartes que l'on peut déplacer entre les états d'avancement: à faire, en cours ou terminée.

***JUnit*:** C'est un cadre logiciel d'outils de tests unitaires pour le langage java. Avec cet

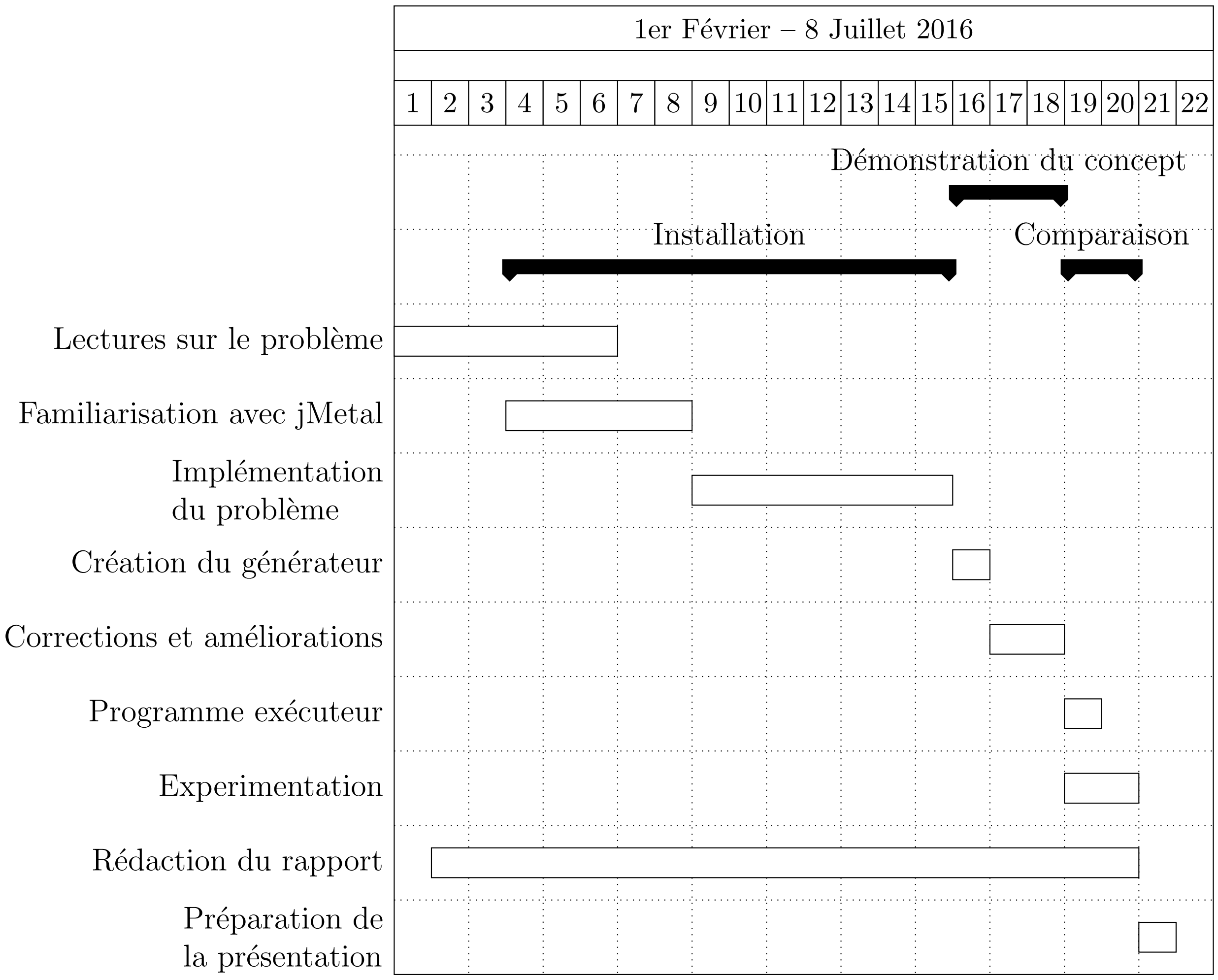


Figure 3: Diagramme de Gantt final

outil, j'ai pu m'assurer que le comportement du programme continuait de fonctionner malgré l'ajout de nouvelles fonctionnalités.

***JFreeChart*:** C'est une bibliothèque java qui permet de présenter des données sous forme de graphiques. Elle est très complète et propose toutes sortes de graphiques, ce qui m'a permis de choisir le plus adapté pour présenter les résultats des experimentations.

Développement

Implémentation du problème

La première étape du développement a consisté en l'implémentation du problème avec l'utilisation de la bibliothèque jMetal. Cela a donné lieu à la création de classes dérivant des entité de base de la bibliothèque comme présenté sur le diagramme de classes de la figure 4.

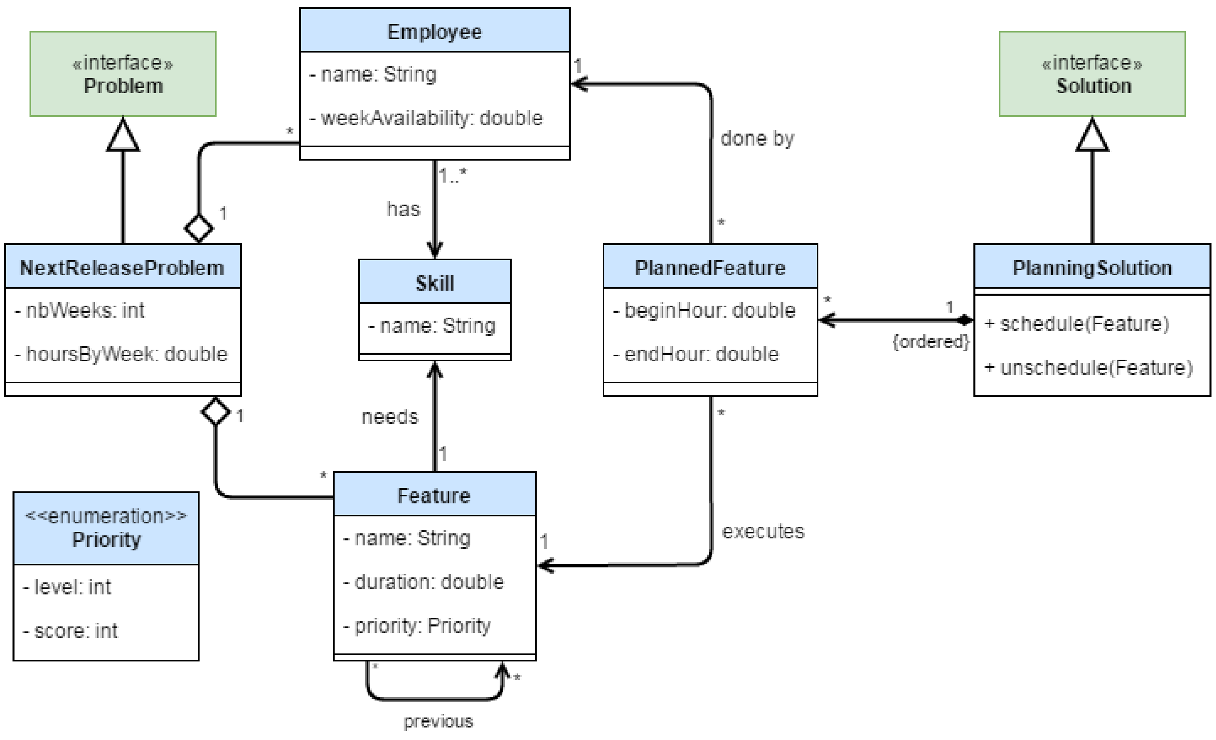


Figure 4: Diagramme de classes du coeur de l'implémentation

Les classes servant à la définition et à la résolution du problème multi-objectifs sont *NextReleaseProblem* qui définit les données du problème et qui évalura les *PlanningSolution* qui contiennent les variables du problème. Chaque solution contient en effet une liste de fonctionnalités à implémenter dans la prochaine itération dans l'ordre de leur plannification.

Outres ces deux classes fondamentales, on trouve les principales entités du problème : *PlannedFeature* qui regroupe la *Feature* à réaliser et l'*Employee* qui s'en chargera, *Skill* qui fait le lien entre une fonctionnalité et les employés qui sont qualifiés pour la réaliser et *PriorityLevel* qui permet de définir la valeur ajoutée pour chaque fonctionnalité.

Il a aussi fallut surcharger les opérateurs des algorithmes génétiques que sont la mutation et le croissement. L'opérateur de mutation passes toutes les fonctionnalités en revue et décide de les modifier ou d'en ajouter de nouvelle avec une probabilité $P_m = \frac{1}{nbtaches}$ pour chaque fonctionnalité. Quant à l'opérateur de croisement, il coupe les parents en deux selon une probabilité $P_c = 0.8$ et inverse les deux fins en faisant attention de ne pas planifier une tâche deux fois.

Finalement, le résultat de la résolution d'un problème produit un planning des fonctionnalités à développer, reliées aux employés qui en sont en charge.

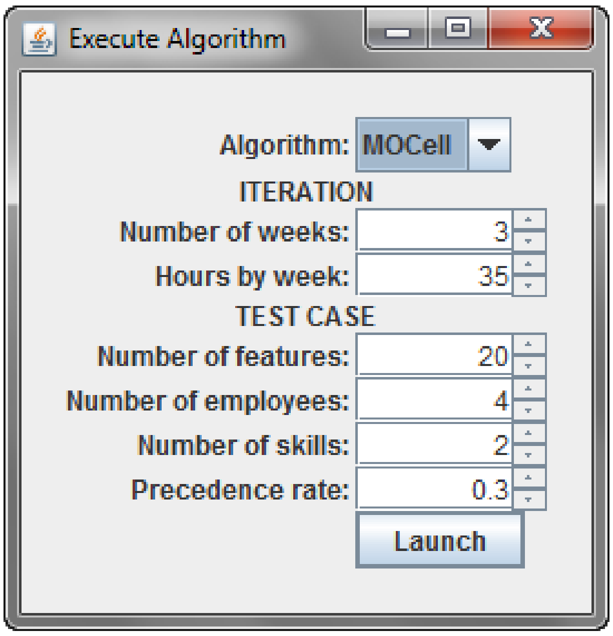
Un ensemble de tests a été développé et exécuté à chaque modification importante pour s'assurer que le programme continue de fonctionner comme attendu.

Utilisation de l'implémentation

Après avoir implémenté le prolbème et sa résolution, deux programmes ont été créés. Le premier est un générateur d'instances du problème qui permet d'effectuer des tests sur des problèmes générés aléatoirement. Ce générateur est en charge de créer les listes d'employés, de fonctionnalités et de compétences.

Afin de générer des instances réalistes des problèmes, il a été extrait des valeurs qui relient le nombre de fonctionnalités aux nombres des autres entités grace à des données fournies par des entreprises participant au projet SUPERSEDE.

Le second programme permet de générer une instance du problème et de lancer sa résolution par l'algorithme voulu à travers une interface (figure 5) qui permet de paramétrer cette exécution.



Algorithm: MOCell
Number of weeks: 3
Hours by week: 35
Number of features: 20
Number of employees: 4
Number of skills: 2
Precedence rate: 0.3

Figure 5: Interface graphique exécutant les algorithmes

Expérimentation

L'implémentation du problème et de son générateur étant fonctionnels, la dernière étape consiste en la définition et l'exécution d'une méthode d'expérimentation. J'ai d'abord défini un indicateur de qualité basé sur la valeur des objectifs d'une solution pour comparer les résultats des différents algorithmes. Il a ensuite été décidé de réaliser trois expériences:

- La première expérience fait varier la taille du problème pour voir si quel est le meilleur algorithme pour une taille donnée.

- La deuxième expérience considère un nombre constant d'employés et fait varier le nombre de fonctionnalité pour voir si cela influe sur le meilleur algorithme.

- Finalement, la dernière expérience fait varier le nombre d'employés pour un nombre constant de fonctionnalités pour voir si un algorithme se comporte mieux lorsque les ressources sont limitées.

Ces trois expériences ont donné un résultat similaire, visible sur la figure 6, qui est que l'algorithme MOCEll apporte des meilleures solutions à problème identique quelque soit la taille du problème. Il met aussi en exergue que l'algorithme PESA-II donne les plus mauvais résultats. Les algorithmes NSGA-II et SPEA-II donnent des résultats intermédiaires voire égaux lorsque le nombre de fonctionnalités à développer et égal au nombre d'employés.

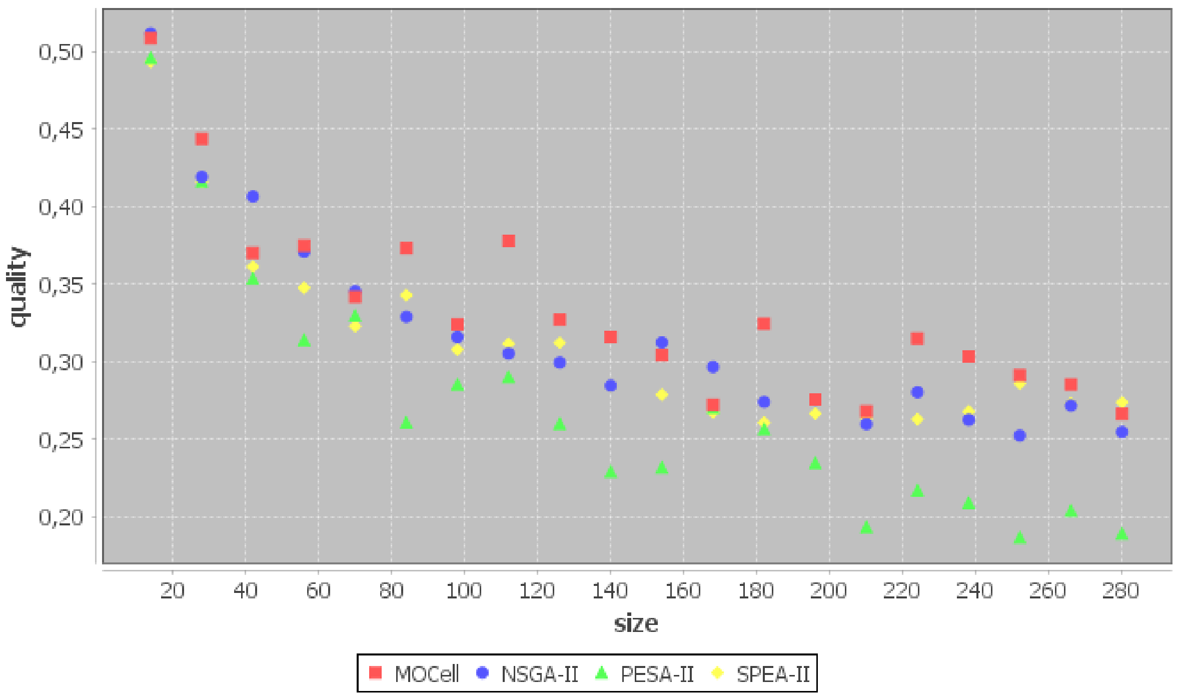


Figure 6: Résultat des expérimentations

En terme de performances, MOCell est aussi l'algorithme le plus rapide (moyenne de 50 ms par exécution) alors que l'algorithme SPEA-II met 200 fois pus de temps (moyenne de 10 secondes).

Conclusion

Cette thèse m'a permis d'acquérir de nombreuses connaissances sur le Next Release Problem et ses méthodes de résolution mais aussi plus généralement sur les méthodes de travail appliquées aux projets de recherche.

Lors de ces 22 semaines, le problème a été redéfinit puis implémanté. Il a été créé un générateur de tests pour pouvoir mesurer les performances des algorithmes lorsqu'ils sont exécutés sur des cas réalistes.

Finalement, un protocole d'expérience a été définit puis exécuté et a déterminé que MOCEll est l'algorithme qui, en plus d'être le plus rapide, est celui qui fournit les meilleures solutions et ce quelque soit la taille du problème ou le taux de ressources disponibles.

Du fait que les méthodes de résolution heuristiques sont en constante évolution, cette étude pourrait être complétée par la comparaison d'autres types d'algorithmes que les algorithmes génétiques

Introduction, Motivation and Goals

Nowadays, there are lots of software, applications and web services and their development is more and more split into development cycles. In fact, instead of developing all the features and deliver only once, the providers and the users often prefer to meet them during the development process in order to discuss and adapt the last improvements done according to their needs.

A difficulty that occurs when a software is developed using development cycles is to determine the order of the features to develop and even more what will be the features to develop in the next cycle. This is the objective of the Next Release Problem. It considers the resources available and determines what has to be developed in the next cycle considering the costs and the importance of each feature.

Because of the complexity of this problem[1], its resolution needs to be done using heuristic algorithms. Although this problem is well known, its resolution considering it as a multi-objectives problem is quite recent and the first paper published about it was in 2007[2].

In this thesis, we will only focus on the genetic algorithms. There are several genetic algorithm implementations, some better suited to solve some problems than others and the main objective of this thesis is to determine which of these performs better to solve the Next Release Problem.

This thesis was made in the context of the European project SUPERSEDE[1] whose global motivation is to incorporate more the users needs and feedback into the software development process (creation, evolution and adaptation).

The rest of the master thesis is organised as follows: First section provides information about the Next Release Problem and the genetic algorithms. The second section describes the context of related work. In section 3, I present the time organisation of the project while in section 4, I present the tools used. Section 5 is dedicated to the development of the thesis and the next one to the experimentation part. Then there is an evaluation and the last section is the conclusion.

[1]`www.supersede.eu`

Table of Contents

Acknowledgements **i**

Abstract **ii**

French Summary **v**

Introduction, Motivation and Goals **xiii**

1 Background **3**

1.1 The Next Release Problem 3

1.2 Genetic Algorithms 5

2 State of the art **9**

3 Planning **11**

4 Tools **13**

4.1 Presentation of the jMetal framework 13

4.2 Other tools used 14

5 Development **15**

5.1 Set-up 15

5.2 Proof of concept 22

6 Experimentation **27**

6.1 Quality of a solution 27

6.2 Experiment protocol 28

6.3 Results 28

6.4 Computing time 30

7 Evaluation **33**

7.1 Results 33

7.2 Planning 33

7.3 Personal comments 33

Conclusion **35**

References **36**

Appendices **39**

A Test Cases **41**

List of Figures

1 Fonctionnement des algorithmes génétiques vi
2 Diagramme de Gantt initial vii
3 Diagramme de Gantt final viii
4 Diagramme de classes du coeur de l'implémentation ix
5 Interface graphique exécutant les algorithmes x
6 Résultat des expérimentations xi

1.1 Illustration of the Next Release Problem 4
1.2 Main steps of Genetic Algorithms 6
1.3 Illustration of Mutations on binary examples 6
1.4 Illustration of a Crossover Operation 7

3.1 Initial Gantt Diagram 12

4.1 Class Diagram of the jMetal's Core Architecture 13

5.1 Class Diagram of the Problem Domain 16
5.2 Class Diagram of Employee's Weekly Planning 20
5.3 Example of a HTML outcome 21
5.4 Graphic Interface which executes algorithms 25

6.1 Results of Experiment 1 30
6.2 Results of Experiment 2 31
6.3 Results of Experiment 3 32

7.1 Final Gantt Diagram 34

A.1 Output of the simplest test case 41
A.2 Output of the simple optimisation test case 42
A.3 Output of the precedence test case 42
A.4 Output of the precedences test case 43
A.5 Output of the skills test case 43
A.6 Output of the employee overflow test case 44
A.7 Output of the employee overflow test case 44

Background

In this section, I'm going to explain and detail the main concepts of my thesis that are the Next Release Problem and genetic algorithms.

1.1 The Next Release Problem

1.1.1 Classical Definition

Any company involved in the development and the maintenance of a large and complex software product is faced with the problem of determining what should be in its next release. The goal to solving this problem is to obtain the list of the features to add in the next release, considering the following inputs[1]:

Features The list of enhancements needed by the customers, with their cost,

Precedence constraints Some features need the realisation of others to be computed,

Customers The list of the customers with their value for the company considering that we have to favour features needed by the most important client for the company,

Requirements The requirements needed for each feature development,

Budget The budget of the company for the cycle is limited and must not be exceeded.

The objective of solving the Next Release Problem is to select the feature of the most important clients within constraints and budget. This decision is very important for the company and can have serious consequences.

Satisfying each requirement entails spending a certain amount of resources which can be translated into *cost* terms. In addition, satisfying each requirement provides some *value* to the software development company. The problem is selecting the set of requirements that maximize total value and minimize required cost. These two objectives are conflicting, which is why the problem is considered as multi-objective[2].

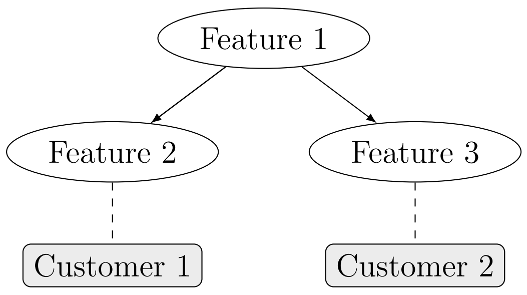


Figure 1.1: Illustration of the Next Release Problem

The Figure 1.1 illustrates an example of a Next Release Problem: assuming that the budget allows the development of only 2 features, we must choose to develop *Feature 1* because it is needed for the other features. Concerning the second enhancement to develop, *Feature 2* and *Feature 3* will be settled according to the importance of their respective customer for the company.

1.1.2 Thesis adaptation

In order to make the problem more generic and to fit with the needs of the SUPERSEDE project, we have adapted it considering the following changes:

- The Customer concept disappears, replaced by a priority level attributed to each feature,
- Apparition of Human Resources hours instead of cost: we attach a list of employees to the development cycle with their weekly availability,
- Apparition of a Skill concept: each feature needs a skill to be performed and can be executed only by employees that possess it,
- The global budget is replaced by an end date (described as a number of weeks and the number of working hours by week)
- The objective of maximising the value becomes that of maximising the number of features, weighted by their priority,
- The objective of minimising the cost becomes that of minimising the end date.

These changes allow the investigation projects to be considered as they are not strictly concerned by customers and as they consider the cost in term of human hours. Moreover, they enable the possibility to produce a precise planning instead of only obtaining the list of features to plan. Finally, it ensures to have the necessary resources to perform the scheduled features.

1.1.3 Formal definition

Let us consider the set of n possible features to execute in the iteration: $F = \{f_1, \dots, f_n\}$ with their corresponding positive duration $d_i > 0$; $i = 1, \dots, n$ and priority value p_i; $i = 1, \dots, n$. Moreover, we associate to each feature a variable $x_i \in \{0, 1\}$; $i = 1, \dots, n$. If $x_i = 1$ the feature f_i will be scheduled otherwise it will not be.

Thus, we consider the problem with the two following objective functions:

$$Minimise \sum_{i=1}^{n} d_i \cdot x_i$$

$$Maximise \sum_{i=1}^{n} p_i \cdot x_i$$

Besides, let us define two types of constraints:

- We have to assume a directed acyclic graph $G = (R, E)$ with E the set of precedence constraints, modeled by arcs (f, f') which means that the feature f' needs the feature f to be terminated to start.
- The development cycle lasts w weeks with h hours by week which means the end date e can not exceed $e = w \times h$.

The trouble with the Next Release Problem is that its solving time grows exponentially with its number of enhancements (it is NP-hard[1]) so that it is unthinkable to get the best solution with traditional solving tools with more than a dozen of features[2].

1.2 Genetic Algorithms

Nowadays, there are some problems that we can not resolve in a reasonable time. For this reason, we can use heuristics. These algorithms do not certify that the optimal solution will be obtained but the result will be close and produced in due time.

In order to solve multi-objective problems, these algorithms execute some operations several times in order to improve the quality of the solutions found and stop only after satisfying a *termination condition* which can be reached by a number of iterations, by an expected result or by a computing time for instance.

The core of multi-objective algorithms evaluate each objective in order to compare the solutions and determine which is the better one.

The comparison of two solutions has the aim of determining if one dominates the other or not. To do this, all the objective values of a solution have to be better (i.e. greater in case of maximisation and lower in case of minimisation) or equal than the ones of the other solution to assert that it dominates (and at least one of these values has to be strictly better).

At the end of comparing all the solutions, the algorithm gives as a result a list of non-dominated solutions.

Genetic algorithms belong to the larger category of evolutionary algorithms, and generate solutions to optimization problems using techniques inspired by natural evolution. Indeed, after creating a base population, the algorithm will apply the three basics operations on its individuals: the selection, the mutation and the crossover (Figure 1.2).

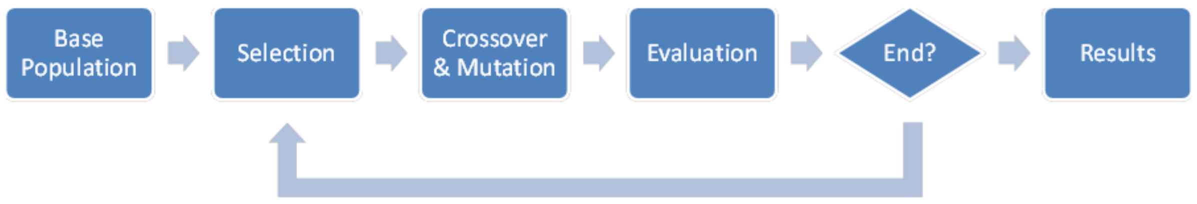


Figure 1.2: Main steps of Genetic Algorithms

1.2.1 The Selection Operation

This operation consists in selecting individuals in the population in order to breed a new generation[3]. There are various strategies to do this: some of these do it entirely randomly which warrants the population diversity and others favour the best ones (such as the *Tournament Selection*).

1.2.2 The Mutation Operation

In order to make more diversity inside the population, each genetic algorithm has a mutation operator. As it can be seen on the Figure 1.3, various mechanisms exist to do a mutation: the alteration, the exchange, the insertion and the deletion. In order to progress gradually, the mutation probability has to be well chosen. In publications, it is recommended to use 0.001, 0.01 or $\frac{1}{length}$[4].

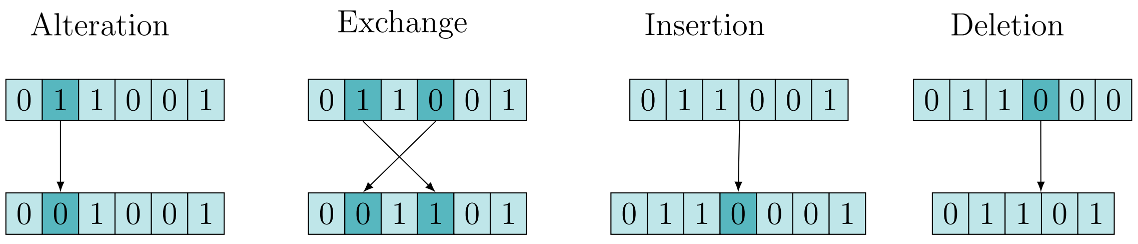


Figure 1.3: Illustration of Mutations on binary examples

1.2.3 The Crossover Operation

Crossover is a process of taking more than one parent solutions (commonly two) and producing children solutions from them. The Figure 1.4 shows an example with two parents cut after the third bit and engendering two children. There could be some crossover methods more complicated that cut the parents into more than two parts or that have a limit number of bits than they can cross but the general principle stays the same.

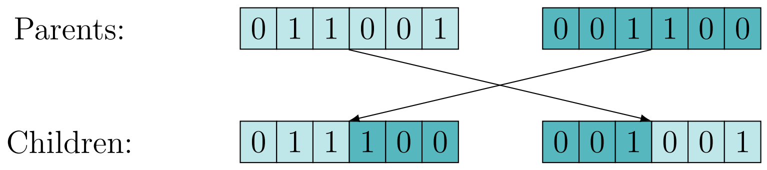


Figure 1.4: Illustration of a Crossover Operation

1.2.4 Existing Genetic Algorithms

There are many genetic algorithms and I am going to briefly introduce the ones used in this thesis that have demonstrated their performance in solving multi-objective optimisation problems.

NSGA-II

Non-dominated Sorting Genetic Algorithm II is a well known multi-objective genetic algorithm[5]. It includes a non-dominated sorting procedure and a constraint mechanism using a modified definition of domination in order to not use penalty functions. Moreover, it uses crowding distance in order to guarantee diversity and spread of solutions. Finally, it implements elitism which stores all non-dominated solutions, and hence enhancing convergence properties.

MOCell

MOCell is a cellular genetic algorithm for solving multi-objective problems. Its main feature is to conserve an external archive of non-dominated solutions and randomly insert some of them into the current population[6].

PESA-II

Pareto Envelope based Selection Algorithm II is an algorithm that instead of attaching a fitness value to each solution, the fitness value is assigned to hypercubes of the objective space[7].

SPEA-II

Strength Pareto Evolutionary Algorithm is an algorithm that archives the non-dominated solution apart from the population which will maintain a front of the better solutions found while it can try to optimise the inside population solutions[8].

State of the art

The Next Release Problem

As the Next Release Problem is often present in software development, it is well documented. The first paper I have read about it is *The Next Release Problem* written by A.J. Bagnall, V.J. Rayward-Smith and I.M. Whittley in 2001[1] which taught me about the modelization of the problem and about its complexity. In addition to providing some typical instances in order to test the results obtained by its resolution, this paper also uses solving methods such as CPLEX and GRASP methods.

Solving Algorithms

At the beginning of the thesis, I was asked to read *Solving the Large Scale Next Release Problem with a Backbone-Based Multilevel Algorithm* written by J. Xuan, H. Jiang, Z. Ren and Z. Luo in 2012[9] which proposes a multilevel approach to solve the Next Release Problem. It is an experiment of executing an approximate and a soft backbone-based algorithm on large generated instances of the Next Release Problem. The paper demonstrates that these algorithms better performs to solve large instances of the Next Release Problem than direct solving approach.

After that, I focused on the problem as a multi-objective problem, reading *The Multi-Objective Next Release Problem* written by Y. Zhang, M. Harman and S.A. Mansouri in 2007[2] which is the first paper published about it. This paper proposes to solve different instances of the Next Release Problem using three different methods: NSGA-II, Pareto GA and Single Objective GA. It resulted that the NSGA-II outperformed the others both in terms of diversity and results but the paper only consider problems without precedence constraints. Additionally, the paper mentions that exceeding 20 features, the problem will need a metaheuristic technique to be solved.

In order to gain knowledge about genetic algorithms, the paper *A summary and comparison of MOEA algorithms*[8] gave an overview of a large panel of evolutionary algorithms in their different versions.

Comparison and Experimenting

Furthermore, with the aims of determining if there is a genetic algorithm that is better than the others, I have read about the No Free Lunch Theorem in Optimization[10] which clearly indicates that an evolutionary algorithm can be the best for a problem area but will be outperformed as the problem changes.

Finally, I have read about experimenting algorithm on the Next Release Problem with *A Study of the Bi-Objective Next Release Problem*[11] which expresses the results by charts including both score and cost of solutions.

Planning

The internship lasts 22 weeks, from February 1st to July 8th, 2016. After 2 weeks of reading about the Next Release Problem and genetic algorithms, we made a plan. Of the remaining 19 weeks, we decided to split into 3 main steps of six weeks each.

1. *Set-up:* The main objective of this step is to implement the problem and its resolution. To do this, the step starts by reading and learning about the Next Release Problem, genetic algorithms and the tools to use. After that, the idea is to improve the program between each meeting with my supervisors every 10 days.

2. *Proof of concept:* During this step, two applications will be created: the first that will generate data adapted to the Next Release Problem and the second that will use the first to generate a set of data and then applying a chosen genetic algorithm on it. Both of these application have to be configurable by parameters. This step ensure that all that have been done during the Set-up step works fine or to correct the bugs otherwise.

3. *Comparison:* This last step consists in defining an experimenting method, to implement it and finally to extract the results from these experiments.

This led to the initial planning on Figure 3.1.
The real final planning can be found on Figure 7.1 in page 34.

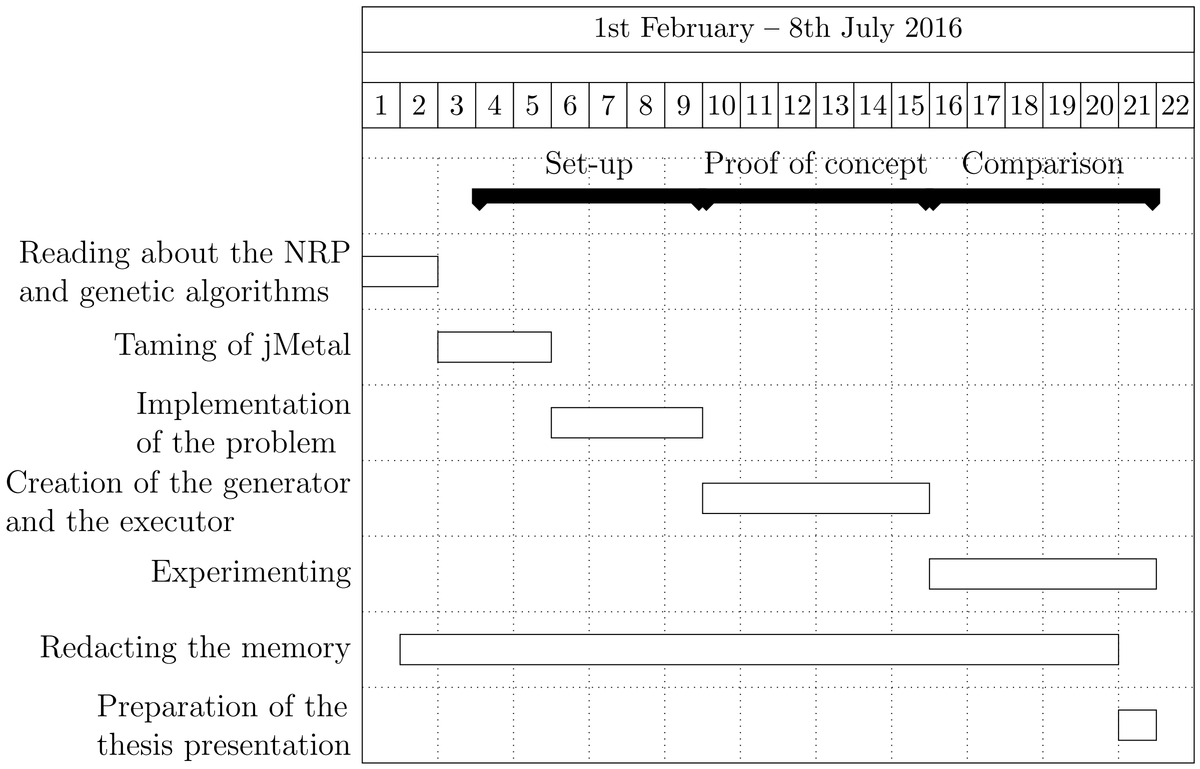


Figure 3.1: Initial Gantt Diagram

Tools

4.1 Presentation of the jMetal framework

In order to use existing implementation of genetic algorithms, it was decided to use a framework. My supervisors have chosen jMetal on it 5.0 version[12] which integrates several mechanisms to resolve multi-objective problems and some tools for experimenting.

The jMetal (stands for Metaheuristic Algorithms in Java) is an object-oriented Java-based framework for multi-objective optimization with metaheuristics. It offers a based structure in order to apply its various algorithms to any multi-objective problem. The Figure 4.1 shows the four main interfaces of the library's core structure:

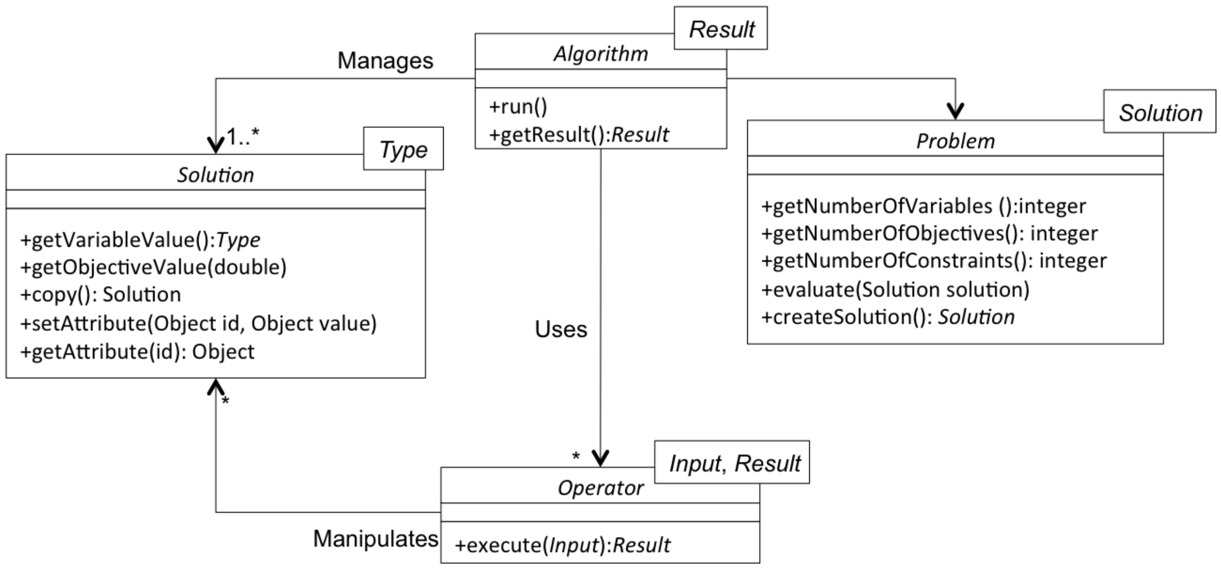

Result
Algorithm
+run()
+getResult():Result
Manages
1..*
Type
Solution
+getVariableValue():Type
+getObjectiveValue(double)
+copy(): Solution
+setAttribute(Object id, Object value)
+getAttribute(id): Object
Solution
Problem
+getNumberOfVariables ():integer
+getNumberOfObjectives(): integer
+getNumberOfConstraints(): integer
+evaluate(Solution solution)
+createSolution(): Solution
Uses
*
*
Input, Result
Operator
+execute(Input):Result
Manipulates

Figure 4.1: Class Diagram of the jMetal's Core Architecture

***Problem*:** In addition to define the variables, the objectives and the constraints of the problem to solve, this interface is responsible for evaluating the solutions,

***Solution*:** This is a solution of the problem, containing the variables and having objectives values,

***Algorithm*:** This interface is implemented by all the algorithms available in jMetal,

***Operator*:** This is the interface for all the operators: selection, mutation and crossover.

The strength of this core structure, with interfaces and generics, is that it defines a base for commons situation but we can also redefine the behaviour of some entities appropriately to our problem by implementing subclasses.

Out of this core structure, jMetal also provides other interesting tools such as an algorithm execution timer and solutions utilities (e.g. comparators, getter method for the best solution).

4.2 Other tools used

In order to manage properly the project and to share the advancements with my supervisors, I have used some more tools:

***Git*:** This is a version control software widely used for software development. I have used it in order to keep an history of the changes, to share it with my supervisors and to have a backup on a *github* server.

***Trello*:** This is a web application that tracks the progress of a project. Its purpose is to define the tasks as cards that we can move from various states: to do, in progress, done, ... Besides allowing to manage the progress of the project, this tool allows it to be shared with the other team members.

***JUnit*:** This is a unit testing framework for the Java programming language. With this tool, I could make sure that the behaviour of the program still works as expected throughout the project despite changes.

***JFreeChart*:** This is a java library to present data into charts. It is very useful because it proposes many types of charts and I could chose the most appropriate to present the experiment results.

Development

In this section, I am going to present the three steps of the development.

5.1 Set-up

The objective of this step was to implement the Next Release Problem and its resolution with genetic algorithms using the jMetal library.

5.1.1 Implementation

Core

After some trials with easier problem considering bits and integers problems in order to familiarise myself with *jMetal*, I could extend the core structure of the library (Figure 4.1 on page 13) to match with the Next Release Problem.

Thus, I created the following structures (class diagram on Figure 5.1):

NextReleaseProblem: This class implements the *Problem* interface of *jMetal* and will be responsible for evaluating solutions objectives and constraints. Moreover, this class contains problem data such as the list of employees, the list of features, the number of weeks of the iteration and the number of working hours by week.

PlanningSolution: This class contains the planned features and implements the interface *Solution* of *jMetal*. The order of the planned features in the list is its order of execution. It contains mainly two methods to modify this list : *schedule()* and *unschedule()*.

PlannedFeature: This is the variable of the problem. It encapsulates a *Feature* and the *Employee* who will achieve it. Moreover, it contains the beginning and ending hours. In this thesis we have considered that the feature can be executed by only one employee.

Feature: This is the feature to realise. It contains the information of the feature such as its *PriorityLevel*, the *Feature*s that need to be executed before its own realisation (precedence constraints) and the *Skill* needed (in this project, we will consider that a *Feature* needs only one *Skill*).

Employee: This is the human resource who can execute the features. It has a weekly availability (expressed in hours) and the list of *Skill*s it possesses.

PriorityLevel: This is an enumeration of priority levels from 1 (the most important) to 5. It contains a score to determine the global score of a *PlanningSolution*. We have decided that the score of a level is twice the one of the lower level because we are considering that doing a feature of the level i is equivalent to do two features of the level $i - 1$.

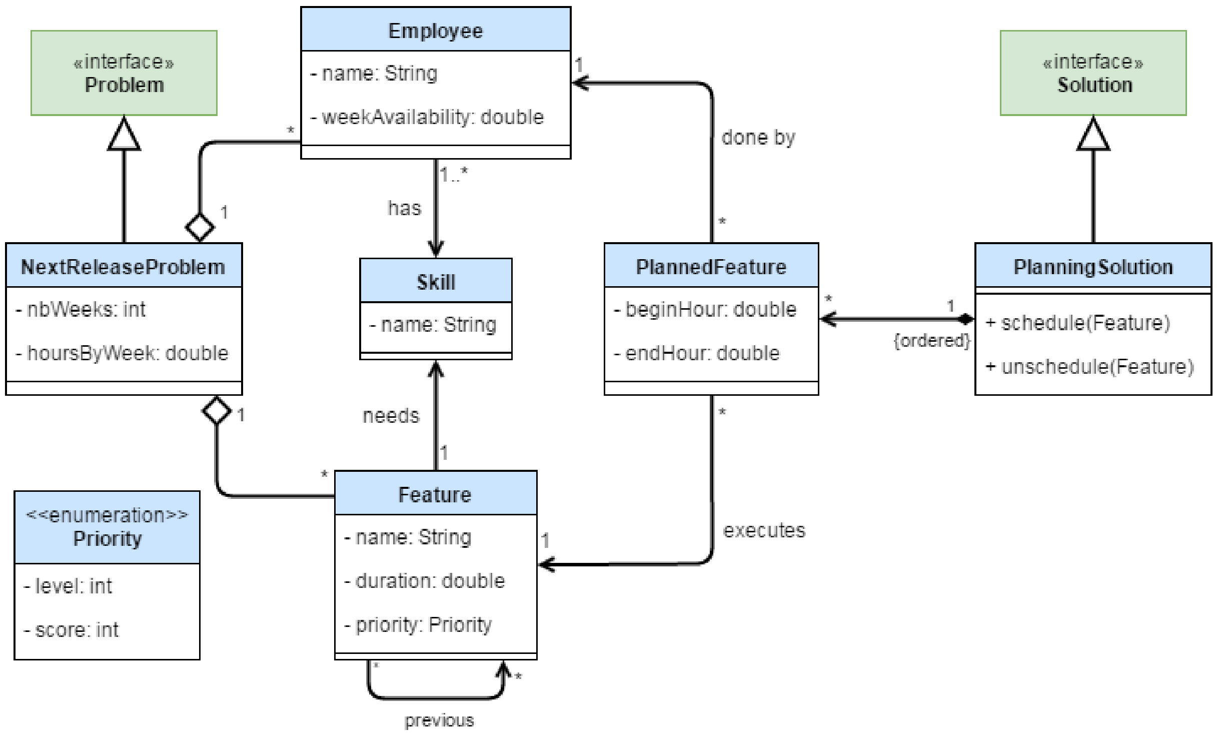


Figure 5.1: Class Diagram of the Problem Domain

Operators

After having done it, in order to make diversity, I have extended the *Operator* interface of jMetal to adapt the behaviour of the mutation and the crossover to the Next Release Problem.

Basically, the mutation operator will draw a random number between 0 and 1 for each feature and, if it is greater than the mutation probability, change the feature or the employee for the features already planned and add it for the ones which are not (Algorithm 1). I have chosen a probability of mutation $P_m = \frac{1}{\text{number of features}}$ which is

```
Algorithm 1: Mutation Algorithm
  Data: parent : The parent solution
  Result: child : The offspring solution
  child ← parent.copy();
  foreach plannedTask in child.getPlannedTasks() do
      if doMutation() then                    // random < mutation probability
          if newRandom < 0.5 then
              changeTask(plannedTask);
          else
              changeEmployee(plannedTask);
          end
      end
  end
  foreach undoneTask in child.getUndoneTasks() do
      if doMutation() then                    // random < mutation probability
          solution.schedule(undoneTask);
      end
  end
  return child;
```

often used for the Next Release Problem[2] and which will realise one change in average each time that the mutation operator is applied on a solution.

Concerning the crossover operator, it splits the two parents into two parts each and reverses it, taking care of not planning a feature twice (Algorithm 2). As it is recommended to chose a crossover probability between 0.5 and 1, I have chosen a crossover probability $P_c = 0.8$ which allows the production different solutions, keeping variety in the population[4].

I did not have to override the selection operator because jMetal already proposed its owns that are compatible with my implementation because they do not consider the variables but the constraints and objectives values in their process.

Objectives Evaluation

There are two objectives: minimise the end date of the planning and maximise the priority score (i.e. the sum of each feature priority score).

The end date objective will be a value between 0.0 (if there is no feature planned) and the end date of the iteration (*number of weeks* $\times$ *hours by week*). To obtain this value for a solution, we first have to attribute the begin and end dates of each planned feature such as presented in the Algorithm 3 and then to extract the last end date of the planned features.

Concerning the score objective, as there are some algorithms that only work with minimisation objectives, I have considered that a solution which has planned all the

Algorithm 2: Crossover Algorithm

```
Data: $parent1$, $parent2$ : The parent solutions
Result: $offsprings$ : The offspring solutions
$offsprings$.add($parent1$.copy());
$offsprings$.add($parent2$.copy());
if doMutation() then                          // random < crossover probability
    $minSize \leftarrow min(offsprings[0]$.getNumberOfPlannedTasks(),
     $offsprings[1]$.getNumberOfPlannedTasks());
    if $minSize > 0$ then
        $splitPosition \leftarrow random(1, minSize)$;
        $endChild1 \leftarrow parent1.getPlannedTasks().sublist(splitPosition)$;
        $endChild2 \leftarrow parent2.getPlannedTasks().sublist(splitPosition)$;
        foreach $plannedTask$ in $endChild2$ do
            $child1.unschedule(plannedTask)$;
        end
        foreach $plannedTask$ in $endChild1$ do
            $child2.unschedule(plannedTask)$;
        end
        foreach $plannedTask$ in $endChild1$ do
            $child1.schedule(plannedTask)$;
        end
        foreach $plannedTask$ in $endChild2$ do
            $child2.schedule(plannedTask)$;
        end
    end
end
return $offsprings$;
```

features will have a priority score of 0.0 and a solution that does not have a planned feature will get the worst possible score (the sum of each feature priority level). Indeed, to calculate this objective value for a solution, it only needs to sum the priority score of each planned feature and to subtract it from the worst score which is stored into the *NextReleaseProblem*.

Algorithm 3: Simplified Evaluation Algorithm

```
Data: solution: The solution to evaluate
solutions.resetHours();                    // set the begin and end hours to 0.0
foreach plannedFeature in solution.getPlannedFeatures() do
    $feature \leftarrow plannedFeature$.getFeature();
    $beginHour \leftarrow$ Max(getEmployeeAvailability(plannedFeature.getEmployee()),
      getMaxEndHour(feature.getRequiredFeatures());
    $plannedFeature$.setBeginHour($beginHour$);
    $plannedFeature$.setEndHour($beginHour + feature$.getDuration()) ;   // This
      will be updated later to consider the employee week availability
end
```

Constraints Evaluation

The system of constraints is already implemented in jMetal, I only needed to make the *NextReleaseProblem* class to implement the *ConstrainedProblem* jMetal interface, to define what are they and to make their evaluation method. There are four types of constraints:

- The respect of the skills,
- The overflow of the employee's weekly availability,
- The global overflow of the planning,
- The precedences between features.

For the first one, as it produced lots of constrained solutions, I have created a *Map* linking the Skills to the Employees that possess it in order to execute a feature so that an employee will be chosen only among the skilled employees. Furthermore this type of constraint cannot be violated anymore.

In order to improve the efficiency of the algorithm, I have also removed the employee's weekly availability overflow constraint by creating employee's weekly planning (Figure 5.2) and fill them independently of the number of weeks of the iteration. Thus I only need to check the global overflow.

The two last types of constraints are classically evaluated in the *evaluateConstraint()* method and the value of the constraint attribute of the solution is set to the number of

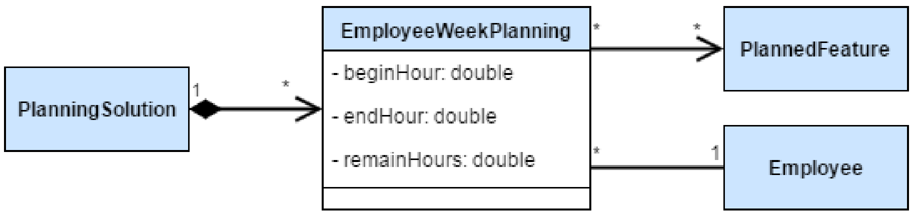


Figure 5.2: Class Diagram of Employee's Weekly Planning

this violated constraints. Moreover, 10% of the generated solution of the base population is generated taking into account the precedence constraints. This was done to ensure that some feasible solutions will exist but its low rate allows to keep variety in the population by generating random solutions in the rest of the cases.

5.1.2 Input files

In order to facilitate the tests, I have implemented the functionality in charge of reading the features, the employees and the skills from files. There are two types of files: those containing the features information and those which contain the employees data.

For the first one, each line corresponds to a feature which is identified by its name. A line contains the following fields, separated by a tabulation:

- the name (unique),
- the priority level (integer between 1 and 5),
- the duration (expressed in hours),
- the required skill name,
- the previous features names, separated by a comma.

Here is an example of a feature data file:

```
Feature 1       2       2.0     Skill 1
Feature 2       3       3.0     Skill 2 Feature 1
Feature 3       4       2.0     Skill 1 Feature 1
Feature 4       1       3.0     Skill 2 Feature 2, Feature 3
```

Concerning the employees data files, they have a similar structure: they contain the following fields, also separated by a tabulation:

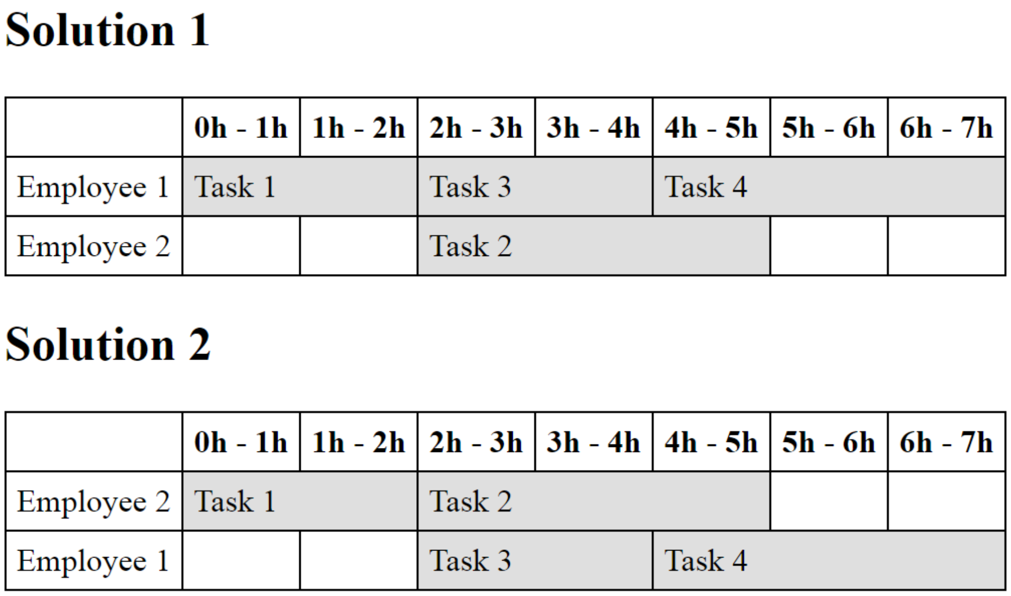
- the name (unique),
- the week availability (expressed in hours),
- the skills names he has, separated by a comma.

Here is an illustration for the employees data files:

```
Employee 1      20.0    Skill 1
Employee 2      15.0    Skill 1, Skill 2
```

5.1.3 Output

In order to have a graphical view of the planning and to quickly have an overview of the solution, I chose to generate the output into HTML (HyperText Markup Language) because it is compatible with most of the systems (readable by a web browser) and the structure of the file is as simple that I did not spend so much time to obtain a suitable view (Figure 5.3).



Solution 1

	0h - 1h	1h - 2h	2h - 3h	3h - 4h	4h - 5h	5h - 6h	6h - 7h
Employee 1	Task 1		Task 3		Task 4		
Employee 2			Task 2				

Solution 2

	0h - 1h	1h - 2h	2h - 3h	3h - 4h	4h - 5h	5h - 6h	6h - 7h
Employee 2	Task 1		Task 2				
Employee 1			Task 3		Task 4		

Figure 5.3: Example of a HTML outcome

5.1.4 Testing

In order to ensure that the program behaves as expected and continue to do it after adding enhancements, I have developed some test cases which I execute with JUnit. The following test cases were created as I went along developing new features (details can be found on Annex A):

Simplest: This is the simplest case with only one feature and one employee that tests if the developed core works normally and if the final solution is not empty and contains the feature, planned at the right dates.

Simple optimisation: This case is to check if the algorithm well distributes the two features between the two employees instead of attributing them to the same resource.

Precedence: With two dependent features, this test case ensures that the precedence constraints are respected by checking the order of the planned features.

Skill: This test case ensures that the program is taking into account the skills needed for each feature to attribute to the employees.

Overflow: Here, we are checking that the feature that overflows the iteration date is not planned. This is to be sure that the overflow constraint is well implemented.

Overflow optimization: In this test case, all the features cannot be planned because their duration exceeds the end date of the iteration. So, we have to check if only the features with the highest priority are planned and that the end date of the output planning is lower than the end date of the iteration.

Employee overflow: In this test case, the employee does not have time to realise a feature in a week so the test case checks if the feature is well distributed on the iteration weeks.

All these tests are executed after each important modification and check crucial information such as dates, objectives values and constraints.

5.2 Proof of concept

During this second stage, the objective was to create two programs: one which generates data set for the Next Release Problem and the other which can execute an algorithm taking into account some input parameters.

5.2.1 Generator

This program has to take three parameters as inputs: the number of features, the number of employees and the number of skills to generate.

After having generated a list of skills (which just consists in generating different names), we can generate the features as presented on the Algorithm 4. This algorithm basically determines the number of precedence constraints to generate and then pick up into the list of previous generated features to add the constraint.

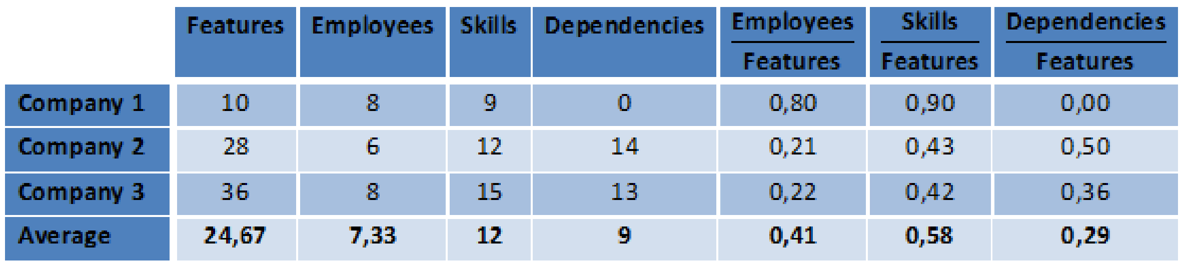
Finally, it generates the employee with a random weekly availability hours and random skills among the list.

After having generated the data, the program encapsulates it in a class with the aims to solve the Next Release Problem made from this data. It can also generate data files in order to conserve it and to execute experiments on this particular case.

Finally, in order to have data compatible with realist problems, I had to find out some key values to determine for instance how many employees there are in average

Algorithm 4: Features Generation

```
Data: $numberOfFeatures$: The number of features to generate; $skills$ : the list of
      skills available; $precedenciesRate$: the rate of precedencies by feature
Result: $features$: The generated features
$priorities \leftarrow$ PriorityLevel.values();
$remainPreviousConstraints \leftarrow$ round($numberOfFeatures \times precedenciesRate$);
$features \leftarrow$ new List($numberOfFeatures$);
for $i \leftarrow 0$ to $numberOfFeatures$ do
    $previousFeatures \leftarrow$ new List();
    if $features.size() > 0$ and $remainPreviousConstraints > 0$ then
        $probability \leftarrow remainPreviousConstraints/(numberOfFeatures - i)$;
        $possiblePreviousFeatures \leftarrow features$.copy();
        while $remainPreviousConstraints > 0$ and
         $possiblePreviousFeatures.size() > 0$ and $newRandom() < probability$ do
            $indexFeature \leftarrow$ newRadom($possiblePreviousFeatures$.size());
            $previousFeatures$.add($possiblePreviousFeatures$.get($indexFeature$));
            $possiblePreviousFeatures$.remove($indexFeature$);
            $remainPreviousConstraints = remainPreviousConstraints - 1$;
            $probability \leftarrow remainPreviousConstraints/(numberOfFeatures - i)$;
        end
    end
    $requiredSkill \leftarrow skills$.get(newRandom($skills$.size());
    $features$.add(new Feature(”Feature ” + i,
     $priorities$[newRandom($priorities$.length)], newRandomDuration(),
     $previousFeatures$, $requiredSkill$));
end
return $features$;
```



	Features	Employees	Skills	Dependencies	Employees / Features	Skills / Features	Dependencies / Features
Company 1	10	8	9	0	0,80	0,90	0,00
Company 2	28	6	12	14	0,21	0,43	0,50
Company 3	36	8	15	13	0,22	0,42	0,36
Average	**24,67**	**7,33**	**12**	**9**	**0,41**	**0,58**	**0,29**

Table 5.1: Release companies data

for x features. To do this, we analysed the real data coming from the three companies participating in the SUPERSEDE project (Table 5.1)

Of this data, I have extracted three interesting indicators:

- The rate of employees by feature of 0.4,
- The rate of skills by feature to 0.5,
- The rate of dependencies by feature to 0.3.

These values are used in a default execution of the program but can be changed in a configuration file.

As example, if the generator generates a 10-features problem data, there will be 4 employees, 5 skills and 3 dependence constraints.

5.2.2 Algorithm Executor

After creating the generator, the next step was to create a program which can receive parameters in order to:

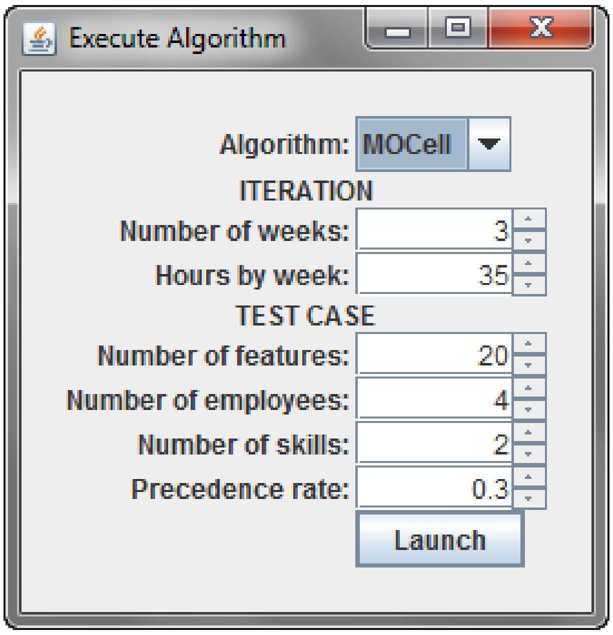
- Generate a data set,
- Perform an algorithm on it,
- Display the resulted planning.

Besides the parameters of the generator, this program has to receive some other inputs such as the iteration ones (number of weeks, hours by week) and the algorithm ones (population size, number of evaluation).

Moreover, the user has to chose which algorithm he wants to use. To define the implemented algorithm of this program, I looked at the jMetal documentation to see what are the subclasses of the *AbstractGeneticAlgorithm*. There were the following: *MOCell*, *NSGAII*, *PESA2*, *SMSEMOA*, *SPEA2* and *SteadyStateGeneticAlgorithm*.

As the *SMSEMO* and the *Steady State Genetic Algorithm* do not consider constraints when they evaluate solutions, the user of the program can finally choose between *MOCell*, *NSGA-II*, *PESA-II* and *SPEA-II* to solve the Next Release Problem with this program.

The program is usable by command line but I have also developed a basic graphic interface using swing. This interface can be seen on Figure 5.4



Algorithm: MOCell

ITERATION

Number of weeks: 3

Hours by week: 35

TEST CASE

Number of features: 20

Number of employees: 4

Number of skills: 2

Precedence rate: 0.3

Figure 5.4: Graphic Interface which executes algorithms

Experimentation

The last step of the thesis was the comparison of the algorithm. In addition to defining an experiment method, I also had to find out a way of comparing the results of the different algorithms.

6.1 Quality of a solution

In order to compare two solutions of two different algorithm executions for the same problem, I had to establish a way to evaluate the solution quality.

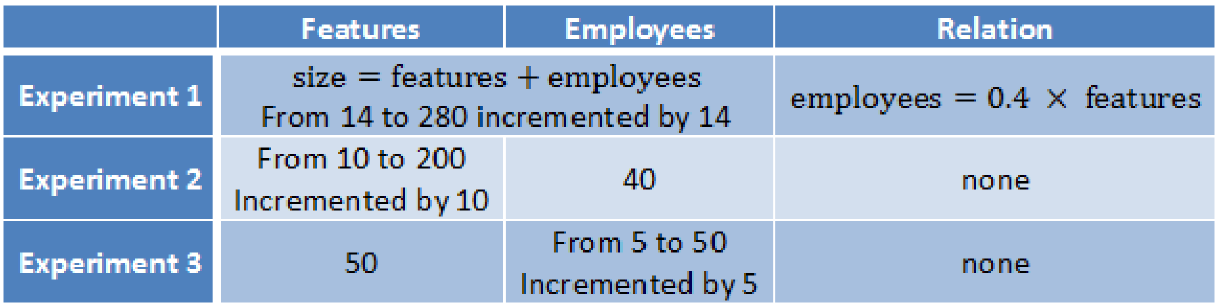
To do this, I have chosen to base my method on the objective values. Concretely, for each problem, the priority score objective value is between 0 and the *worstscore* of the problem. As 0 is the value for planning all the features and *worstscore* the value for having planned none of them, I have decided to attribute a priority quality score between 0 and 1 as $Q_p = 1 - \frac{priority\ value}{worst\ score}$.

The same method was used for the end date objective ($Q_{ed} = 1 - \frac{end\ date\ value}{iteration\ end}$) and the final quality score is the average of this two calculations.

This quality score has a value between 0 and 1, respectively the worst and the best score. This score does not have any other usefulness than to compare two results for the same problem. For instance having a score of 0.1 can seem bad but this result cannot be expressed because it can be a case in which there were a lots of features to plan and very few time in the iteration as a little part of them can be included. Its objective is only to say that an algorithm that obtain a final solution with best quality for the same problem than an other has found a better solution. Finally, I have accepted that a solution with violated constraint has a quality score to 0.

6.1.1 Filter

As some algorithms provide a list of solutions as result, I was asked to extract only one of them. In fact, there are algorithms that return the final population as result and all the solutions are not the best so I had to filter them and I used the quality indicator to do this. Moreover, there still are some solutions that are strictly equivalents in term of objectives (for instance, two employees with the same skills for who we can inverse their



	Features	Employees	Relation
Experiment 1	size = features + employees From 14 to 280 incremented by 14		employees = 0.4 × features
Experiment 2	From 10 to 200 Incremented by 10	40	none
Experiment 3	50	From 5 to 50 Incremented by 5	none

Table 6.1: Experiments

two plannings) so we decided to extract one randomly.

6.2 Experiment protocol

Inspired by some papers which compare algorithm results on the Next Release Problem considering the size but also trying to find out some relation with the number of customers or the number of features[2][11], we have decided of three types of experiments (resumed on Table 6.1):

- Comparing the different algorithms results in function of the size of the problem (considering that the size is the sum of the number of employees and the number of features) with a constant ratio of *employees by feature* extracted from the previous section,
- Comparing the results for a constant number of employees and by varying the number of features in order to see if the best algorithm is different with a different proportion of employees by feature,
- Comparing the results for a constant number of features and by varying the number of employees with the aims of determining if there is an algorithm that has better results with limited resources for instance.

To do this, each algorithm will be executed on the same data set, this is reproduced 50 times (on 50 data sets in total) and the result for each algorithm (executed with 500 evaluations of a 100-size population) is the average of his values. The Algorithm 5 illustrates this on the employees experimenting.

6.3 Results

The following results are presented using the *JFreeChart* library which provides the chart frames. I only had to give the data to present and a couple of presentation parameters to obtain the experimenting graphs.

Algorithm 5: Simplified Evaluation Algorithm

```
Data: $algorithms$: The list of algorithms to experiment
Result: $dataset$ : The set of experiment data
$numberOfEmployees \leftarrow$ INITIAL_EMPLOYEES ;
$dataset \leftarrow$ initializeSeries();
$params \leftarrow$ new GeneratorParameters(NUMBER_OF_FEATURES,
  $numberOfEmployees$);
while $numberOfEmployees \leq MAX\_EMPLOYEES$ do
    $qualityValues \leftarrow$ initializeMap($algorithms$);
    for $i \in 0 \ldots TEST\_REPRODUCTION$ do
        $data \leftarrow$ generateData($params$);
        $nrp \leftarrow$ new NextReleaseProblem($data$);
        $executor \leftarrow$ new AlgorithmExecutor($nrp$) ;
        foreach $algorithm \in algorithms$ do
            $result \leftarrow executor$.executeAlgorithm($algorithm$) ;
            $qualityValues$.get($algorithm$)[$i$] $\leftarrow result$.getQuality() ;
        end
    end
    $dataset$.updateSeries($qualityValues$);
    $numberOfEmployees$ += EMPLOYEES_INCREMENT;
    $params$.setNumberOfEmployees($numberOfEmployees$);
end
return $dataset$;
```

6.3.1 Experiment 1

As it can be seen on Figure 6.1, the algorithm which provides better solutions is MOCell and whatever the size is. NSGA-II and SPEA-II provide solutions not as good but the quality gap becomes smaller and smaller with the increase of the size. On the other hand, PESA-II is always the worst algorithm for the Next Release Problem.

On the chart, it can be observed a decreasing trend of the solution qualities. This is due to the complexity of the problem that is increasing with the size but above all to the iteration time that it is fixed to 3 weeks of 35 hours so the number of features that can be planned is still constant.

6.3.2 Experiment 2

This second experiment confirms the dominance of MOCell in most of cases but the chart (Figure 6.2) also shows that with few features, the other algorithms produces results as good nay better.

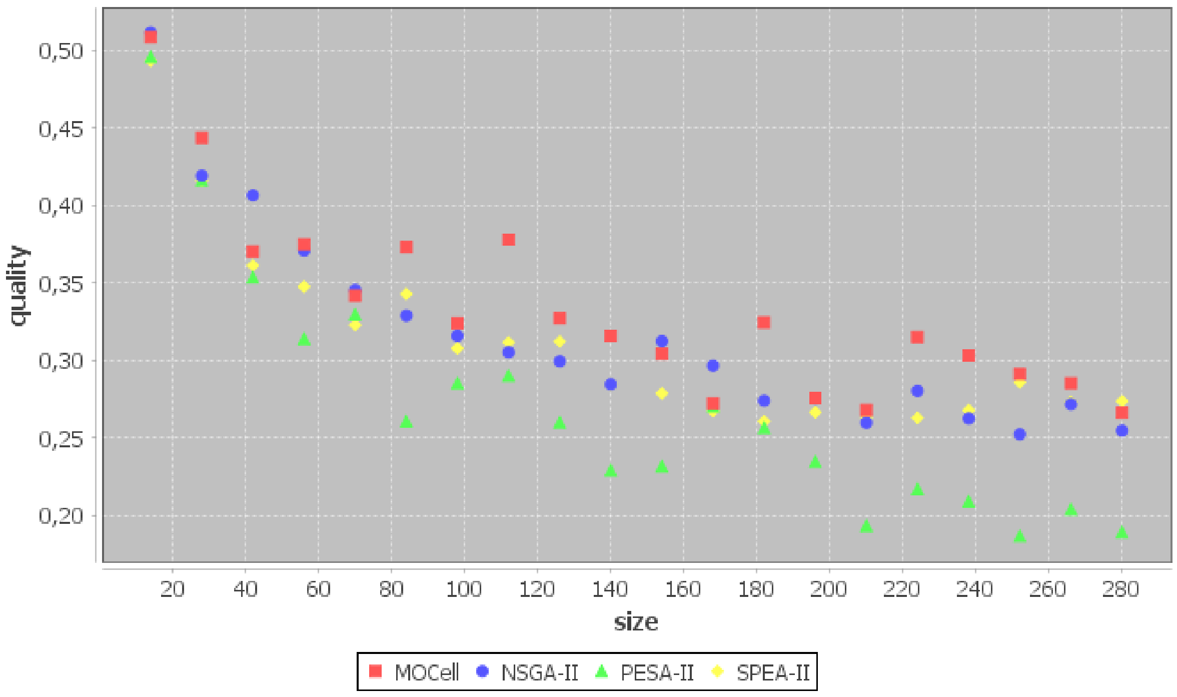


Figure 6.1: Results of Experiment 1

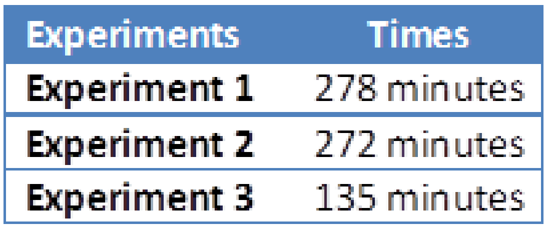
6.3.3 Experiment 3

This last experiment (results on Figure 6.3) confirms the dominance of MOCell to solve the Next Release Problem especially when the resources are limited. But when the number of employees approaches the number of features, NSGA-II and PESA-II provide solutions as good as MOCell and even exceed it when the two variables are equals.

6.4 Computing time

All these experiments were processed on a personal computer using Windows 7 SP1 64bits with an Intel® Core™i3-2350M processor (3M Cache, 2.30 GHz) with 6 Go of RAM.

The times of execution of each experiment is presented on Table 6.2. The Experiment 2 is the faster one because of less experiments done.



Experiments	Times
Experiment 1	278 minutes
Experiment 2	272 minutes
Experiment 3	135 minutes

Table 6.2: Execution times of the experiments

The average execution times of the algorithms can be found on Table 6.3. We notice

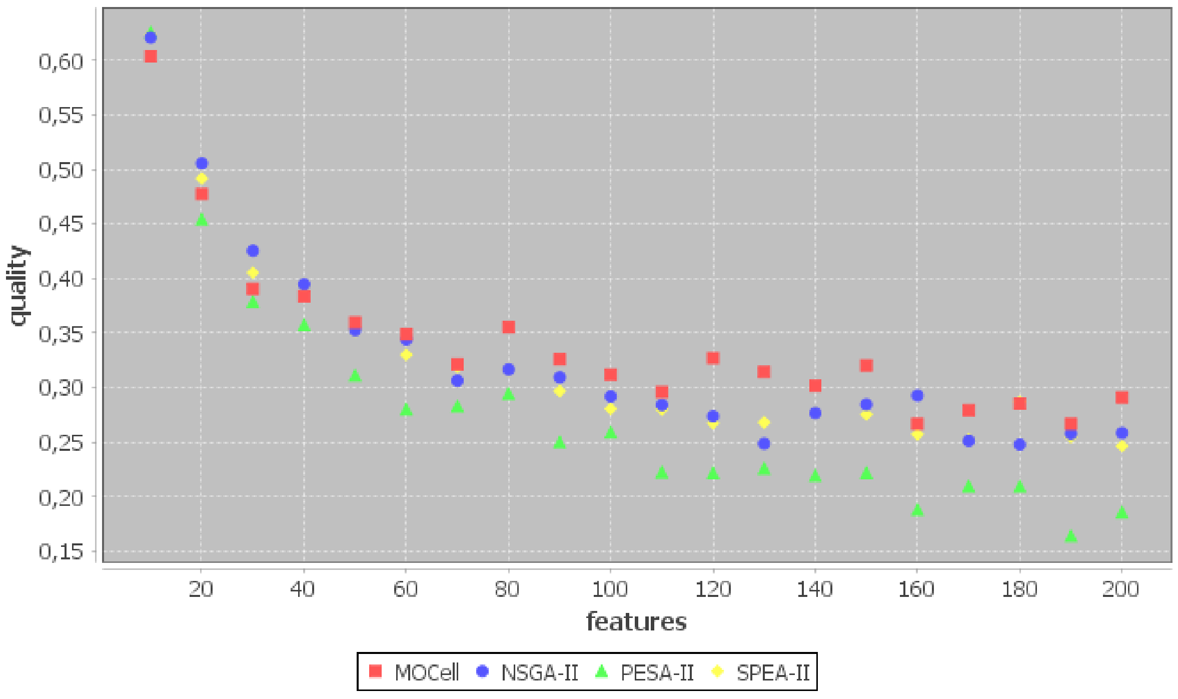
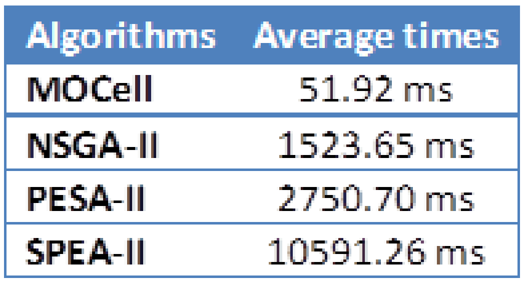


Figure 6.2: Results of Experiment 2

that the MOCEll algoritihm is much faster than all the others and that SPEA-II is the longer and that it is responsible of most of the experiment execution time.



Algorithms	Average times
MOCell	51.92 ms
NSGA-II	1523.65 ms
PESA-II	2750.70 ms
SPEA-II	10591.26 ms

Table 6.3: Execution times of the algorithms

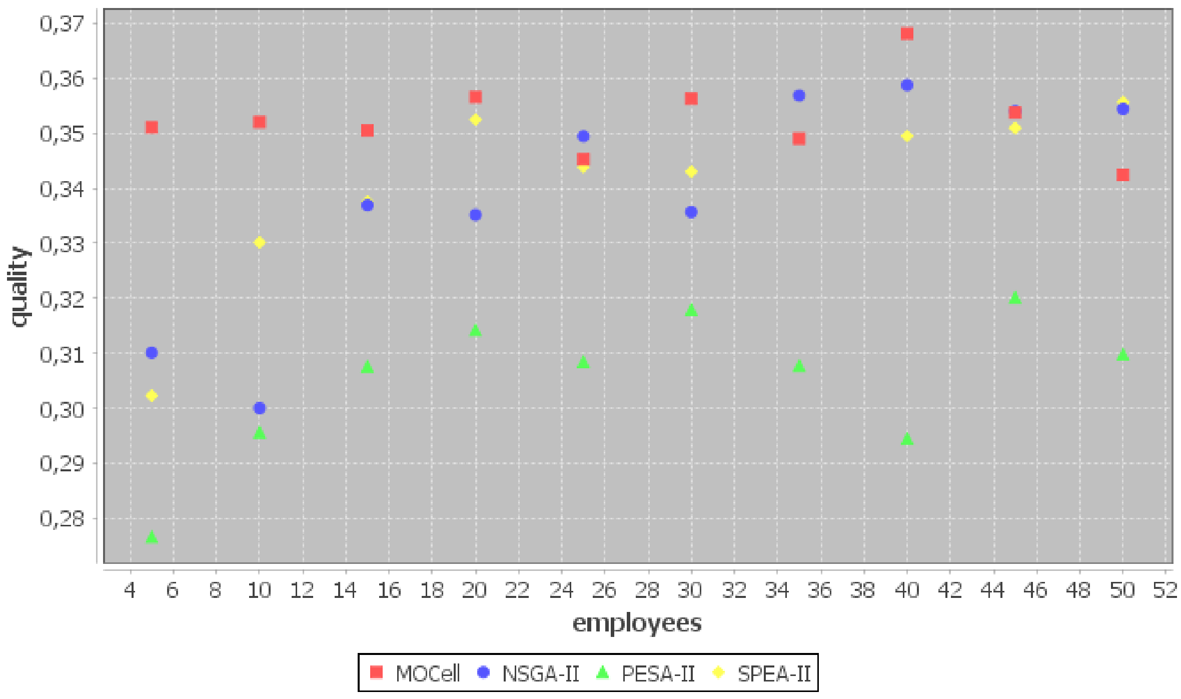


Figure 6.3: Results of Experiment 3

Evaluation

7.1 Results

This thesis has revealed that in addition to being the faster, MOCell algorithm is the one which provides better results without depending on the resources available or the size of the problem. NSGA-II and SPEA-II can also provide close solutions needing little more time. In contrast PSEA-II is not a good option for solving our version of the Next Release Problem.

7.2 Planning

The updated planning can be seen on Figure 7.1 (the initial one is on page 12). If the tasks and their order were respected, there is a large difference in the durations. Indeed, we had split the thesis into three equals parts but the first one, about learning and implementing the problem has lasted much longer. This is due to the need to learn a lot about mutli-objective resolution and to the gaps in jMetal documentation which the core is well explicated but as in our case I had to specialize quite many class behaviours, it was complicated to find information. Moreover, I had faced on optimisation problems thats final solutions were not as optimised as expected and it was complicated to debug because of using random (not reproducible), the use of several threads (complicate to trace) and the amount of data that is processed (hundreds of iterations which manipulate population of 100 solutions).

After passing this step, some corrections have been made but as I got used with the environment, the steps of executing and experimenting were faster than expected.

7.3 Personal comments

This first experience in investigation was very rich for me. Besides having gained knowledge about the Next Release Problem, meta-heuristics and genetic algorithms, it taught me how to find information using scientific publications and about investigation methodology. Before this thesis, I had a more practical approach, accustomed to work in

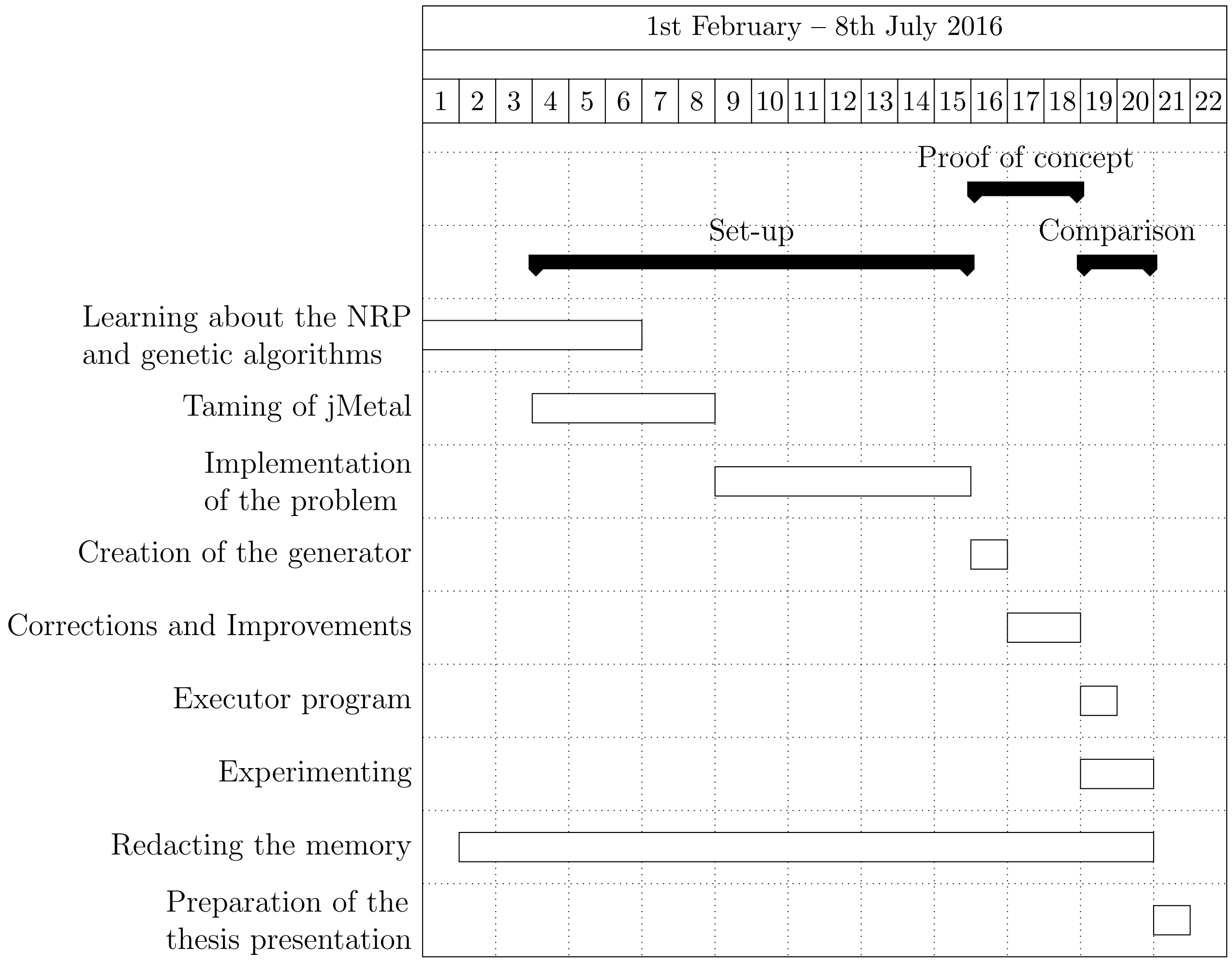


Figure 7.1: Final Gantt Diagram

internship contexts but after an adaptation time and thanks to my supervisors feedback, I could have a more holistic and scientific point of view.

In addition, using professional tools, libraries and frameworks in a computer science context has taught me about technique and allows me to produce better results. Moreover, the lack of some documentation reminds me how important it is to document what is done.

Conclusion

The objective of the thesis was to determine which genetic algorithm performs better the resolution of the Next Release Problem. The problem was adapted to be more generic and include the available resources and to be able to produce a precise planning.

Several programs were created in order to solve the problem, create relevant data set and finally execute the experiment of the thesis. Moreover, it some key values were extracted to modelise realistic problems.

An important work was done in learning and then implement the better genetic algorithm strategies to fit with the concerned problem especially on the operators and on the chosen probabilities.

The experiments figure out that the MOCell algorithm is the better genetic algorithm included in the jMetal library to solve the Next Release Problem. It is the faster one but NSGA-II and SPEA-II provide also good results in a reasonable time.

Finally, as meta-heuristic solutions are constantly evolving, it would be interesting to compare the results extracted from this thesis with other types of algorithms than the genetic ones.

References

[1] Anthony J. Bagnall, Victor J. Rayward-Smith, and Ian M Whittley. The next release problem. *Information and software technology*, 43(14):883–890, 2001.

[2] Yuanyuan Zhang, Mark Harman, and S Afshin Mansouri. The multi-objective next release problem. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1129–1137. ACM, 2007.

[3] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[4] Mais Haj-Rachid, Christelle Bloch, Wahiba Ramdane-Cherif, and Pascal Chatonnay. Différentes opérateurs évolutionnaires de permutation: sélections, croisements et mutations. http://lifc.univ-fcomte.fr/ publis/papers/pub/2010/RR2010-07.pdf, july 2010.

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[6] Antonio J Nebro, Juan J Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[7] David W Corne, Nick R Jerram, Joshua D Knowles, Martin J Oates, et al. Pesa-ii: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the genetic and evolutionary computation conference (GECCO'2001*. Citeseer, 2001.

[8] Daniel Kunkle. A summary and comparison of moea algorithms. *Northeast. Univ. Boston Mass*, 2005.

[9] Jifeng Xuan, He Jiang, Zhilei Ren, and Zhongxuan Luo. Solving the large scale next release problem with a backbone-based multilevel algorithm. *Software Engineering, IEEE Transactions on*, 38(5):1195–1212, 2012.

[10] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[11] Juan J Durillo, Yuanyuan Zhang, Enrique Alba, Mark Harman, and Antonio J Nebro. A study of the bi-objective next release problem. *Empirical Software Engineering*, 16(1):29–60, 2011.

[12] Antonio J. Nebro and Juan J. Durillo. *jMetal 5 Documentation.* University of Málaga, 2015.

Appendices

Test Cases

Here are presented the inputs and outputs of the test cases

Simplest

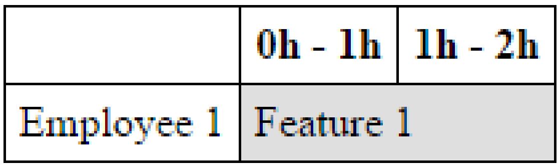
simplest.features:

```
Feature 1       2       2.0     Skill 1
```

simplest.employees:

```
Employee 1      10.0    Skill 1
```

Output:



	0h - 1h	1h - 2h
Employee 1	Feature 1	

Figure A.1: Output of the simplest test case

Simple Optimisation

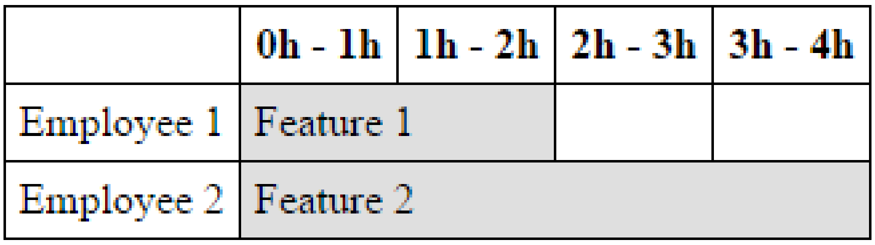
simpleoptimisation.features:

```
Feature 1       2       2.0     Skill 1
Feature 2       2       4.0     Skill 1
```

simpleoptimisation.employees:

```
Employee 1      10.0    Skill 1
Employee 2      5.0     Skill 1
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h
Employee 1	Feature 1			
Employee 2	Feature 2			

Figure A.2: Output of the simple optimisation test case

Precedence

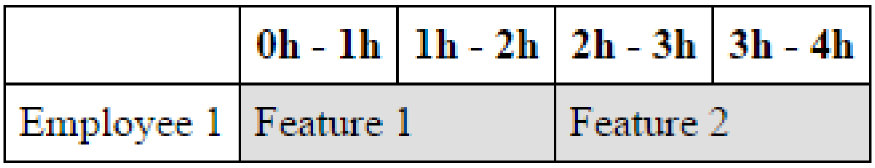
precedence.features:

```
Feature 1       2       2.0     Skill 1
Feature 2       2       2.0     Skill 1 Feature 1
```

precedence.employees:

```
Employee 1      10.0    Skill 1
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h
Employee 1	Feature 1		Feature 2	

Figure A.3: Output of the precedence test case

Precedences

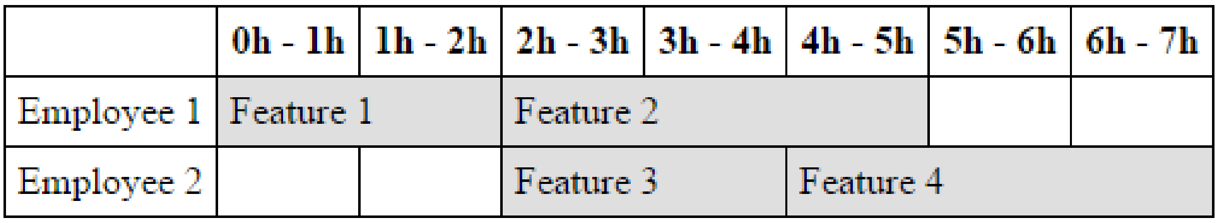
precedences.features:

```
Feature 1       2       2.0     Skill 1
Feature 2       2       3.0     Skill 1 Feature 1
Feature 3       2       2.0     Skill 1 Feature 1
Feature 4       2       3.0     Skill 1 Feature 3
```

precedences.employees:

```
Employee 1      20.0    Skill 1
Employee 2      20.0    Skill 1
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h	4h - 5h	5h - 6h	6h - 7h
Employee 1	Feature 1		Feature 2				
Employee 2			Feature 3		Feature 4		

Figure A.4: Output of the precedences test case

Skills

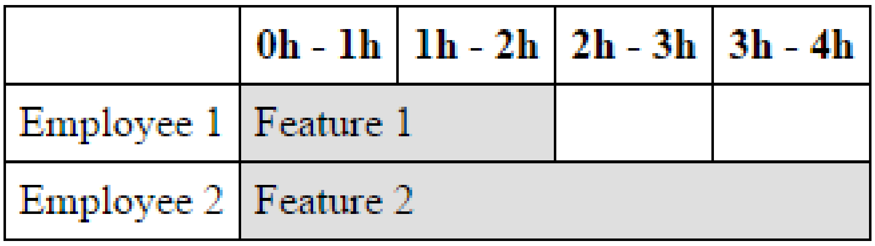
skills.features:

```
Feature 1       2       2.0     Skill 1
Feature 2       2       4.0     Skill 2
```

skills.employees:

```
Employee 1      35.0    Skill 1
Employee 2      35.0    Skill 2
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h
Employee 1	Feature 1			
Employee 2	Feature 2			

Figure A.5: Output of the skills test case

Overflow

skills.features:

```
Feature 1       2       36.0    Skill 1
```

skills.employees:

```
Employee 1      50.0    Skill 1
```

Output:

```
The solution no has planned feature.
```

Employee Overflow

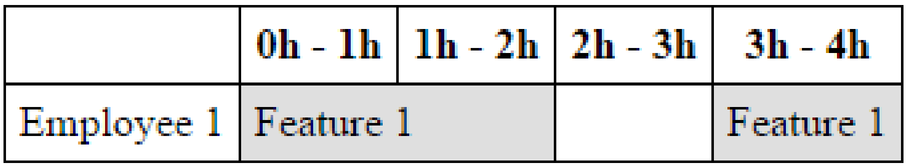
employeeoverflow.features:

```
Feature 1       2       3.0     Skill 1
```

employeeoverflow.employees:

```
Employee 1      2.0     Skill 1
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h
Employee 1	Feature 1			Feature 1

Figure A.6: Output of the employee overflow test case

Overflow Optimisation

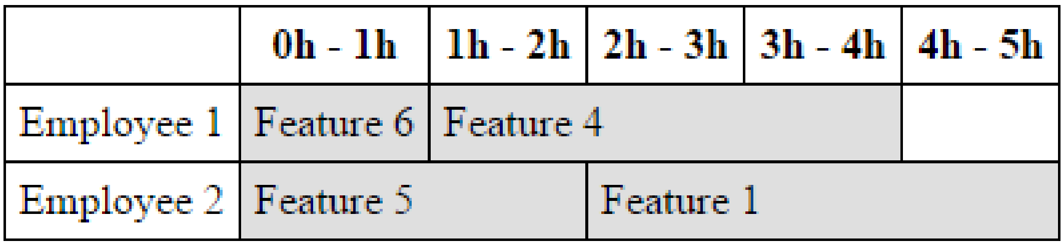
overflowoptimisation.features:

```
Feature 1       1       3.0     Skill 1
Feature 2       2       2.0     Skill 1
Feature 3       3       1.0     Skill 1
Feature 4       1       3.0     Skill 1
Feature 5       2       2.0     Skill 1
Feature 6       3       1.0     Skill 1
```

overflowoptimisation.employees:

```
Employee 1      4.0     Skill 1
Employee 2      5.0     Skill 1
```

Output:



	0h - 1h	1h - 2h	2h - 3h	3h - 4h	4h - 5h
Employee 1	Feature 6	Feature 4			
Employee 2	Feature 5		Feature 1		

Figure A.7: Output of the employee overflow test case