UiO : Department of Informatics
University of Oslo

Using Metaheuristic Algorithms to Reconfigure Context-aware Software

Magnus Røed Hestvik

Master's Thesis Spring 2017

Using Metaheuristic Algorithms to Reconfigure Context-aware Software

Magnus Røed Hestvik

16th May 2017

Abstract

This thesis demonstrates how metaheuristic algorithms can be used to find valid configurations for context-aware software. To represent families of context-aware software, the thesis describes and uses Context-dependent Feature Models (CFMs). Three different metaheuristic algorithms are introduced: Hill-climbing, Simulated Annealing and Genetic Algorithm. These three approaches are implemented in a reconfiguration engine that aims to find valid configurations on different sets of CFMs. Additionally, the paper presents a tool that randomly generates these datasets for the purpose of benchmark testing. Although randomly generated, the CFMs in the different datasets are defined by a range of predefined parameters.

The performance of the metaheuristic algorithms is evaluated based on running time and the number of successful reconfigurations. For comparison, the results are compared to the performance of a CP-solver, called HyVarRec. This thesis highlights some key differences in how the three algorithms perform and how some specific properties of CFMs may cause one algorithm to perform differently from another. The general findings in this thesis suggest that Simulated Annealing (SA) is an efficient algorithm with a high probability of succeeding. Genetic Algorithm and Hill-Climbing (HC) are in general not as successful as SA. However, HC is very efficient and is often able to reconfigure models where SA is unsuccessful. Especially on smaller CFMs, the approach of running SA and HC together performs quite well when compared to HyVarRec.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I would first like to express my gratitude to my thesis supervisors Ingrid Chieh Yu and Jacopo Mauro for their expertise and guidance over three semesters. With enthusiasm and invaluable advises, they helped to steer the thesis in the right direction, while allowing it to be my own work.

A big thanks go to Johanne Teigen, Linn Eirin Paulsen and Elin Røed for their support, and for providing useful comments on the thesis. Also, I would like to thank my family for their support and understanding throughout the process of planning and writing this thesis.

Chapter 1

Introduction

An increasing amount of tasks, we traditionally have carried out manually, are being handed over to be executed by computers. Small devices in our houses, our cars and in our pockets already take care of different tasks for us. Modern technology is able to carry out jobs like finding the best travelling route, keeping track of appointments and turning on the coffeemaker when we wake up, with little to no effort demanded by the user. However, by leaving a machine in charge of carrying out daily life tasks, you also give away some of your control. How can we assure that we get the results we want when we let automatic processes take over the execution?

The fact that our lives are becoming more and more automated involves a number of challenges, which this text will not address. The goal of this project is to investigate ways for different types of software, that performs these automated tasks, to make the right decisions more efficiently. The introduction of the "Internet of Things" to our homes and lives might be the most influential current change, regarding life automation. It allows for entirely new ways of interacting with our surroundings, even unknowingly in many situations. However, in order for this technology to be reliable in performing crucial tasks unassisted, the software should be aware of, and make decisions based on changing external factors. If the coffeemaker turns itself on, while you are away over the weekend, only because that's what it has been instructed to do every other weekend, you might discard this service altogether, and go back to making your coffee manually.

The building blocks that make up a software, and that the software relies on to make a decision, will be called features. To represent the possible composition of features in a software and how the features are dependent on each other, this thesis will use Feature Models (FMs). An FM is a logical mapping of features that a software in a software family may have. The combination of features which are selected and not selected, for a given software, is often called a product or, as in this thesis, a configuration. FMs will be the basis of representation for the problems discussed here. For a software to be able to make the right decision, it must have the correct configuration. However, here the focus is on software that may need to change their decision making when external factors change. For the

software to be able to continue to work properly when the circumstances change it needs a new configuration that is compatible with the new circumstances.

A standard Feature Model represents 2^n possible configurations, where n is the number of features. An engine with the task of reconfiguring a Feature Model is therefore met with the challenge of finding a valid configuration in an exponential search space. Furthermore, many types of software families require representation of features that can take more than two values, a demand that requires an extended Feature Model. Extended FMs are more expressive, but their search space is greatly increased compared to the simpler version [1].

There are many strategies that take on the challenge of solving problems with exponential time complexity in sufficient time. Drawing inspiration from how our brains use shortcuts in problem-solving as well as problem-solving in nature, this project explores e a set of strategies called metaheuristics. Metaheuristics differ from basic algorithms because they cannot guarantee that they will find the best solution. Furthermore, the variants of metaheuristics explored in this thesis don't produce the same result each time they execute on the same problem, which is another trait of basic algorithms. By applying randomness and underlying assumptions, metaheuristics resemble the problem-solving techniques we commonly use in everyday life. This project investigate how these strategies can efficiently help software adapt when external factors change.

Many studies have explored the field of efficient Software testing, by using Feature Models. Recently, researchers have proposed methods that involve the use of metaheuristics both for testing FMs [8], and for generating FMs through reverse engineering [7]. There are however few studies that explore Feature Models extended to represent families of context-aware software. This paper aims to explore how metaheuristics can help reconfiguring a new type of FM, called a context-dependent feature model (CFM), which represents families of context-aware software. CFMs contain a new type of constraint, called Validity Formulas, that connect the features and attributes to external factors in the context.

There are a number of different metaheuristic algorithms. This thesis will present and discuss three different ones: Hill-Climbing, Simulated Annealing, and Genetic Algorithm. They will be evaluated on sets of CFMs, and assessed to answer how they perform compared to each other. Furthermore, CFMs comes in different sizes, and are structured differently. The metaheuristics will be tested on sets of Context-dependent Feature Models with different sets and structures, to find if there are certain types of CFMs that are harder for the metaheuristics to solve than others.

A different approach has been presented for solving the problem of reconfiguring Context-dependent Feature Models. In a project called HyVar, a reconfiguration engine called HyVarRec is proposed [11]. This engine applies a Constraint Programming solver to create a configuration that is valid given the contextual information The tests in this thesis evaluates the performance of the three metaheuristics and compare their performances with the performance of HyVarRec, to investigate if

metaheuristic algorithms can be a good alternative.

In order to carry out the evaluation of the metaheuristic algorithms, the thesis will present a reconfiguration engine and a CFM dataset generator. The reconfiguration engine applies one or more of the metaheuristic algorithms to locate a valid configuration for a CFM given a specific context. The dataset generator will be used to construct collections of CFMs with a variety of sizes and structural differences, which will be used to test the performance of the metaheuristics in the reconfiguration engine.

The results in this thesis show that Simulated Annealing is a quite good algorithm for reconfiguring Context-dependent Feature Models. Simulated Annealing proves to be quite time efficient and has a high probability of success. Hill-Climbing does not succeed as often as Simulated Annealing, but is a very efficient algorithm. The approach of running both Hill-Climbing and Simulated Annealing in sequence performs at a level that is competitive to HyVarRec. This is especially the case for smaller models. Genetic Algorithm performs poorly in the tests presented here. In order to get a good understanding of GA's abilities to reconfigure CFMs, more testing is needed.

CFMs with different structural traits provide different challenges for the metaheuristic algorithms. Most interesting is the performance of SA and HC with models that are constructed with different numbers of two special types of constraints, called Cross-tree-constraints (CTCs) and Validity Formulas (VFs). The results suggest that SA is quite vulnerable to CFMs with a large number of CTCs, but it does not perform significantly worse when the number of VFs is increased. For Hill-Climbing the situation is the opposite. HC is only slightly vulnerable to a large number of CTCs but performs significantly worse when the number of VFs increase.

This thesis begins with an overview of the Context-dependent Feature Models, metaheuristics, and the CP-solver HyVarRec. Then the problem will be formally defined and the formal data structures are described. Chapter 4 describes the design of the Reconfiguration engine that was developed for this project. That includes descriptions of the designs chosen for the three metaheuristic algorithms. In Chapter 5 the approach for evaluating the metaheuristic algorithms is discussed. In that chapter the Dataset Generator is presented. A tool developed for this project to generate large sets of CFMs. Finally, the thesis displays and discusses the results from testing the performances of the metaheuristic algorithms.

Chapter 2

Background

Feature Models are models representing the variances and commonalities in families of software [11]. A product of a Feature Model is a configuration, a map over which features are selected, that satisfies all the constraints defined in the structure of the model. This project explores the challenge of efficiently finding a valid configuration of a special type of Feature Model. The type of Feature Model examined in this project are constructed to represent software that may demand reconfiguration when external factors change. To model these types of context-aware software families this project uses Context-dependent Feature Models (CFMs), a Feature Model extended to include dependencies towards environmental requirements. In this type of Feature Models a configuration is only valid if it also fulfil the requirements defined by the external factors.

First, this chapter describes Context-dependent Feature Models. Then, for the problem of finding a valid configuration for a CFM this chapter will give an overview of metaheuristic algorithms. In particular, three metaheuristic algorithms will be described: Hill-Climbing, Simulated Annealing, and Genetic Algorithm. Finally, the chapter describes the reconfiguration engine HyVarRec, which the performances of the metaheuristic algorithms will be compared to.

2.1 Context-dependent Feature Models

Feature Models (FMs) can be viewed as a representation of variances and commonalities between software products. It was first introduced by [6] in order to introduce traditional product lines to software engineering. Software product lines (SPLs) enables the possibility for reusing software, as opposed to reproducing parts when similar software is developed. An FM maps out the different features of a software and is a convenient model for numerous analysis purposes. I will here describe the basic structure of an FM, as it will be defined in this project.

An FM can be represented as a tree where the nodes represent features, as shown in the example in Figure 2.1. Each feature can either be selected or not, depending on the logical constraints in the tree. Every node, except the root, is dependent on its parent, and cannot be selected unless the parent

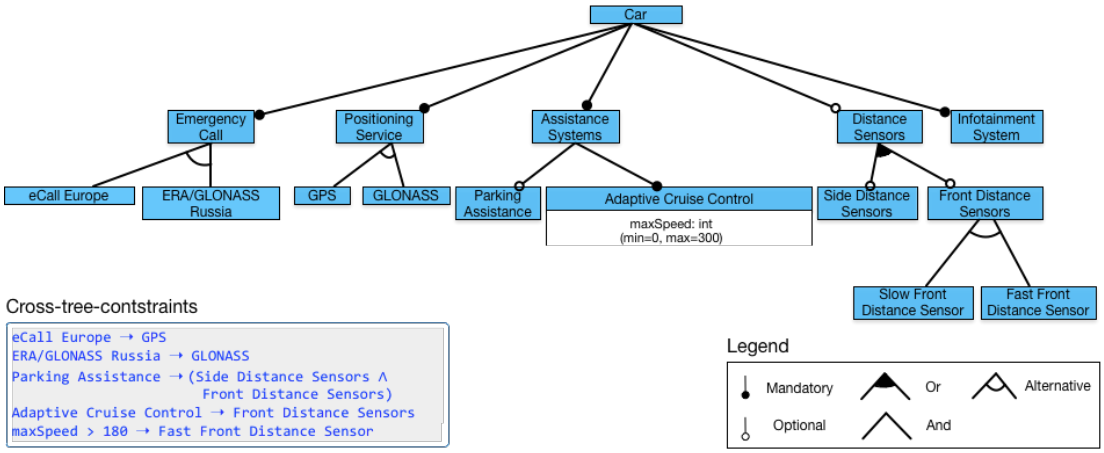


Figure 2.1: Example Feature Model representing the software in a car

This diagram, and the next, are based on the diagram in "Context Aware Reconfiguration in SPL" by Mauro et. al [11]

is selected. Furthermore, the children of a node can be connected by an optional edge or a mandatory edge. An optional relation between a parent and a child is a normal implication, where the parent must be selected if the child is selected. The mandatory edge allows for both the parent and the child to be selected, or none of them.

A parent node can also have its children grouped. The And-group is used to state that all of the nodes, contained in it, must be selected if the parent is. In an Or-group, at least one must be selected. Finally, in the Alternative-group exactly one feature must be selected. Additionally, an FM may have cross-tree-constraints (CTCs). These constraints express that a feature in one sub-tree is dependent on a feature in another sub-tree. For instance, in the example in Figure 2.1, if the feature "eCall Europe" is the selected type of "Emergency Call", then "GPS" must be selected under "Positioning Service", as stated by the first CTC.

A configuration of an FM is an assignment of boolean values to all features, indicating whether a feature is selected or not. Features may also have attributes that can represent non-binary data types like integers and strings. For instance, the feature "Adaptive Cruise Control" in the example above has the attribute "maxSpeed", which is an integer with range between 0 and 300. These attributes may also be dependent on constraints in the model. The current example has a cross-tree constraint where the feature "Fast Front Distant Sensor" is required if "maxSpeed" has value higher than 180. Accordingly, a configuration must, in addition to selecting features, assign values to the attributes in the FM. A configuration is valid if it follows all dependencies expressed by the FM.

This thesis will explore Feature Models where configurations, that might be valid in one situation, could become invalid because of changes in some external factors. Consider the software in a car, controlling features like navigation and max speed. If the car relocates to a country with different speed regulations, or the weather type changes, some

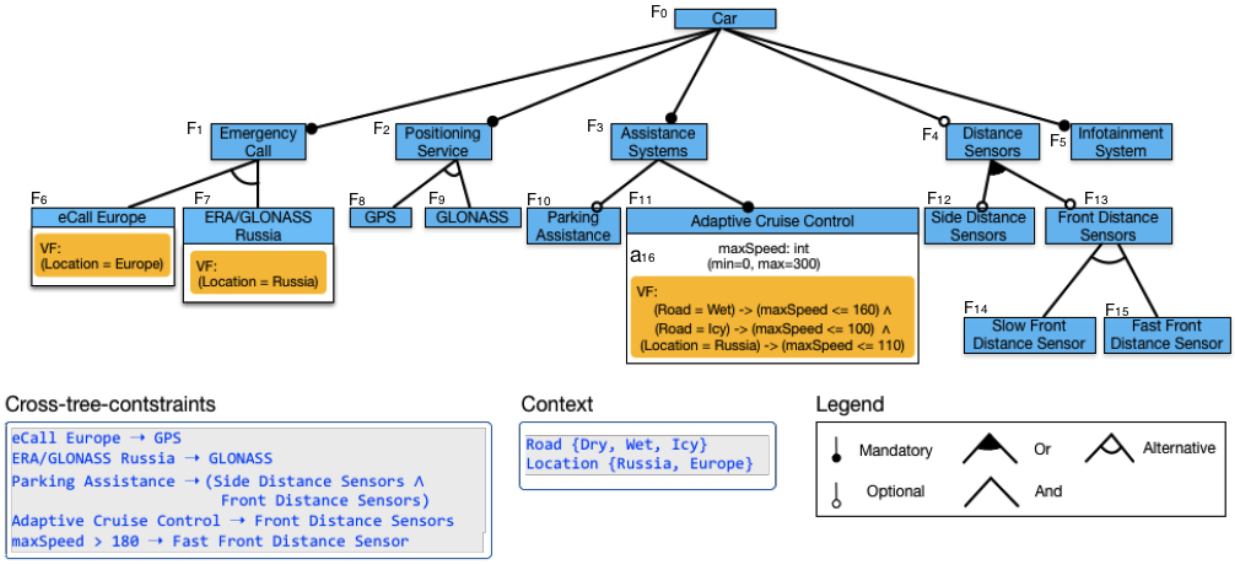


Figure 2.2: Context-dependent Feature Model diagram of the running example

configurations might be undesirable. However, these changes happen in the surroundings and are not directly represented in the FM.

This project will use an extended version of the standard Feature Model, as described in [11]. The extended Feature Model, which will be called a Context-dependent Feature Model (CFM) introduces a context, a set of variables representing external factors. For instance, the Feature Model above can be extended with a context, containing the variable location. The dependencies between the Feature Model and the context are represented by Validity Formulas (VFs).

VFs are part of the Feature Model and are similar to cross-tree constraints. However, instead of describing dependencies between sub-trees, the VFs connect a feature to a context variable. For instance, in Figure 2.2 the feature "Ecall Europe" has a VF stating that if the feature is selected then the context variable "Location" must have the value "Europe". This implies that whenever the context variable "Location" does not have the value "Europe", the feature "Ecall Europe" can not be selected by a valid configuration. Furthermore, VFs may implicate attribute values, and may be combined to affect a feature or an attribute by more than one context variable. The CFM in Figure 2.2 extends the FM in Figure 2.1 with VFs and a context. This diagram represents the context-dependent Feature Model that will be used as an example and running example throughout the thesis. With this addition, a configuration is valid unless it contradicts any of the constraints in the CFM or constraints expressed by the VFs.

2.2 Metaheuristics

In context-dependent systems the validity of a configuration is not constant. A configuration that is valid under certain circumstances will need quick replacement during runtime in order to adapt to new conditions when external factors change. As a result of that, it is crucial for a reconfiguration engine to be efficient. However, with an exponential search space, locating an optimal configuration for a new context is computational expensive. In order to achieve an efficient reconfiguration of CFMs this thesis explores strategies called metaheuristics.

Heuristics are shortcut strategies used by the brain to make quick decisions. Commonly, these strategies make use of, and emphasise, already existing knowledge instead of systematically evaluating each possible solution. Heuristics are therefore often associated with biases and can lead to flawed or non-optimal decisions. Nevertheless, we make use of them every day, because they facilitate for problems to be solved quickly, and usually lead to adequate decisions.

Metaheuristics are not based on any specific heuristic as such but utilises the general idea of taking shortcuts rather than exploring all possible alternatives in search of the best solution. In [15], a metaheuristic is defined as "a top-level strategy that guides an underlying heuristic solving a given problem". What distinguishes metaheuristics from exact algorithms is the notion that only parts of the search space need to be explored to find an adequate solution. In this context, the search space is the set of all possible solutions, good or bad, that a classic algorithm would generate and compare to one another. A candidate solution is a possible solution in the search space.

A challenge in metaheuristics is that, since we don't know the optimal solution, also called the global optima, we don't know exactly what we are looking for. Instead, a metaheuristic assumes that we know when we see it[9]. To actualise this idea, we need some kind of measurement or objective function, to evaluate and compare candidates. How this function is defined depends on the problem. For this section, when a candidate is described to be a good solution or a better solution than the previous one, it implies that an underlying evaluation procedure exists.

A very simple and basic metaheuristic is Random Search. This strategy randomly generates candidate solutions, over and over, within some time limit, and picks the best one. Random search explores parts of the search space and can be instructed to spend a fixed amount of time. Although it does not make use of any underlying heuristic or strategy, it is an interesting approach that can be combined with other, more advanced methods.

Most of the metaheuristics that will be explored in this thesis are based on local search. Where Random Search is a pure exploration approach, the local search metaheuristic applies a strategy that exploits the search space[9] based on how the candidate solutions are structured. For each candidate, we can generate a neighbourhood. The neighbourhood is a set of candidates differing slightly from the current one. The basic idea of a

local search algorithm is to iteratively locate improved candidates in local search spaces. The local search space, in an iteration, is the neighbourhood of the currently selected candidate. If there is a better candidate solution in the local search space, it replaces the currently selected candidate. When there are no better candidates to obtain, the currently best candidate is returned. A strategy that is based on this method is Hill-Climbing.

2.2.1 Hill-Climbing

Hill-Climbing (HC) is a basic local search implementation. From an initial, randomly generated candidate it iteratively replaces the current candidate with a better one, by searching among the local alternatives in the neighbourhood. This is outlined in pseudocode in Algorithm 2.1. In the algorithm, s keeps track of whatever the best candidate, at any time, is. If the score of s reaches 0, meaning it is a global optima, the algorithm returns it. On the other hand, if every neighbour of x have been assessed without replacing s as an improved candidate, the algorithm halts without a global optima. When it reaches a candidate with a neighbourhood containing no improved solutions it returns the current one.

Algorithm 2.1: Pseudocode of Hill-Climbing

Input: A candidate solution x.
Output: Solution s or *null* if unable to find a valid solution

```
1:  s ← x
2:  while score(s) > 0 do
3:      for each x' ∈ N(x) do
4:          s ← min(score(x'), score(s))
5:      end for
6:      if s = x then
7:          return null
8:      else
9:          x ← s
10:     end if
11: end while
12: return s
```

The neighbourhood of a current candidate is usually defined as a set containing all possible candidates that differ from the current candidate in one small alteration. What the alteration is, depends on the structure of the candidates. For vectors, the neighbourhood can be defined as all vectors where one value is increased or decreased by one, compared to the candidate.

There are some drawbacks with this metaheuristic algorithm. Hill-Climbing is great at finding local optima quickly, but often fail to find the globally best solution [14]. That is because it is quite vulnerable to locally optimal solutions and plateaus. As its name illustrates, the procedure is continuously climbing up towards a hill-top, by taking one small step at a

time. When it has no next step that can lead it upwards, it considers itself to be on the top. The problem is that it sees only one step at a time, in terms of improving the candidate. Figure 2.3 illustrates these challenges.

Consider the global optima as the highest hilltop. A local optima is a candidate with lower score than the global optima, but all candidates in the local optima's neighbourhood score even less, which leads the Hill-Climging algorithm to a halt. A plateau is a situation where, for a number of steps of candidates' neighbourhoods, picking the best candidate in each step leads to the same result. When it thinks it has reached the top, there might actually be a way to continue upwards to a better solution if it was willing to continue through some intermediate steps without progress. Computing more steps, in each iteration, is a possibility, but it greatly increases the complexity of the procedure.

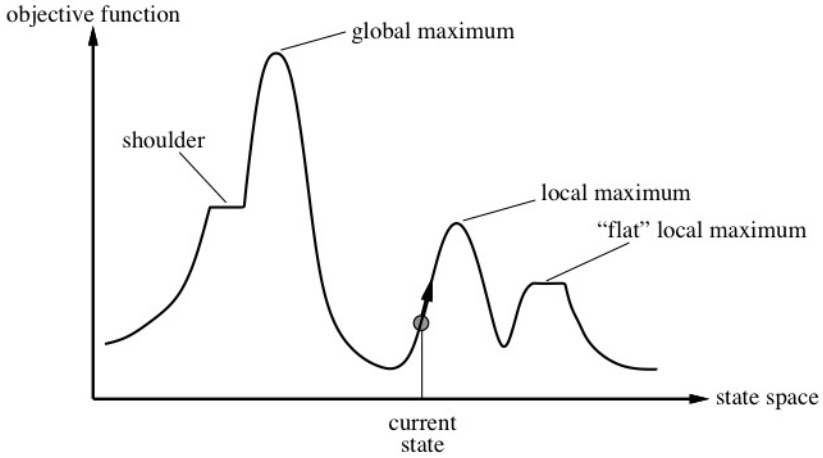


Figure 2.3: A simplified illustration of progress during Hill-Climbing

The shoulder and "flat" local maximum are plateaus. Source: westminstercollege.edu

A great advantage to only ascending is that finding a solution may take quite a few steps and makes this metaheuristic time efficient. We can, therefore, improve the solution without increasing the complexity too much, by running Hill-climbing several times with different initial candidates.

By running Hill-climbing more times, the exploration of Random Search is combined with the exploitation of Hill-Climbing. In this approach Hill-Climbing runs for a fixed number of times or, until it reaches a time limit. Randomly generating an initial candidate each time. This provides different local optimal candidates from which the algorithm chooses the best one, unless one of the executions result in a global optima, which immediately is returned.

Random Search might obtain the global optimal solution if it by chance stumbles upon it, among all the possible randomised candidates. Hill-Climbing, on the other hand, only needs to stumble upon the hill with its local best candidate also being the global one. In terms of vectors, this means that if any of the vector candidates that will lead to the global optima

is selected through Hill-Climbing steps, the algorithm is able to obtain the optimal solution. A drawback might be that there is a chance of returning to the same hill over and over, repeating previous computation steps, without getting closer to an improved solution.

2.2.2 Simulated Annealing

Simulated Annealing (SA) [4, 9, 15] is a metaheuristic that shares its fundamental structure with Hill-Climbing. However, in contrast to Hill-Climbing, Simulated Annealing applies a strategy to avoid being trapped in local optima. Instead of exclusively selecting candidates from the neighbourhoods that improve the solution, non-improving candidates has a probability of being accepted as well.

The strategy it implements is inspired by physical annealing, the process of heating some types of solid metal to their maximum temperature, before slowly and carefully cooling them down. The probability value, expressing the likelihood of choosing a given non-improving candidate, is dependent on a continuously decreasing variable, called temperature. Along with the temperature the probability decreases towards 0 for each iteration.

Algorithm 2.2: Pseudocode of Simulated Annealing

Input: A candidate solution x
Output: Solution s or *null* if unable to find a valid solution

```
1:  Select the temperature change counter k ← 0
2:  Set a temperature cooling schedule T
3:  Select initial temperature t ← T(0)
4:  while while stopping criterion is false and score(c) > 0 do
5:      for each x' in N(x) do
6:          δ = score(x) − score(x')
7:          if δ > 0 or random(0, 1) < exp(δ/T) then
8:              x ← x'
9:              break for each
10:         end if
11:     end for
12:     k ← k + 1
13:     t ← T(k)
14: end while
15: return x
```

With this technique, Simulated Annealing can easily ignore many locally optimal solutions and continue looking for better ones, by expanding its scope beyond the local neighbourhood in the search space. Sooner or later it will conclude on an optimal solution since the procedure primarily accepts improving candidates when the temperature is approaching 0. How many steps this takes depends, among other things, on how fast the temperature decreases.

Algorithm 2.2 outlines the basic structure of Simulated Annealing, based on [3] and [4], but presented as a minimisation strategy instead of

a maximisation strategy. For instance, δ is defined as the score of current candidate c minus the neighbour c', resulting in a positive number if the new result has a lower score, which is better. The acceptance probability function, here formulated as $exp(\delta/T)$ can be switched out with other functions. The function represented here is the same as in the original approach, except that it is written for a minimisation problem. The functions N and *score* are the same as in the Hill-Climbing algorithm.

2.2.3 Genetic Algorithms

Genetic Algorithms (GA)[4, 9, 15] is a set of metaheuristics following a slightly different approach then the ones described above. In these approaches, inspiration is drawn, among other things, from natural selection and gene mutation. Where a local search procedure, like Hill-Climbing, begins with one initial candidate and selects exactly one new candidate, in each iteration, GAs work with groups of candidates. This group is called a population, and each state of the population is called a generation, The initial generation is a set of randomly generated candidates.

The goal of a Genetic Algorithm is to continuously replace candidates in the population, creating new generations, while increasing the average fitness. The fitness is based on the objective function that the procedure uses to evaluate candidates. When working with GAs it is assumed that candidates are represented as vectors. Following the analogy, these vectors are the genomes of the different individuals in a generation. .

Algorithm 2.3: Pseudocode of a basic Genetic Algorithm

```
1:  popsize ← an even number for size of the population
2:  Generate initial population P, of size popsize.
3:  Let s be random solution from P
4:  repeat
5:      for each x in P do
6:          if score(x) < score(s) then
7:              x ← s
8:          end if
9:      end for
10:     Q ← ∅
11:     for i = 0 to popsize/2 do
12:         p_a ← select(P)
13:         p_b ← select(P)
14:         c_a, c_b ← crossover(p_a, p_b)
15:         Q ← Q ∪ {mutate(c_a), mutate(c_b)}
16:     end for
17:     P ← Q
18: until score(s) = 0 or termination criterion is met
```

Starting from an initial random generation, the GA begins the process

of creating a second generation by selecting parent candidates. The *select* function usually applies some sort of technique to do a random selection while making sure that candidates with high fitness has an advantage over candidates with lower fitness. This can be viewed as the natural selection step. For the sake of diversity, sometimes the algorithm may want to select a low fitness candidate to be a parent. Diversity is important in GAs to prevent the algorithm from selecting candidates with similar traits and eventually getting stuck in local optima.

After the parents have been selected comes the reproduction step. The candidates in the parent groups get paired up, and their offsprings are created through a combination of the procedures crossover and mutation. The crossover procedure slices the two parents' vectors, usually at the same place. The starting slice of one parent vector gets switched with the corresponding slice from the other parent, creating two new distinct child candidates. Variations of this procedure include slicing the vectors at more than one point, and, if the length of the vectors can vary between candidates, slicing them at different positions.

In the mutation procedure, each newly created child vector may have one or more of its values altered. The mutation probability is the likelihood of a value within a candidate being altered. An increased probability provides higher diversity, but if set too high the algorithm may risk introducing too much randomness into the procedure, undermining the general strategies.

When a new generation is created, the procedures repeats until termination. If the metaheuristic works as intended, each new generation will generally have an average fitness that is higher than the previous generation. Termination can be invoked when the average fitness hasn't increased significantly, or if the fittest candidate has not been switched out with a better one, in a certain number of steps. Termination can also be set to happen after a fixed number of iterations or within a time limit.

2.3 HyVarRec

The performances of the metaheuristics will in this project be compared with the CFM reconfiguration engine HyVarRec. HyVarRec uses a Constraint Programming-solver to create a new valid configuration for the CFM [11].

Constraint Programming is a paradigm for solving combinatorial problems [12]. The expressiveness of constraint programming makes one general purpose CP-solver able to solve a variety of problems in different domains. CP uses constraints to formulate a problem which makes it suitable for reconfiguration of CFMs, since a CFM is easily translated into a set of constraints. Constraint Satisfaction Problems are problems for which any solution that fulfils each constraint is accepted, while Constraint Optimisation Problems takes an objective function and returns a solution that maximises or minimises that function.

HyVarRec takes as input the CFM, a context, and a configuration. The

configuration in the input usually is the one that was made invalid because of changes in the context. The task is to locate the valid configuration that is most similar to the configuration given as input, which HyVarRec solves as a Constraint Optimisation Problem.

Chapter 3

Formal description of the problem

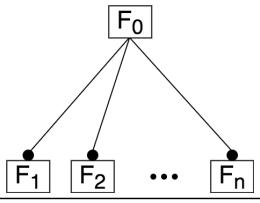
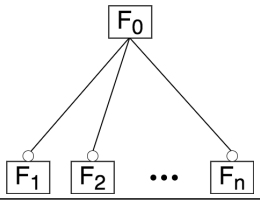
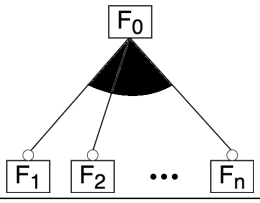
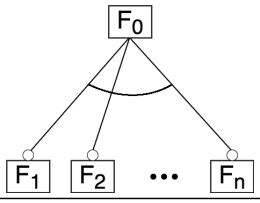
The goal of this thesis is to investigate how metaheuristic algorithms can be used to find valid configurations of context-aware software families. This chapter formalises this goal by introducing the formal data structure of a Context-dependent Feature Model and describes the problem formally.

3.1 Data structures for Context-dependent Feature Models

To represent families of context-aware software this thesis will use Context-dependent Feature Models. A Context-dependent Feature Model is an extended version of attributed Feature Models and was introduced by Mauro et al. [11]. This section describes how the data structure of Context-dependent Feature Models is defined in this project. It also defines a context, how the VF's are turned into rules and it shows how configurations are represented in the project.

The formal representation of a Context-dependent Feature Model in this thesis is a 4-tuple $CFM = \langle F, R, T, A \rangle$ where F is the set of features, R is a set of rules or dependencies, $T \subseteq F$ is a non-empty set of features that must be selected in any valid configuration, and A is the set of attributes. The reason why T is non-empty is that it must at least contain the root feature of the CFM. Features are elements that take binary values. An attribute is an element that is owned by a feature. It has a defined range and can take any value in that range.

The features in F and the attributes in A are named with an incrementing number between 0 and $n = |F| + |A| - 1$. The order of which they are named is not relevant to the rest of the CFM structure, but is a reference to the position in the configuration arrays. The configuration is described in a later section. Any feature and attribute must therefore have a unique number. Additionally, each number between 0 and n must reference either a feature or an attribute. In the CFMs described in this thesis it is assumed that the features are numbered following a level-ordering of the CFM tree

Constraint Type	FM diagram constraint	Rules
Mandatory		$F_0 \rightarrow F_1$ [a] $\dots$ $F_0 \rightarrow F_n$ $F_1 \vee F_2 \vee \dots \vee F_n \rightarrow F_0$
Optional		$F_1 \vee F_2 \vee \dots \vee F_n \rightarrow F_0$
Or		$F_0 \rightarrow F_1 \vee F_2 \vee \dots \vee F_n$ $F_1 \vee F_2 \vee \dots \vee F_n \rightarrow F_0$
Alternative		$F_0 \rightarrow (F_1' + F_2' + \dots + F_n' = 1)$ [b] $F_1 \vee F_2 \vee \dots \vee F_n \rightarrow F_0$
Inclusive CTC	F_0 INCLUDES F_1	$F_0 \rightarrow F_1$
Exclusive CTC	F_0 EXCLUDES F_1	$F_0 \rightarrow \neg F_1$

[a] In the actual data structure F_i is notated as $F_i = 1$ and $\neg F_i$ as $F_i = 0$
[b] F_i' here refers to the numeric value, which is 1 if F_i is true/selected and 0 if F_i is false/not selected.

Table 3.1: Mapping between the constraint types in a Feature Model diagram and the rules in the set R in the data structure

structure. Furthermore, it is assumed that all attributes are given a higher number than the features.

The rules in R are logical formulas. The atomic elements of these logical formulas are comparison expressions that include a feature, an attribute or a context variable, and in one special case a number of features. It is assumed that all features used in the rules in R are represented in F and that all attributes used in the rules in R are represented in A.

The construction of rules made up by features will be explained first. Consider therefore a standard Feature Model without attributes or Validity Formulas. Features in a Feature Model are related to each other by four logically different constraint types: Mandatory, Optional, Or and Alternative. The rules in R are systematically constructed from the constraints in the Feature Model as shown by the mapping in Table 3.1.

Another type of constraint that is common in Feature Models, but not explicitly represented in the mapping above, is the And-constraint. And-constraints are relations between a parent feature F_i and a set of child features $\{F_j, \ldots F_n\}$. It is combined with mandatory and optional relations, so that if F_i has a mandatory relation to one of its children F_k and an optional relation to another child F_l, then if F_i is selected, F_k must be selected while the other child F_l may or may not be selected. However, if any of the child features are selected, then F_i must also be selected. One way to represent this rule is, given a parent feature F_i, to make one set of children with a mandatory relation from F_i and one set of children with an optional relation from F_i. This is the approach that was chosen for the data structure in this thesis.

Let $\{F_a, F_b, F_c\}$ be a set of children of the feature F_i, and let F_i have mandatory relations to all of them. To represent the first direction of the implication, that the parent implies the children, the model could have used one single compact rule, $F_i \rightarrow F_a \wedge F_b \wedge F_c$. Instead, this rule is split into several simpler rules, $F_i \rightarrow F_a, F_i \rightarrow F_b$, and $F_i \rightarrow F_c$, based on an assumption that this structure provides better guidance to the metaheuristic algorithms. The metaheuristics evaluates configurations based on the number of rules that are not satisfied, and it usually makes small changes to the current configuration in each iteration.

In the example above, the associated rule, or rules, only evaluates to true if all the features are selected, or if none of the features are selected. The assumption is that the single compact rule may fail to guide the metaheuristics towards some optimal solutions, because it doesn't reward the selection of a mandatory child when the parent is selected, unless all other children already are selected. This assumption was also supported in initial tests of all three metaheuristics tested in this thesis. The mapping of mandatory rules splits, therefore, And-groups of Mandatory children into one rule for each child, instead of combining them into one rule.

The other direction of the mandatory rule is equal to the Optional-constraints and to the children-to-parent dependency of Or- and Alternative-constraints. These rules are all represented with a single compact rule involving all the children in that group. This choice was made on the assumption that fewer rules demand less resources for the score function.

Initial testing shows that, compared to the alternative of splitting the rule into simpler rules, this approach have a slightly positive impact on efficiency and, additionally, a positive impact on the success rate of Genetic Algorithm.

Or- and Alternative-constraints are, just like And-constraints, relations between a parent and a group of children, but in contrast to And-relations they restrict the range of features that can be selected at the same time. The less restrictive of the two is Or-relations, which requires at least one feature to be selected if the parent is selected. In an Alternative-group, one and exactly one feature must be selected, if the parent is selected. Additionally, there are two types of Cross-tree-constraints to consider. These are simple implications between two features and can, therefore, be included in the data structure with little change.

Context-dependent Feature Model extends the standard Feature Models by implementing Validity Formulas (VFs). Validity Formulas (VFs) constitutes a bridge between the model and the context. Similar to constraints, VFs are relations between a feature and, in this case, an assignment to a contextual variable. The VFs are added to the set of rules R.

In this paper, the context is formally defined as a function $f_c : C \rightarrow V$, where C is the set of context variables and V is the collective range of possible values. For instance, let $C = \{c_k, c_l\}$, where the context variable c_k have the range $\{0, 1, 2\}$ and the context variable c_l have the range $\{2, 3, 4\}$. Then a valid context could be $\{\langle c_k, 1\rangle, \langle c_l, 4\rangle\}$.

A VF may restrict the selection of some features for certain contexts. For instance, assume that a Context-dependent Feature Model CFM has the feature F_i and a VF: $F_i \rightarrow c_k = 2$. In this CFM, the feature F_i can only be selected if the context maps c_k to the value 2. If a configuration has F_i selected, and the current context maps c_k to the value 1, then the configuration is not valid.

VFs can also be added to a CFM to restrict the possible values of an attribute. For instance, assume that CFM has an attribute a_j which is owned by F_i and that the range of a_j is $[0, 20]$. If CFM has the VF: $F_i \rightarrow (c_k = 2 \rightarrow a_j < 10)$ and the context variable c_k maps to the value 2, then a valid configuration must either assign the attribute a_j to a value lower than 10 or the configuration doesn't select F_i.

3.2 Configuration

A configuration of a Context-dependent Feature Model is an array of assignments, where each position refers to a specific feature or attribute. A position in a configuration that refers to a feature can be either 1 or 0, where 1 means the feature is selected and 0 means it is not selected. An underlying assumption when constructing a configuration is that the features are numbered in an unbroken chain from 0 and increasing. For instance, for a CFM with 20 features, the features are numbered from 0 to 19, and a configuration is an array of length 20.

Another assumption is that any attributes in the CFM are numbered

incrementally, continuing from the highest number of a feature. This implies that any Context-dependent Feature Model with n number of features and m number of attributes uses the following structure. All features are named F_i, where $0 \leq i < n$ and all attributes are named a_j, where $n \leq j < n + m$. A configuration of such a CFM is of size $n + m - 1$ and has binary values in the positions referenced by i. The value in position j is limited by the range of a_j.

Consider an CFM with four features, $F = \{F_0, F_1, F_2, F_3\}$, and two attributes, $A = \{a_4, a_5\}$, where the range of a_4 is $[0, 5]$ and the range of a_5 is $[2, 4]$. One possible configuration of this CFM is the array $\langle 1, 1, 0, 0, 5, 2 \rangle$. This array represents a configuration for the CFM that is structurally correct because it meets the restrictions set by the feature and attribute ranges. However, it is not necessarily valid. To evaluate if a configuration is valid it must also fulfil all the rules in R. In the rest of this paper, all configurations are assumed to be structurally correct.

3.3 Running example

This section will demonstrate how the data structure represents a Context-dependent Feature Model using the CFM represented by the diagram in Figure 2.2. First the set of context variables will be defined.

The diagram states the context variables as *Road* and *Location*, and their ranges of context values. For the formal representation, it is not important exactly what the context variables and their ranges represent. Similar to naming the features and attributes, the context variables are named c_k, where k is a positive number. However, since context variables are not represented in a configuration, the value of k does not need to follow the incrementing indexing used to name the features and attributes. The range of, for instance, "Road" could be translated into a numeric interval $[0, 2]$, where the numbers index the actual value, and these are being kept track of outside of the reconfiguration engine. For readability, the data structure of this example contains the text representation of the context range values.

The resulting representation of the context variables is the set $C = \{c_0, c_1\}$, where c_0 has range $V_0 = \{"Dry", "Wet", "Icy"\}$, and c_1 has range $V_1 = \{"Russia", "Europe"\}$. The context is a function $cf : C \rightarrow V_0 \cup V_1$ that upholds the restriction set by the context variable ranges, so that $cf(c_0) \in V_0$ and $cf(c_1) \in V_1$.

Let $CFM_a = \langle F, T, A, R \rangle$ be the data structure that represents the Context-dependent Feature Model in Figure 2.2 The CFM contains 16 features and one attribute. The features are named in the diagram as F_0 to F_{15}, according to the assumption that the features are named in an unbroken chain starting on 0 and contained by F. Then, the root feature is located, which in this case is F_0 and added to T. The attribute is named a_{16}, it is marked as owned by F_{11}, given as range the interval $[0, 300]$, and put in A. The last step is to translate the constraints into rules and add them to R. The resulting data structure is presented in Figure 3.1

$$
\begin{array}{lcll}
F & = & \{ & F_0, F_1, F_2, \ldots, F_{14}, F_{15}\}, \\
T & = & \{ & F_0\} \\
A & = & \{ & a_{16}\}, \\
R & = & \{ & r_1 : F_0 \rightarrow F_1 \\
 & & & r_2 : F_0 \rightarrow F_2 \\
 & & & r_3 : F_0 \rightarrow F_3 \\
 & & & r_4 : F_0 \rightarrow F_5 \\
 & & & r_5 : F_1 \vee F_2 \vee F_3 \vee F_5 \rightarrow F_0 \\
 & & & r_6 : F_4 \rightarrow F_0 \\
 & & & r_7 : F_1 \rightarrow (F'_6 + F'_7 = 1) \\
 & & & r_8 : F_6 \vee F_7 \rightarrow F_1 \\
 & & & r_9 : F_2 \rightarrow (F'_8 + F'_9 = 1) \\
 & & & r_{10} : F_8 \vee F_9 \rightarrow F_2 \\
 & & & r_{11} : F_3 \rightarrow F_{11} \\
 & & & r_{12} : F_{11} \rightarrow F_3 \\
 & & & r_{13} : F_{10} \rightarrow F_3 \\
 & & & r_{14} : F_4 \rightarrow F_{12} \vee F_{13} \\
 & & & r_{15} : F_{12} \vee F_{13} \rightarrow F_4 \\
 & & & r_{16} : F_{13} \rightarrow (F'_{14} + F'_{15} = 1) \\
 & & & r_{17} : F_{14} \vee F_{15} \rightarrow F_{13} \\
 & & & r_{18} : F_6 \rightarrow F_8 \\
 & & & r_{19} : F_7 \rightarrow F_9 \\
 & & & r_{20} : F_{10} \rightarrow F_{12} \\
 & & & r_{21} : F_{10} \rightarrow F_{13} \\
 & & & r_{22} : F_{11} \rightarrow F_{13} \\
 & & & r_{23} : F_{11} \rightarrow ((a_{16} > 180) \rightarrow F_{15}) \\
 & & & r_{24} : F_6 \rightarrow (c_1 = \text{"}Europe\text{"}) \\
 & & & r_{25} : F_7 \rightarrow (c_1 = \text{"}Russia\text{"}) \\
 & & & r_{26} : F_{11} \rightarrow ((c_0 = \text{"}Wet\text{"}) \rightarrow (a_{16} \leq 160)) \\
 & & & r_{27} : F_{11} \rightarrow ((c_0 = \text{"}Icy\text{"}) \rightarrow (a_{16} \leq 100)) \\
 & & & r_{28} : F_{11} \rightarrow ((c_1 = \text{"}Russia\text{"}) \rightarrow (a_{16} \leq 110))\}
\end{array}
$$

Figure 3.1: CFM_a. Data structure for the Context-dependent Feature Model running example

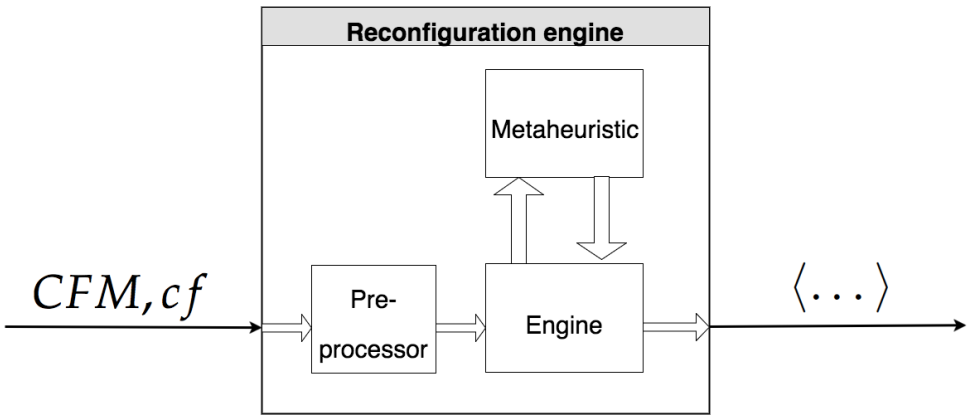


Figure 3.2: Basic overview of the reconfiguration engine with input and output

3.4 Problem description

Given a Context-dependent Feature Model and a context, the problem is to find a valid configuration. The Context-dependent Feature Model $CFM = \langle F, R, T, A \rangle$ and a context function $cf : C \rightarrow V$ are given as input to a reconfiguration engine as outlined in Figure 3.2. Since the reconfiguration engine relies on metaheuristic algorithms it is not guaranteed that it will locate a valid configuration. If the reconfiguration engine is successful it returns a vector, representing a valid configuration for CFM given the context cf. If the engine is not successful it returns the configuration it found that was the closest to an optimal solution and lets the user know that the result is not optimal. The next chapter shows how the reconfiguration engine was designed and implemented.

Chapter 4

Design and implementation of the reconfiguration engine

As a part of the project, a reconfiguration engine was developed in order to implement and test the metaheuristic algorithms. This chapter describes the design of the three different metaheuristic algorithms used in the reconfiguration engine and shows how the reconfiguration engine was constructed. The reconfiguration engine is given as input a Context-dependent Feature Model and a context function that assigns values to a finite set of context variables. Its task is to output a configuration for the CFM that is valid given the context. For any given input, the engine is constructed to run the input on one or more of the metaheuristic algorithms: Hill-climbing, Simulated Annealing, and Genetic Algorithm. If any of the metaheuristics succeed in finding a valid configuration the engine outputs the configuration, otherwise it tells the user that it was unsuccessful. The full source code of the reconfiguration engine can be found at github.com/magnurh/CFMmetaheuristicReconfig_thesisProject.

First, the chapter defines a neighbourhood which is an essential part of Hill-Climbing and Simulated Annealing's strategy in exploring local search spaces. Additionally, the score function will be defined. This function is essential to all three metaheuristics, as it enables them to evaluate configurations and to decide on their next step. Then, it will show how the input is pre-processed in order to limit the search space before running the metaheuristics. The chapter will then describe the three algorithms and explain what design choices were made when implementing them. Finally, the architecture of the engine is presented.

4.1 Defining neighbourhood and configuration score

Hill-climbing and Simulated Annealing share some of the same principles. One of these principles is that they both continually search for better results by looking at candidate solutions that differ slightly from the current candidate solution. A candidate solution is a vector representing a configuration for the Context-dependent Feature Model. If the candidate is evaluated as valid for the CFM the reconfiguration engine can output it

as a result. Otherwise, the metaheuristic algorithms will each of them in their own way, use the candidate to locate better candidate solutions.

For Hill-climbing and Simulated Annealing, the step of finding better candidate solutions involves generating and searching through the neighbourhood of the current candidate solution. The neighbourhood of a candidate solution can be defined in different ways, and these variations may impact the performance, both in term of time efficiency and results. For any of the three algorithms to be able to compare candidate solutions, so that it can choose the best of them, they need an evaluation metric. This section will describe how this project defines a neighbourhood of a candidate solution. It also describes how a candidate solution is evaluated and how the reconfiguration knows that a candidate solution represents a valid configuration for the CFM.

In metaheuristics like Hill-climbing and Simulated Annealing, a neighbourhood is a subset of the search space where they search for the next candidate solution. The neighbourhood is based on the current candidate solution and is therefore different in each iteration. This thesis will use the notation $N(\vec{c})$ for the neighbourhood of a candidate solution vector $\vec{c}$.

A common approach is to define the neighbourhood as a set of candidates that differs as little as possible from the current candidate without being exactly the same. In this thesis, the neighbourhood of a candidate solution vector $\vec{c}$ is defined as a set of all vectors that are different from $\vec{c}$ at exactly one position. Furthermore, the value at that position is either increased or decreased by one if the position refers to an attribute in the CFM, or, if the position refers to a feature, the value is flipped.

Consider the following example for a small Context-dependent Feature Model CFM with only three features, $F = \{F_0, F_1, F_2\}$ and two attributes $A = \{a_3, a_4\}$. The features all have, by definition, the same binary domain $\{0, 1\}$ where 0 indicates that the feature is not selected in that configuration and 1 means that it is selected. An attribute, however, can take different domain ranges, but for simplicity, both a_3 and a_4 will have as domain the range $\{0, 1, 2\}$. Note that neither the relations between the different features, the relations between attributes and features, nor the context has any impact on the vectors in the neighbourhood. One thing that does impact the neighbourhood is the set T in CFM, which contains all the features that must be selected. Let F_0 be the root of CFM. Then $F_0 \in T$. In this example this is the only feature that will always be selected. Now, let $\vec{c} = \langle 1, 1, 0, 1, 2 \rangle$ be a candidate solution vector. Table 4.1 shows what $N(\vec{c})$ looks like. The value in each neighbour that differs from the original candidate $\vec{c}$ is highlighted in red.

The neighbourhood of $\vec{c}$ contains five configurations. First, notice that the value on position 0 in $N(\vec{c})$ is not different in any of the configurations in $N(\vec{c})$. The configuration $\langle 0, 1, 0, 1, 2 \rangle$ is not represented in $N(\vec{c})$, even though it has a distance of 1 from $\vec{c}$, because the position 0 refers to a feature F_0 that must be selected in all valid configurations. A configuration where a feature in T is not selected is of no value to the algorithm and is therefore ignored by all neighbourhoods, no matter what configuration it is generated from. In this example, T contains only one feature, but

Position:		0	1	2	3	4
$\vec{c}$:		$\langle$ 1	1	0	1	2 $\rangle$
	$\vec{x}_0 =$	$\langle$ 1	0	0	1	2 $\rangle$
	$\vec{x}_1 =$	$\langle$ 1	1	1	1	2 $\rangle$
$N(\vec{c})$:	$\vec{x}_2 =$	$\langle$ 1	1	0	0	2 $\rangle$
	$\vec{x}_3 =$	$\langle$ 1	1	0	2	2 $\rangle$
	$\vec{x}_4 =$	$\langle$ 1	1	0	1	1 $\rangle$

Table 4.1: Example of a neighbourhood for a candidate vector $\vec{c}$ where the Context-dependent Feature Model has three features and two attributes

depending on the pre-processing of the CFM, it can be larger, which in turn results in smaller neighbourhoods.

The next two values, in position 1 and 2, also refers to features. Each position referring to a feature that is not in T results in one configuration in the neighbourhood where the original value is flipped. The configurations x_0 and x_1 shows this in practice.

The two last positions in the configurations, position 3 and 4, refer to the attributes a_3 and a_4. Unless the corresponding value in $\vec{c}$ already is the maximum or minimum of the attribute range, an attribute position results in two new candidate configurations in the neighbourhood. One configuration with the value increased by one and one with the value decreased by one. The configurations $\vec{x}_2$ and $\vec{x}_3$ are both results from a change in the attribute position 3. The final position only results in one configuration. Since $\vec{c}$ in position 4 has the value 2, and 2 is an upper bound of the attribute a_4, it cannot increase any further. The only resulting configuration from this position is, therefore, $\vec{x}_4$ with the value in position 4 decreased from 2 to 1.

The purpose of the neighbourhood is for the metaheuristic to search locally for a better candidate configuration. Based on the score of the different configurations in $N(\vec{c})$, the metaheuristic may select one and start over in a new iteration. Alternatively, a configuration in the neighbourhood may evaluate to be valid for the CFM. Next, the paper will present how a configuration is evaluated.

In order to determine whether a candidate solution is a valid configuration or not it is necessary to have an evaluation metric. Here, a candidate is evaluated by a score function. Determining the score of a configuration is essential to all of the metaheuristics for different purposes. It is used to guide them towards an optimal solution, to indicate that an optimal solution is found, and it is needed to decide if the algorithm is stuck and should terminate. The score is a function that takes a candidate configuration and a Context-dependent Feature Model and returns the number of rules in the CFM that are not satisfied by the configuration. A configuration is valid if it scores 0.

The basic process of the score function, as it was designed for the configuration engine in this project, is simply to go through every rule of the CFM, count the number of rules that are not satisfied, and return that

number. For each rule, the score function calls an evaluation function. This function takes a rule along with the values that the configuration assigns to the features and attributes of the rule. It outputs true if the rule is satisfied and false if it is not.

Since the score function is called many times throughout an execution of a metaheuristic algorithm, certain steps were made when designing it to increase its efficiency. In the implementation, a new score function is constructed based on a Context-dependent Feature Model and a context. It can only evaluate a configuration for the CFM given that context. This enables the score function to store and reuse all intermediate results, and to store the score of a candidate configuration so that it is not calculated more than once. When the evaluation function is called, it will look up previous results to see if the rule's logical formula has already been evaluated with the set of assignments given as input. This means that all logical formulas are evaluated only once per set of assignments. A drawback of this implementation is that the score function can not reuse the intermediate results when a new context or a new CFM is introduced.

Since a neighbour of a current candidate configuration only differs slightly from the current one, the score function only needs to evaluate some of the rules in R. That is, the rules involving the single feature or attribute where the neighbour configuration differs from the current candidate. All the other rules will evaluate to the same results, since the assignments are unchanged. For Hill-Climbing and Simulated Annealing, which uses the neighbourhood described here, this has great benefits in terms of time efficiency.

In general, the number of rules in the CFM impacts the efficiency of the score function greatly. Fortunately, not all of the rules are strictly needed and can be removed without loss of expressiveness in the CFM. Reducing the number of rules is one of the tasks that the reconfiguration engine handles before it runs the metaheuristics. The next section describes how the reconfiguration engine pre-processes a Context-dependent Feature Model.

4.2 Pre-processing the input

The first step in the reconfiguration engine is to pre-process the input. This step involves looking for some of the features that must be selected in all valid configurations and add them to T. The number of features added to T will, in turn, limit the size of the neighbourhood because all configuration in the search space must select these features. Since some features now are set to always be true, some of the rules in R become obsolete as they can not be falsified by any configuration in a neighbourhood. These rules are removed from R by the pre-processing procedure. Finally, the procedure does some work on the attributes in A, to limit their impact on the size of the search space.

The pre-processing procedure locates features that must be selected, by building a queue from T. For each feature F_i in the queue, the procedure

adds it to T unless it is already there and searches for rules where the selection of F_i implies the selection of another feature F_j. For each such rule, the implied feature is added to the queue, unless it is in T since all features in T either already have been or are in the queue. Additionally, if it is possible to remove the rule from R without affecting clauses containing other features, attributes or context variables, the procedure removes it. When all rules have been traversed, F_i is removed from the queue and another feature is selected from the queue until the queue is empty.

Let Q be a queue, built from T in CFM_a. The procedure starts by picking F_0 from Q since this is the only element contained in the queue at this point. The set T already contains F_0 and is not added to it. There are four rules where F_0 is the antecedent of an implication. Those rules are r_1, r_2, r_3, and r_5, whose consequents are single features. The four features implicated by F_0 because of these rules, F_1, F_2, F_3, and F_5 are all added to Q. The four rules are then removed from R since their consequents always will be true making these rules always true. F_0 is then removed from Q and a new feature is selected.

Assume that F_1 is the next feature to be selected. The rules r_5 and r_7 are the only rules where F_1 appears as the antecedent of an implication. In the rule, r_5 F_1 appear in a disjunction consisting of several features with F_0 as the consequent. Since it is a disjunction it is not relevant what the other features are. If F_1 is true the whole clause is true, although, in this situation, the other features are already added to T, so it is just by coincidence that F_1 is the feature to encounter the rule in the pre-processing procedure. Since the consequent F_0 already exist in T it is not added to Q again, but the rule r_5 is removed from R.

The other rule, r_7 has a more complex consequent where either F_6 is true or F_7 is true, which means that no specific feature is directly implied by F_1. Since no features are explicitly implied in this rule, no features are added to Q, and because a configuration can make the rule false it is not removed from R. A similar case happens for the feature F_2 when it is selected from Q while F_5 is no longer a part of any rule in R, so when that one is selected, no effect is taken.

The fourth of the features, F_3 results in a similar situation as its siblings in Q. Unlike the other features in Q, there is a rule where F_3 implies a new feature. In rule r_{11} the feature F_{11} is a logical consequence of F_3, and is therefor added to Q and the rule r_{11} is removed from R. The feature F_{13} in turn follows from F_{11} in rule r_{22} which represents a Cross-Tree Constraint, and is also added to Q while r_{22} is removed as a rule. Notice that the three rules at the end (r_{26}, r_{27} and r_{28}), all representing validity formulas, also have F_{11} as an antecedent. It would be possible to simplify these rules, by removing F_{11} and the first implication connective. However, the procedure here is only concerned with rules that can be removed.

Now, F_{13} is the only feature in Q. As previously, it is added to T and the procedure searches for rules where it occurs on the left side of an implication. It finds one rule, r_{15}, due to the same reason as r_5 was selected when F_1 was selected before. The rule states that if F_{13} or F_{12} is selected, F_4 must be selected, thus F_4 is added to Q and r_{15} is removed from R. Finally,

F_4 is selected and Q is left empty. One rule, r_6 has F_4 as an antecedent, but it has F_0 as the consequence. Since F_0 already is in T, nothing is added to Q. The rule r_6 is removed from R and the procedure of populating T is finished.

Notice that $F_4 \in T$, although it is an optional child of the root feature F_0. F_4 was added as a result of a chain of dependencies going through several different rules or constraints. This demonstrates how constraints, and perhaps in particular Cross-Tree Constraints can create complexity in a CFM. Assume that F_3 was an optional child to F_0. The selection of F_3 by a configuration during the execution of a metaheuristic algorithm would set in motion implications that ultimately demands that F_4 is selected. If F_4 was connected to the context with a VF it might not be valid to select, which would have as a consequence that F_3 could not be selected. So a seemingly innocent change may, in fact, be illegal, due to a VF that exists on a feature somewhere else in the CFM.

$$
\begin{array}{lcll}
F & = & \{ & F_0, F_1, F_2, \ldots, F_{14}, F_{15}\}, \\
T & = & \{ & F_0, F_1, F_2, F_3, F_5, F_{11}, F_{13}, F_4\} \\
A & = & \{ & a_{16}\}, \\
R & = & \{ & r_7 : F_1 \rightarrow (F'_6 + F'_7 = 1) \\
 & & & r_9 : F_2 \rightarrow (F'_8 + F'_9 = 1) \\
 & & & r_{16} : F_{13} \rightarrow (F'_{14} + F'_{15} = 1) \\
 & & & r_{18} : F_6 \rightarrow F_8 \\
 & & & r_{19} : F_7 \rightarrow F_9 \\
 & & & r_{20} : F_{10} \rightarrow F_{12} \\
 & & & r_{23} : F_{11} \rightarrow ((a_{16} > 180) \rightarrow F_{15}) \\
 & & & r_{24} : F_6 \rightarrow (c_1 = \text{"}Europe\text{"}) \\
 & & & r_{25} : F_7 \rightarrow (c_1 = \text{"}Russia\text{"}) \\
 & & & r_{26} : F_{11} \rightarrow ((c_0 = \text{"}Wet\text{"}) \rightarrow (a_{16} \leq 160)) \\
 & & & r_{27} : F_{11} \rightarrow ((c_0 = \text{"}Icy\text{"}) \rightarrow (a_{16} \leq 100)) \\
 & & & r_{28} : F_{11} \rightarrow ((c_1 = \text{"}Russia\text{"}) \rightarrow (a_{16} \leq 110))\}
\end{array}
$$

Figure 4.1: CFM_b. Result of pre-processing CFM_a

Now that T is filled with features that must be selected, some additional rules are no longer of any use. Before finishing the process, the rules are again traversed. A rule is removed from R if it is an implication so that a feature $F_i \in T$ appears as the consequent or is part of a disjunction that appears as the consequent. The result of this is that the rules $r_8, r_{10}, r_{13}, r_{14}, r_{17}$, and r_{21} are all removed from R. The resulting data structure is represented in Figure 4.1.

An attribute may have a range that expands from 0 to a high number, but only a few rules that reference it. Consider a CFM with an attribute a_k that has the range $[0, n]$, for a large number n and only one rule in R that references a_k. Assume, for instance, that the rule is $F_l \rightarrow (a_k \leq 1)$ and that all valid configurations require F_l to be true. The rule can easily be satisfied by setting a_k to 1. However, if the current configuration assigns the value n to a_k, for Hill-climbing and Simulated Annealing, it will take

near n iteration with no improvement on the configuration score to satisfy the rule. This is an infeasible situation. Luckily the search space can be limited. The pre-processing procedure searches through the rules in R to look for references to attributes in A, and creates intervals of what the next value in their ranges are. For instance, in the example above, an interval $\langle 0, 1, n \rangle$ would have been added to the attribute a_x. The cardinality of a_x's range, with respect to the search space, is reduced from n to 3. Now, when the reconfiguration engine generates a neighbourhood of a configuration where a_x has the value n a configuration is included where a_x is 1.

CFM_a has one attribute a_{16} which is referenced by four rules: r_{22}, r_{25}, r_{26}, and r_{27}. The range of a_{16} is as previously defined $[0, 300]$. Based on the range and the four rules the pre-processing procedure creates the interval $\langle 0, 100, 110, 160, 180, 300 \rangle$ and adds it to a_{16} to indicate what the next value in its range is. For instance, a configuration where a_{16} has the value 100 will generate a neighbourhood containing one configuration where a_{16} is 0 and another where a_{16} is 110. As the neighbourhood generation is designed, the rest of the configurations will keep the current assignment of a_{16}.

4.3 Metaheuristic designs

Metaheuristics can be designed in a variety of ways and may be used for a number of different problems and domains. In deciding on a design for each of the algorithm, several tests were made to investigate what works best for reconfiguring CFMs. The designs presented in this section draws inspiration from different implementations and general techniques. This section describes the designs of the three metaheuristics by explaining the different parts of the algorithms, and by displaying examples of how they work when reconfiguring CFMs.

4.3.1 Design and implementation of Hill-climbing

There are two main changes in the algorithm design shown in Algorithm 4.1 compared to the basic version outlined in Algorithm 2.1. In the design, the algorithm takes a number p that tells the algorithm how long it is allowed to search for better solutions when it has reached a plateau. Allowing to stay longer on a plateau lets the algorithm spend a little more time in order to be a little more likely to locate a global optimal solution. The other difference is that instead of evaluating every configuration in a neighbourhood in search of an improvement, it goes through each neighbour in random order. When doing this, it selects the first improved configuration that it finds. This change increases the efficiency because the number of evaluation in each iteration is decreased, Additionally, It turns out that it also improves the algorithms ability to locate global optimal solutions. Both of these statements were supported by initial tests of the algorithm.

The algorithm starts off with its plateau-counter p' at 0 and stores the currently best configuration, which initially is x, in s. Then it iterates

Algorithm 4.1: The design of Hill-climbing in this thesis

Input: A candidate solution x, a number p of how long the algorithm is allowed to stay on a plateau.
Output: A locally or globally optimal solution s.

```
1:  p' ← 0
2:  s ← x
3:  while score(s) > 0 and p' < p do
4:      Plateau ← {}
5:      for each x' in random(N(x)) do
6:          if score(x') < score(x) then
7:              s ← x'
8:              break for each
9:          else if score(x') = score(x) then
10:             add x' to Plateau
11:         end if
12:     end for
13:     if score(s) = score(x) then
14:         if Plateau = {} then
15:             break while
16:         else
17:             x ← selectRandom(Plateau)
18:             p' = p' + 1
19:         end if
20:     else
21:         p' ← 0
22:         x ← s
23:     end if
24: end while
25: return s
```

towards an optimal solution as long as the score of s is not 0 and the plateau counter has not reached the upper limit p. The iteration has two parts. In the lines, $5 - 12$ the neighbourhood is being searched through and in $13 - 23$ the algorithm checks if an improvement has been made.

For each random neighbour x' of x, their scores are compared. If the neighbour x' has a better score, x' is assigned to be the currently best solution s and the neighbourhood iteration immediately terminates. Otherwise, if their score is identical x', is stored in a set of plateau candidates. This set contains configurations that may be picked for the next iteration if the algorithm is stuck in a plateau. If the score of x' is worse than the score of x, no action is taken.

In the next part, if no improvement has been made, s and x will have identical scores. This situation is caused either by the algorithm being on a plateau or that it is stuck in a local optimal. If the *Plateau* set is empty, it means that no configuration of an equal score to x' was found, and there are no valid paths to take. The algorithm breaks the while loop, and the locally

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	score	(diff)
$x =$	⟨ 1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	160 ⟩	4	
$N(x)$:																			
$x'_1 =$	⟨ 1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	160 ⟩	4	(0)
$x'_2 =$	⟨ 1	1	1	1	1	1	1	0	1	0	0	1	1	1	1	1	160 ⟩	1	(-3)
$N(x'_2)$:																			
$x''_1 =$	⟨ 1	1	1	1	1	1	0	0	1	0	0	1	1	1	1	1	160 ⟩	2	(+1)
$x''_2 =$	⟨ 1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	160 ⟩	2	(+1)
$x''_3 =$	⟨ 1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	160 ⟩	3	(+2)
$x''_4 =$	⟨ 1	1	1	1	1	1	1	0	1	0	0	1	1	1	1	0	160 ⟩	0	(-1)

Table 4.2: Visited configurations during an example execution of Hill-climbing on CFM_b

optimal solution, which currently resides in s, is returned. Otherwise, a plateau candidate is selected to be a new x, and the plateau counter is increased by one. However, if an improvement was made, x is set to be s, so that the next iteration can start from the currently best solution. The plateau counter is reset to 0

The *random* function in line 5 makes sure that x' is selected in a random order, rather than in lexicographic order. This is to prevent bias towards changes in features close to the root, which in this implementation are represented at the beginning of a configuration array. Furthermore, attributes are represented at the end of the array, so this function gives them the same emphasis as the features have. The *selectRandom* function simply selects an arbitrary configuration from the *Plateau* set. Since the configurations were added to *Plateau* in a randomised order, selecting at random doesn't provide any specific purpose. However, it is included here to show how random selection of plateau candidates could be done if the algorithm was modified to iterate through neighbourhoods in a non-random order.

Let CFM_b from Figure 4.1 be the CFM and $f_c = \{\langle c_0, "Wet"\rangle, \langle c_1, "Europe"\rangle\}$ the context. This section will go through an example execution of Hill-climbing for finding a valid configuration. For this execution, let p be 0, meaning that the algorithm is not allowed to proceed if it is stuck on a plateau. Assume as input an initial configuration x as the following:

Positions:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$x =$	⟨ 1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	160 ⟩

The score of x is 4, because four of the rules in R are not satisfied. The rules r_7 and r_{16} are not satisfied, because they represent Alternative groups where exactly one of the features in the group must be selected. In both of these cases x assigns 1 to all features in the group. The rule r_{19} is not satisfied because F_7 is selected while F_9 is not, and r_{25} is unsatisfied because

F_7 is selected but $f_c(c_1) \neq$ *"Russia"*. An execution of Hill-climbing on CFM_b with x as the initial candidate is outlined, by the configuration it visits, in Table 4.2.

Let x'_1 be the first neighbour of x to be visited by the algorithm so that x'_1 differs from x in that it has F_{10} selected. The feature F_{10} is represented in the rule r_{20}, but because F_{12} is selected by the configuration, the score stays the same. Therefore, the Hill-climbing algorithm adds x'_1 to the *Plateau* set and retrieves a new neighbour. The new neighbour x'_2 differs from x in that it does not have F_7 selected. This improves the score because three of the rules previously unsatisfied becomes satisfied. In rule r_7, F_7 is a part of an Alternative group, consisting of F_6 and F_7. Since F_6 is selected, the selection of F_7 in x'_2 makes the Alternative clause, and consequently the rule, satisfied. In two other rules not satisfied by x, r_{19} and r_{25}, the feature F_7 is the antecedent of the implication. By not selecting F_7 these rules are satisfied as well. The resulting score of x'_2 is 1, which causes the Hill-Climbing to stop the search through the rest of the neighbourhood of x. Since an improvement was made, the *Plateau* set is emptied and a new iteration starts with the neighbourhood of x'_2 as the local search space.

In the second iteration, x''_1 is the first neighbour to be visited. The difference from x'_2 is that in x''_1 F_6 is not selected. Since F_7 was changed from selected to not selected in the first iteration, not selecting F_6 will make r_7 unsatisfied. The change does not improve by satisfying other rules. In fact, the only rule not satisfied at this point is r_{16} in which F_{16} is not represented, so the score is not improved. The same is the case for the next two configurations to be visited by the iteration, as shown in the table. However, the fourth neighbour x''_4 changes the value for F_{15} to not selected, which satisfies r_{16} without making any other rules unsatisfied. The configuration x''_4 has score 0, so the algorithm terminates with a valid result.

The result returned by this example of a Hill-climbing algorithm execution can be used as a valid configuration for the CFM in Figure 2.2 with a context where $Road =$ *"Wet"* and $Location =$ *"Europe"*. In the diagram, this configuration selects among others "eCall Europe", "GPS", and "Adaptive Cruise Control". Additionally, "Distant Sensors" and all of its descendants, except from "Fast Front Distance Sensor" are selected. The $maxSpeed$ is set to be 160.

In the example above the algorithm did not encounter any plateaus. With the same Context-dependant Feature Model CFM_b and the same context f_c as in the example above, imagine a case where at some point during the Hill-climbing procedure the algorithm selects the candidate configuration $c = \langle 1,1,1,1,1,1,1,0,0,1,1,1,1,1,0,1,300 \rangle$. This candidate has all, but the features F_7, F_8, and F_{14} selected and it assigns a_{16} to be 300. The score of c is 2, because r_{18} and r_{26} are not satisfied.

Two of the candidates in the neighbourhood $N(c)$ will satisfy r_{18}. One of them is the candidate that differs from c in that F_6 is no longer selected. While this satisfies r_{18} it breaks the rule r_7, since neither F_6 nor F_7 is selected. The other neighbour is the candidate where, instead of removing the selection of F_6, it selects F_8. However, this breaks the rule r_9, since both

of the features in an Alternative group, F_8 and F_9 are selected.

The other rule that is not satisfied by c is r_{26}. This implicates that, since F_{11} is selected and since $f_c(c_1) = "Wet"$,a_16 must be lower than or equal to 160. Since $F_{11} \in T$ there are no neighbours in $N(c)$ that chooses not to have it selected. Another neighbour candidate will set a_{16} to 180, which is a closer step to a solution, but it does not provide an improved score. Any other neighbours in $N(c)$ alter features that are not involved in these two rules and will consequently not improve the score, although they might not worsen it either. Consequently, this configuration leads to a plateau.

In Algorithm 4.1 the basic Hill-climbing algorithm is modified to allow for a certain amount of iterations after encountering a plateau. This addition does impact the overall success of the algorithm,, as it allows the algorithm to search a little longer. However, it is still limited by the plateau cap p and it does not solve the problem of local optima. In general, depending on the level of p, Hill-climbing is fairly strict in terms of how many iterations it allows. Let the initial, randomly generated configuration have the score s, which is maximum the size of R, and the plateau cap be the number q. If the Hill-climbing procedure fails, it will do so within $s(q-1)+1$ number of iterations, worst case. The worst case is that for each score between s and 1 it needs $q-1$ iterations to improve, except from in the last iteration when the score is 1, it stays on a plateau for q iterations, before terminating.

Usually, the situation is much better, since the result may improve by more than one score point when it improves. Furthermore, it is not likely to fall into a plateau lasting $(q-1)$ iterations each time the score changes. Since the number of iterations usually is manageable, depending on the size of R and the plateau cap p, it might often be a good strategy to repeat Hill-climbing with random starting configurations. When a cap r is set defining the number of restarts allowed, the Hill-climbing algorithm will be executed r times, unless a valid configuration is returned. In this thesis, Hill-climbing was measured with different levels set for p and r.

The next section presents the design of Simulated Annealing, which is very similar to this Hill-climbing design, but it applies a different strategy in order to avoid plateaus and local optima.

4.3.2 Design and implementation of Simulated Annealing

Just like Hill-climbing, Simulated Annealing takes as input a candidate configuration, and gives as output an optimal solution if successful, or the best candidate configuration that was found, if unsuccessful. Additionally, Simulated Annealing requires a number K that defines the maximum number of iterations the algorithm is allowed to make, and an initial temperature t.

The design of Simulated Annealing, as it is used in this thesis is outlined in pseudocode in Algorithm 4.2 and is based on [3, 4]. The two parameters, K and t represent two of the adjustments made in this design compared to the basic version of the algorithm, that was previously outlined in Algorithm 2.2. The iteration limit K defines a stopping criterion. If the

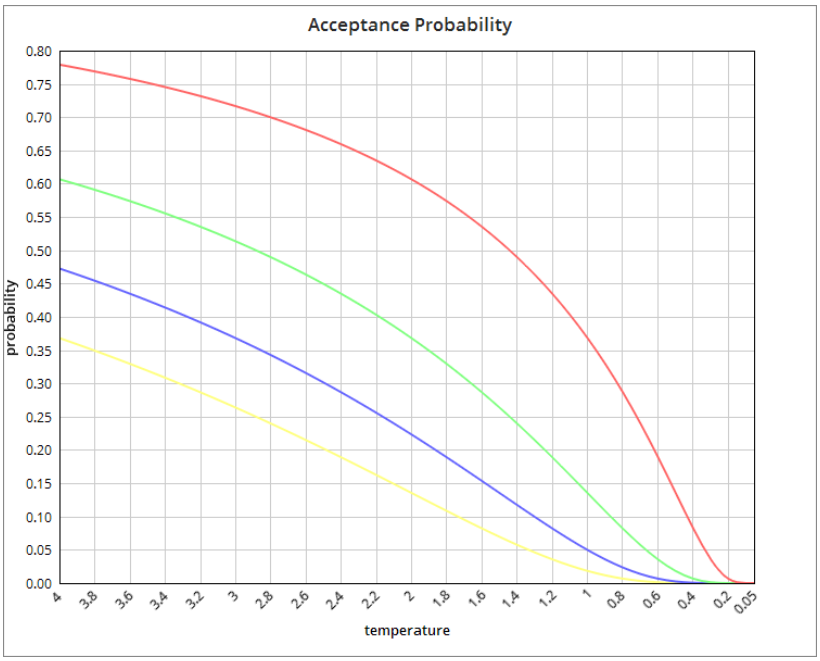


Figure 4.2: Graph of the acceptance probability function in Simulated Annealing with a linear reduction of t

Red line shows the acceptance probability when $\delta = -1$, green line when $\delta = -2$, blue line when $\delta = -3$, and yellow line when $\delta = -4$

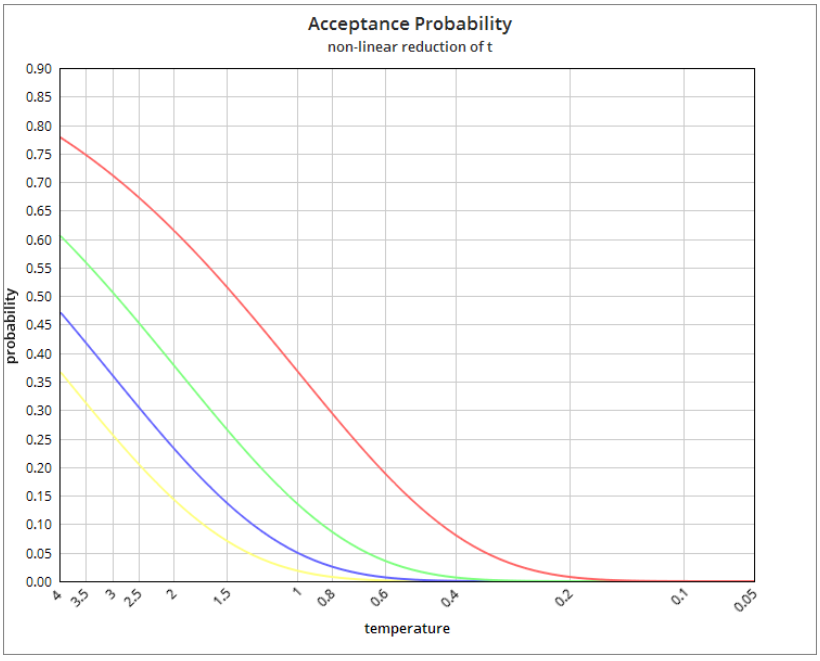


Figure 4.3: Graph of the acceptance probability function with the actual reduction of t selected for this design of SA

Red line shows the acceptance probability when $\delta = -1$, green line when $\delta = -2$, blue line when $\delta = -3$, and yellow line when $\delta = -4$

Algorithm 4.2: Pseudocode of Simulated Annealing

Input: A candidate solution x, a number K of maximum iterations, and an initial temperature t.
Output: Solution s or *null* if unable to find a valid solution

```
1: k ← 0
2: coolingRate ← t/K
3: s ← x
4: while score(s) > 0 and k ≤ K do
5:     for each x′ in random(N(x)) do
6:         δ = score(x) − score(x′)
7:         α = acceptanceProbability(δ, t)
8:         if α > random(0, 1) then
9:             x ← x′
10:            break for each
11:        end if
12:    end for
13:    if score(x) < score(s) then
14:        s ← x
15:    end if
16:    t ← t × (1 − coolingRate)
17: end while
18: return s
```

algorithm has successfully selected K number of candidate configurations without locating a global optimal solution, the currently best candidate is returned. The parameter K is also used to define the cooling schedule.

In designing the algorithm, several tests were made on CFMs to find a suitable cooling schedule. In this design t is reduced by the value *coolingRate*. The *coolingRate* is set initially as the difference between the initial temperature and K.

For each neighbour, x' of a current candidate x, x' has a probability of being selected called the acceptance probability.. The acceptance probability is calculated based on the score of x' relative to the score of x and the current temperature t. The acceptance probability is the same as in the basic version, and as suggested by [3]:

$$acceptanceProbability(\delta, t) = \begin{cases} 1.0, & \text{if } \delta > 0 \\ e^{\delta/t} & \text{otherwise} \end{cases}$$

In this function, δ is the difference between x and x'. If the score is improved by x', the difference is positive. In such cases, the *acceptanceProbability* function will return 1.0. Note that if the scores of x' and x are equal the probability will also be 1.0, since $e^0 = 1.0$. The graph in Figure 4.2 shows how the acceptance probability develops for a temperature that decreases linearly. In Figure 4.2 the same result is shown, but here the development takes into account the cooling schedule selected for the design of Simulated Annealing in this thesis. In both figures the

development is shown for cases where the difference δ is 1 through 4.

Because Simulated Annealing allows for worse candidates to be selected, this design keeps track of the best configuration found, so far. Each time the algorithm replaces current candidate with one from its neighbourhood the algorithm also compares it with the optimal configuration seen so far. If the result is improved the new candidate is stored by the algorithm in the variable s. Whichever candidate is stored in s when the algorithm terminates is the one to be restored. Furthermore, if the configuration stored in s is a global optimal at the start of an iteration, the algorithm immediately terminates.

A possible situation in the algorithm design presented here is that no candidates in the neighbourhood is selected. This could for instance occur in the unlikely, but possible event that all neighbours provide a worse score than the score of the current candidate. For this to happen, it is also necessary that the randomly generated threshold gives a higher value than the acceptance probability. Because of that, this event is more likely during later iterations when the acceptance probability is generally low for all negative values of δ, A situation like this could lead to several iterations without replacing the current candidate. To prevent that, the algorithm used in the project stores the neighbour with the lowest score, as long as no neighbour is accepted. If, at the end of an iteration, no neighbour has been selected to replace the current candidate, the locally best neighbour will take the place.

Table 4.3 shows the configurations visited in an execution of Simulated Annealing on CFM_b with $f_c = \{\langle c_0, "Wet"\rangle, \langle c_1, "Europe"\rangle\}$. In this execution, the initial temperature t was set to 2, and K was set to 12. This example uses a linear reduction of t. The initial candidate configuration x_0 is shown at the top.

The execution starts with a candidate with score 3. The rules that are not satisfied are r_7, r_{23}, and r_{26}. There is, in fact, only one neighbour of x_0, that will improve the score. That is the neighbour that changes the value of a_{16} to 180, and by that satisfy r_{23}. Any of the neighbours that make any adjustments related to features that will make a false rule satisfied will at the same time set another rule unsatisfied. Choosing the candidate that reduces the value of a_{16} would be the strategy of the Hill-Climbing algorithm, however, Simulated Annealing is a bit less predictable.

As the table shows, the first neighbour encountered by the algorithm is x_1, which does not improve the score. Since a zero difference gives a perfect acceptance probability, it must be selected. The next line shows the first configuration visited in $N(x_1)$. This configuration, x_2, has a score that is worse than the score of $x1$ by as much as 2. However, since the temperature is still high, it gets as acceptance probability 0.3. In this example execution, this was enough for x_2 to be selected.

The first configuration encountered in $N(x_2)$ also have a score that is worse by 2. Now, with a slightly cooler temperature, the acceptance probability is 0.26. This time, it is not enough to be selected. The next configuration in the neighbourhood differs only by one and is accepted with probability 0.51, which, once again, was not enough. The last

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	score	δ	α
$x_0 =$	⟨ 1	1	1	1	1	1	0	0	0	1	1	1	1	1	1	0	300 ⟩	3		
$N(x_0)$:																				
$x_1 =$	⟨ 1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	300 ⟩	3	0	1.0
$N(x_1)$:																				
$x_2 =$	⟨ 1	1	1	1	1	1	0	1	0	0	1	1	1	1	1	0	300 ⟩	5	-2	0.3
$N(x_2)$:																				
$x_3 =$	⟨ 1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	0	300 ⟩	7	-2	0.26
$x_4 =$	⟨ 1	1	1	1	1	1	0	1	0	0	1	1	1	1	0	0	300 ⟩	6	-1	0.51
$x_5 =$	⟨ 1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0	300 ⟩	4	1	1.0
$N(x_5)$:																				
$x_6 =$	⟨ 1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	300 ⟩	4	0	1.0
$N(x_6)$:																				
$x_7 =$	⟨ 1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	300 ⟩	5	-1	0.42
$N(x_7)$:																				
$x_8 =$	⟨ 1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	300 ⟩	3	2	1.0
$N(x_8)$:																				
$x_9 =$	⟨ 1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	300 ⟩	1	2	1.0
$N(x_9)$:																				
$x_{10} =$	⟨ 1	1	1	1	1	1	1	0	1	0	0	1	1	1	0	1	300 ⟩	1	0	1.0
$N(x_{10})$:																				
$x_{11} =$	⟨ 1	1	1	1	1	1	1	0	1	0	0	1	1	1	0	1	180 ⟩	1	0	1.0
$N(x_{11})$:																				
$x_{12} =$	⟨ 1	1	1	1	1	1	1	0	1	1	0	1	1	1	0	1	180 ⟩	2	-1	0.04
$x_{13} =$	⟨ 1	1	1	1	1	1	1	0	1	0	0	1	1	1	0	1	160 ⟩	0	1	1.0

Table 4.3: Visited configurations during an example execution of Simulated Annealing on CFM_b

configuration in $N(x_4)$ shown here improves the score and must, therefore, be selected.

In the next steps, the overall score is made slightly worse before it is improved, while the temperature is reduced. In the last iteration, here shown with $N(11)$ as the neighbourhood, the probability of being accepted with a score that is 1 worse than the current score is as little as 0.4. This time, the algorithm locates a candidate configuration, $x13$, with score 0, and it returns with a global optimal solution.

Although it is a quite straightforward algorithm, Simulated Annealing can be adjusted in many different ways. In this design, some of the adjustments are defined by the user. The maximal number of iterations and initial temperature is set as parameters. Another adjustment that may have some impact on how the algorithm performs is the cooling schedule.

In the design presented here, the cooling schedule is defined by the difference between the initial value of t and K. Different strategies could be implemented to make the cooling schedule slower or faster. Alternatively the cooling schedule could be made dependent on the progress, defined by, for instance the score of the best configuration found so far [5].

4.3.3 Design and implementation of Genetic Algorithm

The Genetic Algorithm has several possible ways it can vary. It combines several strategies, and makes use of a number of parameters. This section presents the design of the Genetic Algorithm that was used in this thesis and discusses some of the ways it could be adjusted further.

The design is presented in pseudocode in Algorithm 4.3. The algorithm includes a number of parameters and functions and can be divided into four phases in addition to the initialisation phase in line 1-10. In line 12-17 is the solution improvement phase, the third phrase in line 18 is the natural selection phase, in line 19-25 is the general fitness assessment phase, and in line 26-33 the evolvement phase takes place.

In the initialisation phase, a number of parameters are set. The parameters in line 1-6 needs to be defined before running the algorithm. The parameter n, decides the size of the population set P. The next parameter *term* is similar to the plateau threshold in Hill-Climbing and is used to set a limit of how long the algorithm is allowed to run without improvement. The random selection rate r decides the percentage of the population that is added regardless of their scores. The parameter b is used in the reproduction phase to decide the number of places where the parent molecules are split. The mutation probability m is also used in the reproduction phase and is the probability of a single value to be tweaked in a child chromosome. Finally, the e parameter is the number of elites who survives each generation and continues to live in the new population.

Two of the parameters above, *term* and the number of elites, e were set according to initial tests, and were not adjusted throughout the project. This was done to limit the complexity from all the different settings that can be set when evaluating Genetic Algorithm in this thesis. Due to how expensive each generation often is, compared to an iteration in Hill-

Algorithm 4.3: Pseudocode of the chosen design for Genetic Algorithm

```
1: Set n, an even number deciding the population size
2: Set term, a termination threshold.
3: Set r, random selection rate
4: Set b, number of crossover breakpoints
5: Set m, the mutation probability
6: Set e, the number of elite survivors each generation
7: P ← randomly generated population of size n,
8: s ← selectRandom(P)
9: bestGeneralFitness ← calculateGeneralFitness(P)
10: t ← 0
11: while t < term do
12:     for each x in P do
13:         if score(x) < score(s) then s ← x
14:         end if
15:         if score(s) = 0 then return s
16:         end if
17:     end for
18:     P ← naturalSelection(P, n, r)
19:     generalFitness ← calculateGeneralFitness(P)
20:     if generalFitness < bestGeneralFitness then
21:         bestGeneralFitness ← generalFitness
22:         t ← 0
23:     else
24:         t ← t + 1
25:     end if
26:     Q ← ∅
27:     for i = 0 to n do
28:         p1 ← tournamentSelection(P)
29:         p2 ← tournamentSelection(P)
30:         ca, cb ← crossover(p1, p2, b)
31:         Q ← Q ∪ {mutate(ca, m), mutate(cb, m)}
32:     end for
33:     P ← Q ∪ elitism(P, e)
34: end while
35: return s
```

Climbing and Simulated Annealing, the termination counter was set quite low. Before the results of running GA is presented the other parameters will be discussed and set.

The population P is generated randomly in the initialisation phase. This is done by generating configurations at random and adding them to the population n times. The size of P has a great impact on the performance of the algorithm. A large population demands more time for each iteration since during an iteration, each candidate in a population is replaced. On the other hand, if the population size is too small, it might take many iterations to get to a solution. Alternatively, the traits that would lead to a solution might be lost because there is no room for candidates with such traits. The initial solution s is selected at random among all elements in P.

The general fitness of a population is used to evaluate if there is an overall improvement. What the function $calculateGeneralFitness$ does is to take the mean of the scores of all the configurations in the population. The termination counter t is used to count the number of subsequent iterations where the general fitness has not improved. If t reaches the termination threshold, defined by $term$, the algorithm terminates with a sub-optimal solution. The counter t is updated in the third phase, line 19-25, on the population, after it has undergone natural selection.

Natural selection is one of the standard techniques in Genetic Algorithm but how it is applied varies greatly. In this design, natural selection usually starts with a large version of the population and reduces it. Since P usually is smaller after natural selection has occurred, the phase of updating s is executed just before it. In the first phase of the iteration, the algorithm goes through the current population P and looks for improvements of s. If it finds an improvement, s is updated. If the improvement is a configuration with score 0, the algorithm terminates by returning the solution.

The function $naturalSelection$ takes the population, which is at least of size n. In fact, In this scheme of the algorithm the population will usually be $2n$, when entering $naturalSelection$. The function also takes as input n, which will be the size of the population returned by the function, and the random selection rate r. As seen in Algorithm 4.4, the first step in the natural selection process is to locate the n candidates in P with the highest score and cut off the rest. Let the set of surviving candidates be S. This is the part of natural selection that represents the survival of the fittest.

This strategy alone could lead to a group that is too homogenous, where certain significant traits may not be represented. Following the evolution analogy, the strategy could lead to a population with too much inbreeding, where lack of biodiversity within a species ultimately leads to a halt in development. Because of this, it is common to introduce some sort of random selection. One common way to implement this is to replace a random configuration in S with a random configuration from the set of cut-off candidates, $P \setminus S$.

However, in testing the algorithm, a slightly different random selection strategy proved to work better. The strategy is inspired by a common technique in conserving endangered species. To secure the genetic

Algorithm 4.4: Natural Selection

Input: the current population P, an even number n, and a random introduction rate r
Output: a population S of size n

1: $P_{sorted} \leftarrow sort(P)$
2: $S \leftarrow \emptyset$
3: $i = 0$
4: **for each** c_i **in** P_{sorted} **until** $i = n$ **do**
5: $\quad S \leftarrow S \cup \{c_i\}$
6: $\quad i \leftarrow i + 1$
7: **end for**
8: $release \leftarrow n \times r$
9: **while** $release > 0$ **do**
10: $\quad c_r \leftarrow generateRandomCandidate()$
11: $\quad randomlyRemoveOne(S)$
12: $\quad S \leftarrow S \cup \{c_r\}$
13: $\quad release \leftarrow release - 1$
14: **end while**
15: **return** S

diversity of a small population of a certain species, nature conservationists release groups of individuals of that species from a different population. These introduced individuals are results of an evolutionary development under different circumstances and have fresh genetic material to bring into the new generation of that population.

In the *naturalSelection* function, these foreign individuals are generated randomly. While this does not ensure that the released individuals are sustainable, in the sense that the material they bring will lead to higher scores, it ensures a greater genetic variation than what reintroducing candidates from the previous generation do. The fact that some of these candidates might not be sustainable does, however, indicate that a significant number of candidates should be introduced. In this function, the number of releases is defined by r and is one of the parameters that was tested for this thesis.

Finally, after the population has been assessed and undergone natural selection comes the evolvement phase. The evolvement phase applies several different techniques to create a new generation Q of the population. The techniques used in this design are called crossover, mutation, and elitism.

Like natural selection, crossover is also a standard technique in Genetic Algorithm, but just like natural selection, it comes in different forms and may vary greatly for different problems. Usually, crossover involves the selection of two or more parents from P and splitting and splicing the parent chromosomes to reproduce new candidate children.

One way this strategy varies is in the selection of the parents. In this design, the parents are selected by using a form of tournament selection.

A pre-defined number of random candidates are selected from P. These are competitors in the tournament and the one with the highest score is returned. The size of a tournament can have an impact on the performance of the algorithm. Usually, the size is 2 or more, since a tournament of size 1 is simply random selection. Smaller tournament sizes mean more randomness in what parents are selected while larger tournaments could be causing more homogenous generations. Different tournament sizes were tested to find a good level.

The *crossover* function takes the two parent candidates that were selected. It also takes b, the number of crossover breakpoints, set by the user. Assuming that the parents are of the same length l, the *crossover* function selects b numbers that all are between 0 and l. Then the two parent vectors p_1 and p_2 are both split into $b+1$ slices, decided by the breakpoints. The pieces are merged together, making up to new child vectors. For instance, if $b = 2$ and the three breakpoints are 3 and 10, the first three positions in the first child vector c_1 will be equal to the first three positions in p_1. The same will be the case for c_2 and p_2. The positions 3 to 9 in c_1 will, on the other hand, be taken from p_2, while c_2 gets these vales from p_1. For the last positions, 10 to l, they switch again.

When crossover is finished, the child vectors are added to Q, but not before they undergo possible mutation. The *mutate* function takes a candidate vector and a probability m. In *mutate* each value in the vector has a probability of m to be tweaked, which follows the same principle as the way a neighbour is generated in Hill-Climbing and Simulated Annealing. For binary positions, referring to a feature, the value is flipped, while attribute values are either increased or decreased by one.

In the last step of the evolvement procedure, P is switched out with Q. This involves dismissing all the candidates of the previous generation. However, by applying elitism, the e candidates in P with the highest scores are also added to the new generation. Notice that, since two children were added in each iteration of the evolvement phase and after applying elitism, $|P| \leq 2n + e$, even though n is the population size. This is taken care of during the natural selection phase in the next iteration of the algorithm.

Table 4.4 shows an example of the evolution throughout an execution of Genetic Algorithm. It only shows the configurations that are part of the evolution leading to s. The seven configurations represented in the table can be structured into a binary tree by parent relations as shown in Figure 4.4. Each level in the tree represents what generation the configurations are part of.

In this example, the generated population P in the first generation consists, among others, of the configurations a, b, c, and d. The configurations a and b are selected for crossover, resulting in two children, where one, p is represented in this example. The black values in the parent configurations represent the parts of the vectors that are spliced together to create p, while the grey values are spliced together to represent the other child, which is not represented here.

The final configuration p is also a result of a mutation in position 7, where the value is flipped from 1 to 0. p is added to the second generation

Gen			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		score
1*st*	$a =$	$\langle$	1	1	1	1	1	1	1	0	0	1	0	1	1	1	1	1	0	$\rangle$	2
1*st*	$b =$	$\langle$	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	110	$\rangle$	4
2*nd*	$p =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	$\rangle$	1
1*st*	$c =$	$\langle$	1	1	1	1	1	1	0	0	0	1	0	1	0	1	0	1	180	$\rangle$	2
1*st*	$d =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	110	$\rangle$	1
2*nd*	$q =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	180	$\rangle$	1
2*nd*	$p =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	$\rangle$	1
2*nd*	$q =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	180	$\rangle$	1
3*rd*	$s =$	$\langle$	1	1	1	1	1	1	1	0	1	0	1	1	1	1	0	1	0	$\rangle$	0

Table 4.4: Evolution during an execution of Genetic Algorithm on CFM_b

The table shows a possible path to the solution s through three generations. The bottommost table shows how s is generated from applying crossover on its parents p and q. The two tables above that shows how p and q were generated from their parents. The red value in p indicates that a mutation occurred.

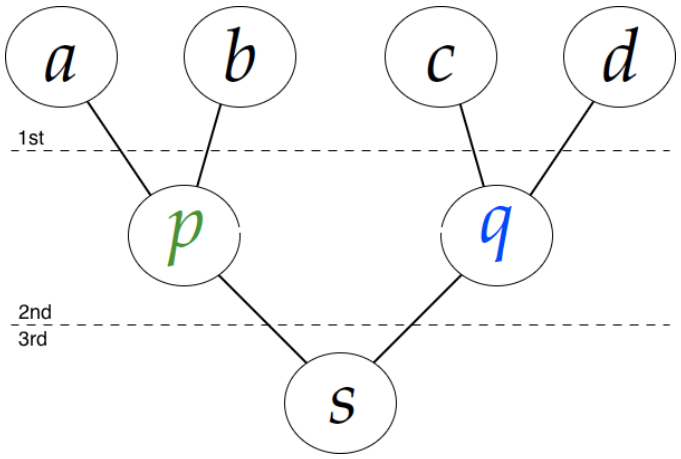


Figure 4.4: Evolution during a Genetic Algorithm execution represented as a tree

The graph shows the ancestors to solution s through three generations, where grandparents a, b, c, and d are in generation 1

and is later picked as one of the parents that will result in s. The creation of the other parent, q, is shown in the second part of the table, where it is a result of crossover of c and d. In the third part of the table, s is created and added to the third generation, where the algorithm acknowledges it as an optimal solution.

There are several ways to vary Genetic Algorithms, and different variations may show significant improvements for different problems. In designing GA for this project a few different strategies were tested, while other could prove to improve the results with further testing. One variation is that during an iteration, the algorithm could, by some probability, decide not to execute crossover. Instead it may select one candidate that undergoes mutation. To prevent this strategy from being a pure copy of candidates from one generation to another, when the mutation probability is low, it would perhaps be wise to ensure that at least one, alternatively, exactly one value is mutated. This could replace the mutation of newly generated candidates, or be an addition to that.

Different crossover strategies also exist. One strategy is to pick three parents x, y, and z, instead of two. The new candidate c is generated by going through each position i, and let the value c_i be x_i if $x_i = y_i$. If x_i is different from y_i, then c_i gets its value from z_i.

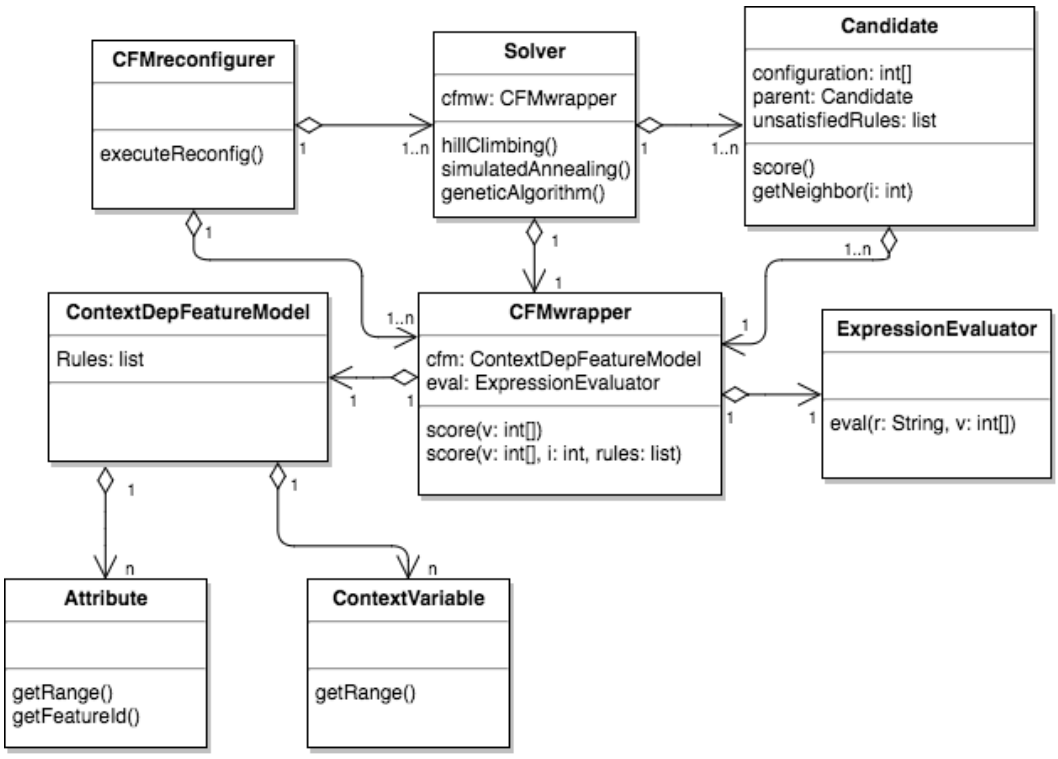


Figure 4.5: Class diagram of the CFM Reconfiguration Engine

4.4 Architecture and technical specifications

In order to effortlessly apply the metaheuristics on large sets of CFMs, and access the results, a reconfiguration engine was developed as a part of the project. The reconfiguration engine implements the designs of the metaheuristics that are presented in this thesis. It takes as input a file with information about the data set and a list of where to find the CFMs in the set. This section introduces and describes the reconfiguration engine's architecture, as well as the format and structure of the different inputs it uses and the output it produces.

4.4.1 Architecture of the reconfiguration engine

The architecture of the CFM Reconfiguration Engine is outlined as a class diagram in Figure 4.5. The class CFMreconfigurer needs an input file of a specific format. The input file contains a list of the Context-dependent Feature Models that are to be reconfigured. It also contains additional information such as the size of the models and size of the data set. To be run, the CFMreconfigurer requires some settings to be defined. It can be set to run one or more of the three metaheuristic algorithms separately. If more metaheuristic approaches are turned on their result will be stored independently of each other.

Each of the metaheuristics requires certain settings of their own. For instance, initial temperature may be adjusted for Simulated Annealing and Size of the population can be set for Genetic Algorithm. In Table 4.5 the different settings and their default values are shown. When fully set up, the CFMreconfigurer can be initiated by running executeReconfig().

The main task of the class CFMreconfigurer is to execute the metaheuristics on the CFMs, according to the settings given to it. It collects the results and presents them in the output file. For each CFM object referred to in the input file the CFMreconfigurer imports the model. It then pre-processes it and runs the selected metaheuristics in order to try and find a valid configuration. Importing and pre-processing the CFM is taken care of by the CFMwrapper, while the reconfiguration is done by the class Solver.

The CFMwrapper class takes care of many of the tasks concerning the CFM and is used by other classes to access the CFM. The ContextDepFeatureModel class is where the results of importing and pre-processing the external model are stored. When an CFMwrapper is created by the CFMreconfigurer, it is given the path to an external model in a JSON file. The JSON model is imported and the CFMwrapper creates an instance of the class ContextDepFeatureModel to contain all the imported data. This data normally includes the CFM, as earlier defined with rules, features, and attributes, as well as a context. When importing, the CFMwrapper also preprocesses the ContextDepFeatureModel instance. This involves filling the set T, removing redundant rules and creating intervals for attributes. Attributes and context variables are stored by their own classes in the ContextDepFeatureModel class.

When the CFMreconfigurer has constructed an CFMwrapper and

Setting	Description	Default value
applyHillClimbing	Turns Hill-Climbing on	*false*
applySimulatedAnnealing	Turns Simulated Annealing on	*false*
applyGeneticAlgorithm	Turns Genetic Algorithm on	*false*
setHillClimbMax-PlateauIterations	Sets the number of allowed iterations on a plateau in HC	0
setHillClimbNumber-OfExecutions	Sets the number of times HC is repeated until it finds a global optimal solution	1
setSimAnnealInitial-Temperature	Sets the initial temperature of SA	2.0
setSimAnnealMax-Iterations	Sets the number of iterations to run SA with no global optimal solution	1000
setSimAnnealNumber-OfExecutions	Sets the number of times SA is repeated until it finds a global optimal solution	1
setGeneticAlgInitial-PopulationSize	Sets the population size in GA	64
setGeneticAlgCrossover-BreakPoints	Sets the number of break points where configuration vectors are split during crossover in GA	1
setGeneticAlgMutation-Probability	Sets the probability for a single value in a configuration to be mutated in GA	0.02
setGeneticAlgRandom-SelectionProbability	Sets the probability for a single value in a configuration to be mutated in GA	0.08

Table 4.5: Settings in CFMreconfigurer

initiated it with a CFM, it proceeds by creating an instance of the Solver class and feeds it the CFMwrapper. The solver is where the metaheuristic algorithms are executed on a CFM and where the results of these executions are saved. Because a Solver object is connected to one specific CFMwrapper, a new solver must be created for each CFM. The CFMreconfigurer stores each solver, and can withdraw all results after it has attempted to reconfigure all models.

The different metaheuristics in the solver instance get started by CFMreconfigurer according to the settings previously defined. When any of the metaheuristics are executed, the solver first check if the CFM can be reconfigured by a trivial configuration. Here, a trivial configuration refers to a configuration where all attributes are 0 and a feature is selected if, and only if, it exists in T. Additionally, if Hill-Climbing is allowed to attempt more than one execution in the first execution, the algorithm will be given the trivial configuration as its initial candidate.

In order to compare the different approaches, the solver stores the results for Hill-climbing, Simulated Annealing and Genetic Algorithm separately. In each of the metaheuristics, the solver instance generates and makes use of candidates. The Candidate class stores a vector representing that candidate's configuration. Candidate also has a reference to the CFMwrapper, containing the CFM which it is a candidate for.

Another reference in Candidate is to its parent. This is set when Hill-Climbing or Simulated Annealing generate a neighbour from a candidate. The getneighbour() function takes a neighbourhood index i, indicating which neighbour to return, and generates a new neighbour. The new neighbour gets a reference to the original candidate as a parent. The index is not to a pre-generated set of neighbours but an indication to what position that will be tweaked. In the case of attributes, the index also indicates if the value is increased or decreased. The reason for this functionality is to increase efficiency by only generating neighbours that are actually used by the algorithm. This prevent the algorithms to generate a whole neighbourhood each time the main candidate is replaced. The neighbourhood indexes are set in the pre-processing phase and take into account what features are contained in T.

One of the commands most often run during the execution of a metaheuristic is the score function. The score function calculates the score a candidate for the current CFM and context. When a candidate is asked for its score it sends the request to the CFMwrapper, which in turn goes through each of the rules from ContextDepFeatureModel. For each rule, the rule and configuration is sent to an instance of the ExpressionEvaluator to evaluate if the configuration satisfies it. Finally, the CFMwrapper returns the number of rules that were not satisfied, making up the score of the candidate.

When a candidate c has a reference to a parent p it uses the results in p to calculate its own score. Instead of calling on CFMwrapper to evaluate each rule, it only asks for it to evaluate the rules that contain a reference to the feature or attribute in which c differs from p. The parent p has a set of rules that it doesn't satisfy. The CFMwrapper receives a clone of this list and

updates it by adding rules that are not satisfied c and removes rules that are. The score of randomly generated candidates needs to be calculated by evaluating each rule, but in Simulated Annealing and Hill-Climbing, the more efficient procedure is applied in all iterations following the first.

The ExpressionEvaluator is used by the CFMwrapper and is set up individually for each CFM. It saves intermediate and final results to speed up the evaluation throughout an execution of a metaheuristic.

When the CFMreconfigurer has run through all the CFMs it creates a result file by going through and collecting data from each solver. Additionally, if there is a file with results from HyVarRec in the path of the dataset, the CFMreconfigurer goes through the result and combines it with the metaheuristic results. Since the metaheuristics can't determine if a CFM is void, this procedure is necessary to analyze results based on the subset of the dataset that only contains valid models. It is assumed that the file with results from HyVarRec contains objects in the same order as the dataset input file. Finally, the CFMreconfigurer can be instructed to only select a given number of CFMs from the data set. If the result file from running HyVarRec exist and a number of models to import is given, the CFMreconfigurer will skip all CFMs where the result list reports that there are no solutions. This allows the reconfiguration engine to run the metaheuristics exclusively on non-void CFMs.

4.4.2 Input and output formats

The CFMreconfigurer requires a text file as input containing, among other things, a list of models to be imported. The models must be in a JSON format and follow a specific structure. This section gives a brief description of the input and output specifications. First the main input file, here called the dataset file, and the output file, containing analysis data, will be described. Then, the JSON structure for the external models and finally the file containing external results is described.

The dataset file is a text file that begins with some lines of information and then a list of file names of the models to be imported and additional information of the model. The first line contains the path to the dataset file and the second contains the path to the folder where all the external models exist. Then follows an optional number of lines with additional information about the dataset. These lines must begin with a string followed by ':' making up the information description, then follows a whitespace and the information part as an integer. These lines end with a newline. One of these lines should be "Size_dataSet: x", where x is an integer telling the number of files that the CFMreconfigurer are about to import. Otherwise, these lines of information only serve the purpose of adding information to the analysis file and are not used as part of the reconfiguration procedure.

To mark the end of the information lines and the beginning of the list of models, the dataset file must have a line with the tag "feature_models". The next line after that is reserved for headlines and will be skipped. The rest of the lines must all contain the following five parts, in order: the name of the file containing the model, in JSON format, the number of features in the

model, the number of attributes, the total size of the CFM, and the number of variables in the context. These parts are separated by tabs.

The output file is written to the directory "analyses", and is also a text file. These files can be divided into three parts. The first part is the general dataset information, directly reproduced as stated in the input file. Then, for each of the metaheuristics that were used by the application, the settings for the metaheuristic and its results are presented. Examples of results presented in these parts are what share of the executions gave successful results, and how much time did the metaheuristic spend, on average. If the application found a file with a complete list of results from HyVarRec, including results telling that a model is void, then these calculations ignore all results made on void models.

Finally, the file produces a list of individual results for each combination of a model and a metaheuristic that was used by the application. These lines are separated by tabs and contain that name of the file containing the original model, an abbreviation of the metaheuristic, the time performance and number of iterations used by the metaheuristic for that model, the resulting score of the configuration, and the configuration. The configuration is a string of integer values where the attribute values are prefixed with a hyphen, "-". Additionally, the configurations imported from the file containing HyVarRec results are also included in the file, after each set of metaheuristic results and marked with the abbreviation "HVR". If HyVarRec concluded that the model was void, instead of a configuration, the value -1 is presented.

The CFMreconfigurer is built to accept external CFMs in JSON format, with structure defined by HyVarRec [10]. This makes it possible to compare the results of the metaheuristic algorithms of the application presented here and the constraint programming approach provided by HyVarRec. An external model is a JSON object with the following properties: constraints, attributes, contexts, configuration, and preferences. The constraints are logical rules in the form of strings. Their atomic parts are made up by an id reference to a feature, attribute, or a context variable and a value. They are connected by one of the following operators: $=, \neq, <, >, \leq$, and $\geq$. These constraints are directly translatable to the rules described previously in this thesis, and presented in Table 3.1.

The attributes and contexts, in this paper called context variables, are objects with an id and their ranges. Additionally, attributes keep a reference to the feature that owns it. The configuration contains three different arrays of data. One array contains the feature ids of the features that must be selected, another assigns values to the attributes, and the third assigns values to the context. The first of these arrays is analogous to the set T in the formal data structure of this thesis. The assignment of values to attributes will be ignored by $CFMreconfigurer$ since the metaheuristics start with randomly generated configurations. The last array is the actual context function and is added to the ContextDepFeatureModel instance in the application. The last property, preferences, is ignored.

The optional file containing external results also follow the structure defined in HyVarRec. This file must contain a list of JSON objects with two

properties. One is "result", which either holds the string "sat" or "unsat". If CFMreconfigurer reads this file and finds an object where the result has the value "unsat", it will conclude that the corresponding model is void. The other property is an array of features selected in the satisfying configuration and the values that the configuration assigns to the attributes.

Chapter 5

Evaluation

The main goal of this thesis was to investigate the ability of metheuristics to reconfigure Context-dependant Feature Models. To measure their ability, the metaheuristic algorithms in the reconfiguration engine were evaluated based on the amount of successful reconfigurations and their running time. In order to carry out the evaluation, the algorithms were tested on large sets of CFMs that were representative of real-life problems. To obtain such a set of CFMs, an option could be to use pre-defined CFMs. The optimal approach would then be to run the tests using a collection of real industry models. However, as CFMs are not a standard that is widely used in software engineering to this date, that was not possible.

The alternative pursued in this project was to create a generator that could produce large and varied sets of Context-dependent Feature Models randomly. One advantage of this option is that the results are not limited by the lack of generalisation that one might get from pre-defined models. Even though pre-defined models would have represented realistic structures and situations, they might not have taken into account currently unusual, yet possible problems, which could potentially have weakened the validity of the evaluation.

A disadvantage with the strategy of generating datasets at random is that there are no guaranteeing that a randomly generated CFM does represent real-life situations. For instance, it is possible for a Context-dependent Feature Model generated at random to have a valid configuration where the root is the only feature selected. Even if that CFM have sub-trees with complex structures, the correct brute force approach would find the configuration in milliseconds. Besides, a configuration like that would not represent a practical software. Another challenge is to avoid producing void Feature Models and Feature Models with dead features. Dead features may especially be a problem if they have many descendants in the tree structure since it causes all of the descending features to be non-selectable as well. In this chapter, this is referred to as a dead sub-tree. All of these challenges will be addressed in this chapter.

To generate CFMs, this project made use of BeTTy [2, 13], a Java framework "supporting benchmarking and testing on the analysis of Feature Models". With BeTTy it is possible to create large numbers

of Feature Models, with or without attributes, in a short amount of time. Furthermore, BeTTy provides a number of parameters to be set, to specify certain traits of the Feature Models in a dataset. The framework was implemented in a self-developed tool, here called the Dataset Generator. The full source code of the Dataset Generator can be found at github.com/magnurh/CFMmetaheuristicReconfig_thesisProject. The Data Set Generator applies BeTTy to generate Feature Models. It then extends the Feature Models with Validity Formulas and a context to create a Context-dependent Feature Model.

The parameters set in the Dataset Generator, many of which are used by BeTTy, may have an impact on the complexity of a CFM. Furthermore, some traits in a CFM may create a complex problem for one metaheuristic, while other traits increase the complexity for another. In general, CFMs with few products may be more difficult to reconfigure. However, since the metaheuristic algorithms rely on the score of a configuration, which is closely connected to the constraints in the CFM, there might be other characteristics of a CFM that impacts the algorithms' success. In that regard, this project aimed to test the metaheuristics on CFMs with a varied level of constraints, to investigate if they are more vulnerable to some types of constraints over other types.

In this project, a number of datasets were created, differing in the traits of the CFMs and their sizes. Based on the datasets, the metaheuristic algorithms were evaluated and compared with one another and to the CP-solver HyVarRec. The metaheuristics' performance is evaluated based on the number of valid CFMs it could successfully reconfigure, as well as their solving time. Since metaheuristics don't exhaust the search space, they are never able to detect if a Context-dependent Feature Model is void. To ensure that the tests were not carried out on void CFMs, HyVarRec was applied to pick out CFMs without products, so that they could be ignored.

This chapter introduces the Dataset Generator that was developed and used for the purpose of evaluating the metaheuristics. Then, the thesis will present the approach in testing the algorithms, including the datasets that were generated and used.

5.1 Generating sets of Context-dependent Feature Models for testing

In order to generate sets of Context-dependent Feature Models, for testing the Reconfiguration Engine, this thesis used a self-developed application, here called Dataset Generator. The Dataset Generator is constructed to create a fixed number of Context-dependent Feature Models that are restricted by certain pre-defined parameters. It uses the frameworks BeTTy [2] and FAMA [1] to generate the basic Feature Models. These models are then extended with VFs and a context before they are exported to files in JSON format according to the format used in the Reconfiguration Engine.

The Dataset Generator offers two ways to generate datasets. One is quite straight forward and lets BeTTy do most of the work without being

Setting	Description	Default value
sizeDataSet	Number of features in the dataset	10
numberOfFeatures	Number of features in each CFM	50
percentageCTC	amount of Cross-Tree-Constraints in the model	30
probMand	probability for a relation to be Mandatory	random
probOpt	probability for a relation to be Optional	random
probAlt	probability for a relation to be Alternative	random
ProbOr	probability for a relation to be Or	random
minAttrRange	bottom of attribute range	0
maxAttrRange	top of attribute range	100
contextMaxSize	max number of variables in the context	10
contextMaxValue	max number of values a single context variable can take	10
maxNumberOfVFs	max number of validity formulas in the model	12

Table 5.1: Settings in DataSetGenerator

concerned with void models or dead features. In the other technique, the Generator can be asked to apply a reasoner to reduce the number of void CFMs in the datasets. Additionally, it will apply some strategies to avoid some common cases where randomly generated cross-tree constraints (CTCs) creates dead features and dead sub-trees. This section will describe the architecture of Dataset Generator application and discuss how the Generator solves some of the cases with dead features.

5.1.1 The Dataset Generator architecture

The Dataset Generator is set up with a number of parameters, which can be viewed in Table 5.1. Some basic parameters define the size of the dataset and the size of the models. Other parameters affect the structure of the CFMs. Among these are the different parameters that define the probabilities and amount of relations that are of a specific type, as for example Alternative and Cross-Tree-Constraints. These constraint parameters are used by BeTTy to generate the basic Feature Model structure. Parameters concerning the context and VFs are used by the Dataset Generator application itself.

Figure 5.1 shows the process of how each model in a dataset is created. The Dataset Generator runs BeTTy, and feeds it some of the parameters.

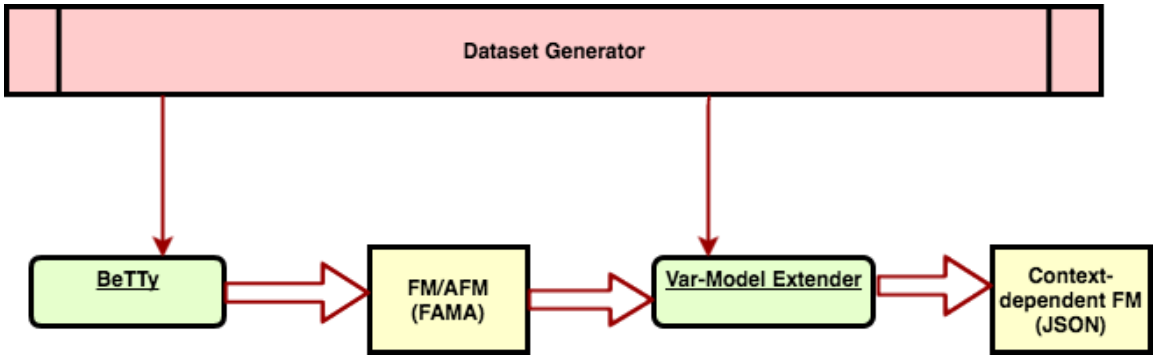


Figure 5.1: The Dataset Generator's process for creating a Context-dependent Feature Model

For each model in the dataset, the Dataset Generator uses BeTTy to create a Feature Model (with or without attributes) in FAMA format. The model is read by a Var-Model Extender that transform and extends the model to create a Context-dependent Feature Model in JSON format

BeTTy outputs a Feature Model in FAMA format, a format defined by the developers behind BeTTy. Then, the generator executes a process, here called a VarModelExtender, that reads the Feature Model. The process makes an instance of the Feature Model and generates a context. Based on the context it extends the Feature Model with Validity Formulas before writing it as a CFM to a JSON file. The Dataset Generator repeats this process for the number of times set beforehand. Finally, it outputs a text file, with information about the dataset, that can be used as input by the FMreconfigurer.

The application makes no promises that the produced dataset does not contain void models or models with dead features. However, two techniques are applied to try and reduce the number of CFMs that are "uninteresting", because they are unlikely to represent real-life problems. The first technique is used to prevent any of the basic Feature Models from being void. BeTTy provides a SAT-reasoner that evaluates a basic Feature Model, only consisting of features and constraints between features, to establish if the model is void. The reasoner is applied on the first stage Feature Model, and if it is recorded to be void, BeTTy is executed again, to create a different Feature Model.

The other technique tries to eliminate Feature Models that are made satisfied when only a few number of features are selected. An example of such a Feature Model is one where the root only has optional relations to its children, or it has one mandatory relation to a child, which in turn only have optional relations leading down from it. The assumption is that Feature Models like this does not sufficiently represent real life problems as the dataset was intended to. The reason is that when a Feature Model can be configured by selecting the root alone there is in practice little difference between a complex and a simple Feature Model.

The technique tries to increase the probability that a Feature Model

represents the intended complexity by following the constraints from the root and count the minimal number of paths down to a given level. Optional constraints are ignored since they in many cases can be excluded from the configurations. Mandatory constraints are followed, and in the Alternative and Optional groups, the one feature with the lowest result is added to the total number of paths. When counting the end-points in the selected level, all constraint types are counted as 1, except optional constraints which are not counted.

For instance, the CFM in Figure 2.2 has a minimum of three paths at two levels, where the root level is number zero. F_1 leads to an alternative group which counts as 1. The same is the case for F_2. The feature F_3 has one mandatory and one optional constraint leading from it, which adds 1 to the sum. The optional constraint from F_0 to F_4 is ignored. F_5 does not lead anywhere, but it is still counted. One practical reason for this strategy is to increase the probability of VFs to come into play.

The application runs this restriction check repeatedly and selects a model if it reaches some threshold, or if it runs out of time, the Feature Model with the most path end-points is selected. The threshold is dependent on the size of the Feature Model. When the process of generating a basic Feature Model is finished the Dataset Generator executes the Var-Model Extender, which takes the Feature Model and produces an instance of a Context-dependent Feature Model with a context.

The context is created randomly, within the restrictions set by the user, as a set of variables with a certain range and a value within that range. The VFs and attributes are generated simultaneously, under the assumption that attributes are only relevant in the CFM if they are included in a constraint. VFs are the only type of constraint created by this generator that contains attributes, as a part of the formula. It is also assumed that both attributes and VFs must belong to a leaf feature and that when an attribute is included in a VF, the VF and the attribute must share the same feature.

A VF is generated as a conditional between a feature and a context variable, or between a feature and an expression involving a context variable and an attribute. The first of these is of the form $F = 1 \rightarrow C \circ v$, where $F = 1$ means that feature F is selected, $\circ$ is a comparison operator and v is the value that the value of the context variable C is compared to. The latter is of the form $F = 1 \rightarrow (C \circ v \rightarrow a \diamond w)$, where $\diamond$ is a comparison operator, a is an attribute and w is the value compared to the value of a. The Var-Model Extender makes sure that F is the owner of a and that the values v and w are within the defined ranges of, respectively, C and a.

When a VF is generated, first, one of the leaf features is selected at random, then one of the context variables. Which of the two forms a VF takes is selected randomly, except from in one case. If the selected feature has a "mandatory path" leading to it from the root, the second form is selected, since the first form would make the CFM void in some instances of the context. An example of this case is the feature F_{11} in Figure 2.2.

If the VF is on the second form, an attribute is generated and added to F and simultaneously added to the VF. It is, however, possible for a feature to

have several VFs, analogous with the three VFs connected to F_{11} in Figure 2.2, which are combined into one VF by conjunctions. An assumption in this generator is that a feature can own maximum one attribute, so in the case that more VFs are added to one feature the same attribute is re-used.

It is worth noting that, although, the probability that the basic Feature Model is void is low, there are no guarantees that it stays that way after context and VFs have been added. Furthermore, for large models, checking if they are void can be very time consuming and the user may want to turn that function off.

5.1.2 Dead features and dead sub-trees

A flaw in the BeTTy framework, when Cross-Tree-Constraints (CTCs) are added, is that the constraints are added by choosing two sub-trees at random, and then, a feature in each of these sub-trees is randomly picked. In most cases this provides good CTCs and make intuitive sense. However, there may occur incidents where the two features are not logically in separate sub-trees.

The consequence of a CTC being logically a constraint that doesn't go across two sub-trees may be none, but it may also produce dead features and it may make a whole sub-tree void. Here, this is called a dead sub-tree. The best case, the constraint has no logical implications and is already captured by the constraints in the tree. An example of this is if features F_a and F_b are mandatory children of F_p, and a CTC $F_a \rightarrow F_b$ is added. Since the Feature Model already contains the constraints $F_a \rightarrow F_p$ and $F_p \rightarrow F_b$, this is already captured.

On the other hand, if the CTC was $F_a \rightarrow \neg F_b$, which is the same as $\neg(F_a \wedge F_b)$ it would be a contradiction to the constraints in R. In this situation F_p becomes a dead feature, because no valid configuration of the Feature Model can select it. If a configuration selects F_p it must also select F_a and F_b, because of the mandatory relations, which contradicts the CTC. Since F_p is a root of a sub-tree, all the features in the sub-tree are dead features, making it a dead sub-tree.

A dead sub-tree is not necessarily a significant problem if it only contains the three features above. However, if F_p is a child of the root, the situation may result in half of the features in the Feature Model being dead. In this project, dead sub-trees are in fact an important part of the problem that is explored, if they are created by the current state of the context. Thus, most of these CTCs are removed from the model, so that situations with dead features and dead sub-trees mainly happen as a result of the current context values and the VFs.

It is worth noting that the situation will most likely not occur if F_p is in T, which means that it is either the root, or it has a mandatory path from the root leading down to it. This situation would lead to the whole model being void, which is taken care of earlier in the production chain. The procedure check for void Feature Models does not, however check for void sub-trees. It is therefor necessary for the Dataset Generator to add this additional step to ensure that the Feature Models are sound.

The Dataset Generator check for two common situations where a randomly generated CTC is not strictly a constraint between features in two sub-trees. The first situation is that the feature on the left-hand side, F_l can logically be interpreted to be a predecessor of the feature on the right-hand side, F_r. A feature which is the root of a sub-tree is a predecessor of all features in that sub-tree. Reversely, the features of the sub-tree are descendants of the root feature. In the other situation the two features F_l and F_r both have predecessors in the same Alternative-group. If the generator finds that a CTC, generated by BeTTy, fall into one of these two situations, it is not added to the final Context-dependent Feature Model. Since The Dataset Generator may remove a number of CTCs without adding any, the different CFMs in the final dataset will have different numbers of CTCs.

To check for the first situation, the Dataset Generator locates the feature closest to the root which has a mandatory path leading to F_l. Lets call this mandatory predecessor F_{ml}. If the parent of F_l does not have a mandatory relation to F_l, then F_l itself is selected. If F_{ml} is a predecessor of F_r through the regular constraints in the Feature Model tree (CTCs are not counted) the generated CTC is not included in the final Context-dependent Feature Model.

The reason behind this action is that F_l must be selected if F_{ml} is selected. Consequently, the CTC can switch out its left-hand side feature F_l with F_{ml} without losing its meaning. Now, there is a CTC from a root in a sub-tree to a feature further down in the sub-tree, which does not strictly qualify as a Cross-Tree-Constraint. The example above falls in under this situation, since F_p is a mandatory predecessor of F_a. If the CTC is on the form $F_l \rightarrow \neg F_r$, then F_r is a dead feature, and the sub-tree of which it is a root, is a dead sub-tree. A CTC on the form $F_l \rightarrow F_r$ creates, logically, a mandatory path from F_{ml} to F_r, no matter what relations already exist in the path $F_{ml}, \ldots, F_r$. This turns all Optional- and Or-relations in $F_{ml}, \ldots, F_r$ into mandatory relations. This does not necessarily create dead features, but it does violate the initial logic of the tree-structure. If any of the relations in the path are Alternative-relations, on the other hand, the CTC does create dead features. This is because, in any Alternative-groups passed through by the path $F_{ml}, \ldots, F_r$, there are features which are not in the path. These features can no longer be selected, meaning that they are dead features. Thus, the CTC is not added to the CFM.

To check for the second situation, the Dataset Generator looks through all Alternative-groups and check if they contain one predecessor of F_l and a different predecessor of F_r. If it locates a predecessor for each of the feature, the CTC is removed. Let the predecessor of F_l be called F_{pl} and the predecessor of F_r be F_{pr}.

The logic behind this action is quite straightforward. If the CTC is on the form $F_l \rightarrow \neg F_r$ it is superfluous. F_l can only be selected if F_{pl} is selected, and F_r can only be selected if F_{pr} is selected. Since F_{pl} and F_{pr} can not be selected at the same time, by a valid configuration, F_l and F_r can not be selected at the same time either. When turned around, this implies that for any descendant of F_{pl}, F'_l and for any descendant of F_{pr}, F'_r, there is

Group	CTCs	VFs	Context max-size	Mand prob.	Opt prob.	Alt prob.	Or prob.
S1	10	40	25	30	10	20	40
S2	30	10	10	25	25	10	40
S3	50	20	8	25	25	40	10

Table 5.2: Dataset used for separate testing of the metaheuristic algorithms

The values are percentages relative to the number of features (CTCs, VFs, and Context) or number of relations (Mand, Opt, Alt, and Or) in the respective Context-dependent Feature Model. Each group contains 300 non-void models where there is an equal number of CFMs of size 60, 90, and 120. In total the set contains 900 non-void models

an implied constraint that one excludes the other, $\neg(F_l' \wedge F_r')$. With that in mind, consider the case where the CTC is on the form $F_l \rightarrow F_r$. This CTC turns F_l into a dead feature, because the selection of F_l means that F_r must be selected, otherwise the CTC is violated. However, because of the implied constraint $\neg(F_l' \wedge F_r')$, the two features can't be selected at the same time. Consequently, F_l can not be selected and is a dead feature.

Applying these two strategies helps avoiding some of the dead features that may be created, but it does not guarantee that the final model comes without dead features. There may still exist less obvious relationships which ultimately creates dead features. For instance CTCs contradicting each other. As earlier mentioned, the state of the context, may also create dead features in the CFM. These cases, however, are assumed not to be undesirable, since they represent possible realistic problems and add complexity to the CFMs which is assumed to be related to the real-life problem of configuring Context-aware Software given different surroundings.

5.2 Executing tests

The three metaheuristic algorithms were evaluated with different groups of datasets over different steps. In the first step, the algorithms were evaluated separately to see how they perform with different settings. In the second step, the metaheuristics were each given settings, based on the findings in the first step, and their performance were compared against one another and in combination with one another. In the third step, the performance of the metaheuristic algorithms were compared to the performance of the Constraint Programming solver HyVarRec. This section will present how the different datasets differ from each other, and how the testing was organised. The results are presented in the next chapter.

5.2.1 Evaluation of the metaheuristic algorithms separately

As described in Section 4.3 the metaheuristic algorithms Hill-Climbing, Simulated Annealing and Genetic Algorithm all can be tuned with different

settings. Hill-Climbing can be set to run a number of times from different initial candidates. It can also be set to allow for a certain number of iterations when it encounters a plateau before it terminates. Simulated Annealing requires an initial temperature and the number of iterations it will take from the starting point before it terminates without an optimal solution. Similar to HC, SA allows for a parameter that lets it repeat executions with different initial candidates. Genetic Algorithm requires a number that defines the size of the population, a mutation probability, a random selection rate and the number of breakpoints used during the crossover procedure.

In order to establish how the different parameters affect the algorithm, a number of tests were run with different metaheuristic parameter settings. The tests on Simulated Annealing and Genetic Algorithm are both executed on the same datasets. The dataset was created by generating a number of Context-dependent Feature Models and selecting a subset of non-void models. The set was divided into three groups of 300 models each, where a group represent a set of specifications that makes a CFM in one group different from the CFMs in another. These sets of specifications will be called traits. Table 5.2 shows the compositions of traits represented by the three groups. For each of the three groups, there were generated an equal number of CFMs of size 60, 90, and 120.

The values in the dataset table are percentages relative to the number of features and the number of relations in the CFM. For instance, the set S1 contains CFMs with CTCs counting a maximum of 10% of the number of features in the CFM. The four last values are relative to the number of relations, not counting the CTCs. These are used by BeTTy. For instance, S1 contains CFMs where 30% of the relations are Mandatory relations and 10% are Optional relations. As a rule these four relation values adds up to 100%.

Hill-climbing was not evaluated separately since the effect of adjusting its parameters usually is predictable. By increasing the plateau threshold and the number of executions, the algorithm takes a longer time before terminating without an optimal solution. At the same time, the probability of succeeding is increased. An execution of HC rarely run for a very long time before terminating, compared to GA and to SA, unless the number of iterations in SA is set very low. Therefore, it is assumed here that both parameters can be set to a somewhat high level and still be competitive in terms of time consumption. One exception to this assumption is cases where the search space contains many plateaus of size, slightly smaller than the plateau parameter set for the algorithm. Such a case could lead to a small increase in number of successes, with a large cost in terms of time.

Simulated Annealing was evaluated on the dataset with the temperature t spanning from 100 through 300. The number of allowed executions was set to 40, with 1000 iterations per execution. Additionally, a second test was conducted with only one execution and 50.000 iterations, with the temperature spanning from 12 to 400. The purpose of this test was to establish whether SA provides better results when they are run many times

Prameter	Description	Values
m	The mutation probability	0.01, 0.03, and 0.06
bp	Number of Cross-over breakpoints	1, 5, and 12
$popSize$	Size of a population	128, 256, and 512
r	The rate of random candidates introduced in each generation	0.01, 0.04, 0.08, and 0.12

Table 5.3: Genetic Algorithm parameters that were tested

with few iterations, or one time with a large number of iterations.

Genetic Algorithm demands four parameters to be set, as shown in Table 5.3. Each of the parameters was tested on 3-4 different values. The tests were carried out on all combinations of the first three of these parameters, m, bp, and $popSize$. Then, the random introduction rate r was tested with combinations of the first parameters that proved to perform well.

5.2.2 Evaluation of the metaheuristic algorithms together

Following the separate tests of the metaheuristic algorithms, a larger dataset was constructed for comparing the algorithms to one another. This test, which will be referred to as the main test, has three principal purposes. The first purpose is to compare the performance of the three metaheuristic algorithms to one another, as well as how they perform when used together. The second purpose is to investigate how certain traits in the CFMs affects the algorithms' abilities to locate optimal solutions. The third is to compare the performance of the metaheuristics compared to the Constraint Programming solver HyVarRec.

For comparing the metaheuristics with one another, they were measured by the number of successes and time performance. Time performance includes two metrics, first of which being how much time was needed on average to obtain an optimal solution. The second metric was how much time on average it took before termination when no solution was found.

Additionally, the algorithms were measured in combination, so that the success rate and time performance of running two, or all three algorithms in sequence was recorded. For instance, if Simulated Annealing and Genetic Algorithm was to be applied, in that order, to reconfigure a CFM, it counts as a success if either of them was successful. How the time performance is recorded depends on the order which the algorithms are applied. If SA, in this case, was successful, the time needed to run GA was not considered. On the other hand, if GA was successful and SA wasn't, the recorded time would be the time needed to run both of them.

The combinations measured here assumes that the order in which

Trait set	CTCs	VFs	Context max-size	Mand prob.	Opt prob.	Alt prob.	Or prob.
A	15	17	13	30	20	30	20
B	35	17	13	30	20	30	20
C	15	33	13	30	20	30	20
D	35	33	13	30	20	30	20
E	15	33	27	30	20	30	20
F	35	33	27	30	20	30	20
G	35	17	13	25	25	25	25
H	35	17	13	50	0	25	25
I	35	17	13	0	50	25	25
J	35	17	13	25	25	50	0
K	35	17	13	25	25	0	50
L	35	17	13	10	10	40	40

Table 5.4: The traits represented in the main dataset

The values are percentages relative to the number of features (CTCs, VFs, and Context) or number of relations (Mand, Opt, Alt, and Or) in the respective Context-dependent Feature Model. Each group contains 1200 non-void models where there is an equal number of CFMs of size 60, 90, and 150.

the algorithms would be applied is first Hill-Climbing, then Simulated Annealing, and finally Genetic Algorithm. The four different combinations measured in this test were the cases of running HC, followed by SA, HC followed by GA, SA followed by GA, and running all three of them beginning with HC and ending with GA. The results from these combinations are compared with the results of running only one of them, in order to see if there are cases where they complement each other, or if they generally solve the same CFMs.

The performance of each algorithm, as well as the combinations, were recorded for the different groups of CFMs that make up the main dataset. The CFMs in the main dataset represents 12 different sets of traits as well as three different sizes. For each of the 12 trait sets, 400 CFMs of size 60 was used included in the main dataset along with 400 CFMs of size 90 and the same number of models with size 150.

The 12 sets of traits used to generate CFMs for the main dataset are displayed in Table 5.4. The first six sets, from A to F, contains varying levels of cross-tree constraints, the number of validity formulas, and the number of context variables. One of the goals of setting these variations is to investigate how much an increased number of CTCs impact the different metaheuristics' performance, and if an increase in the number VF's has a

similar impact. Another goal is to investigate the effect of increasing the number of VF's while the number of context variables stays the same, compared to both the number of VF's and the size of the context are increased.

In the next 6 trait sets, with names G-L, it is the proportion of relation types, within the trees, that are varying. The general goal of these CFMs is to have CFMs representing a large range of structural variations present in the dataset. The first of the sets, set G, is a control group with an even distribution of probability values throughout the four relation types. Following are four cases where one relation probability is increased to 50 while another is decreased to 0, while the balance between group relations and one-to-one relations are kept at the same level. Here, the goal is to investigate how large of an impact the different relations have on the metaheuristc algorithms' performances.

In the final set L, the balance between the two types of relations is broken as the probabilities related to group relations are increased to 80 in total. This should give an indication of how one would expect the algorithms to perform on CFMs with a large amount of group relations, at the expense of one-to-one relations. The other traits, concerning CTCs, VFs, and context size, are set to the same level as in trait set B.

In the main test, the metaheuristic algorithms are tested with a fixed set of parameters. The levels of the parameters were selected to achieve a balance between a high probability of success and good time performance, and was based on findings in the initial tests.

5.2.3 Evaluation of the metaheuristic algorithms compared to HyVarRec

The final evaluation performed is a comparison between the performance of the metaheuristic algorithms and the CP-solver HyVarRec. The comparison was based on their performances in locating configurations for the CFMs in the main dataset, which was presented above. In contrast to the metaheuristics, HyVarRec guarantees a solution and is instructed to locate the solution that is most similar to a given configuration. In all executions, the configuration given was the one where only the root is selected.

HyVarRec was run using the platform Docker, as described in the documentation of HyVarRec [10]. The measurement of time for an execution was performed using the Unix shell Bash. The running time was measured from the point of running the command that executes HyVarRec, to when a solution was received. The average time presented for HyVarRec was obtained from executions on CFMs with the same traits as the CFMs in the main dataset.

The comparison to HyVarRec is made with one fixed set of parameters for the metaheuristics. However, the metaheuristic algorithms may be adjusted to increase their success rates, by allowing them to spend more time. To get a good basis for comparison, two of the metaheuristic algorithms were tested on their performance when some parameters

impacting their running time were increased.

Chapter 6

Results

This chapter presents the results of the different tests that was run on the metaheuristic algorithms. The results from tests done separately on Simulated Annealing and Genetic Algorithm will be presented first. Following that, the three algorithms are compared with one another, based on how they performed in general and how they performed on certain subsets of the dataset. Finally, the metaheuristics are compared to the CP-solver HyVarRec.

The general results in this chapter, from the tests on the main dataset suggest that Simulated Annealing is a quite good algorithm for CFM reconfiguration. It has a high probability of succeeding in very short time, especially for CFMs of size 90 and smaller. Even better performance is obtained by running SA together with the highly efficient but less successful Hill-Climbing. The findings from the general results does not suggest that Genetic Algorithm have much to contribute when run together with HC and SA. In order to get a good understanding of GA's abilities for this problem, more testing is needed.

The results from analysing performance on CFMs grouped by their levels of CTCs, VFs, and context-size show interesting differences between HC and SA. Simulated Annealing show a considerable decline in number of successes on CFMs, when the level of CTCs increases. This decline gets bigger as the CFMs grow in sizes. Hill-Climbing, on the other hand, show a much more moderate decline. Furthermore, as the CFMs grow larger HC's moderate vulnerability towards CTCs does not grow. In contrast, SA show very little vulnerability towards an increase in the level of VFs and context-size, while HC show a significant decline in successes.

The results from comparing the metheuristics to HyVarRec support the general findings that HC and SA, when run together, is a well suited alternative. This is especially the case for the small and mid-sized CFMs in the dataset. Hill-Climbing and Simulated Annealing are able to reconfigure CFMs with a quite low cost, compared to HyVarRec. Their ability of succeeding decreases for larger CFMs. If SA is allowed to run longer it proves to be able to locate configurations for more CFMs.

The performance of an algorithm is measured in form of amount of successes and running time. The success rate is represented by a decimal

t	60	90	120	Avg
100	.937	.793	.640	.790
150	.963	.857	.663	.828
200	.963	.863	.713	.847
250	.957	.883	.670	.837
300	.967	.877	.733	.859
400	**.990**	.883	.753	.876
500	.977	**.907**	**.807**	**.897**

Table 6.1: Amount of successes when running SA with different levels of temperature 40×1000 iterations

number, where 1.0 means a hundred percent successful and .800 means that an optimal solution was obtained for 80% of the Context-dependant Feature Models in the set. Running time is shown in milliseconds. There can in many cases be a large difference between the average time spent on arriving at an optimal solution and average time needed before terminating with no optimal solution. Here the focus is on time needed to obtain a solution, while the running time in unsuccessful cases is sometimes presented for context. For the general results, the metaheuristics are measured on a metric, called cost of execution This is the total average running time of an execution and combines the running times of successful and unsuccessful cases.

6.1 Results from testing the metaheuristic algorithms individually

The algorithms were tested with three different parameters on a dataset with Context-dependant Feature Models of sizes 60, 90, and 120, and with the traits displayed in the different rows in Table 5.2. For each combination of size and each set of traits, the tests were run on 100 non-void CFMs. This section will go through the results from running these tests and show some of the findings. These findings constitute the basis of the parameters later used in the main tests, where the algorithms are compared to one another.

6.1.1 Results from separate tests on Simulated Annealing

Simulated Annealing was tested with five different high levels of temperature to see if they have a significantly different impact on the algorithm. The results are presented in Table 6.1. The test here was conducted on Simulated Annealing set to run for a maximum of 40 times with 1000 iterations per executions.

The general findings here are that a high temperature may in many cases provide better results compared to lower temperatures. Additional

t	60	90	120	Avg
12	.700	.383	.243	**.442**
25	.617	.333	.203	.384
50	.603	.297	.173	.358
100	.523	.250	.153	.309
200	.463	.237	.140	.280
400	.433	.210	.120	.254

Table 6.2: Amount of successes when running SA with different levels of temperature and 1×50.000 iterations

tests with lower temperatures than 100, not shown here, support this. However, the higher the temperature is, the more is left to chance. What makes the algorithm successful is whether or not it finds a candidate that may ultimately lead to an optimal solution when the procedure has reached a point with low temperature. If the temperature is set too high, the algorithm may waste iterations in the initial phase on selecting neighbours more or less at random. The variations in the results between some of the higher temperatures in the table might be an indication of this when compared to the three lowest temperatures. Between temperatures 100, 150, and 200 there seems to be a steady increase in successfulness, while above 200 the results begin to vary without a clear pattern. Nevertheless, there seems to be significant evidence that Simulated Annealing, when run with 1000 iterations and 40 executions, performs better with a high temperature.

The second table, Table 6.2 shows performances of running SA with one execution and 50.000 iterations. Here the temperature spans over six different levels from 12 to 400. When compared to the results in Table 6.1, it becomes apparent that the first of the two settings outperforms the second, even though the second case have more iterations in total. From the results shown here, the test with 40 executions of 1000 iterations and 300 temperature has almost twice as many successes as the best result with one execution over 50.000 iterations.

Albeit, the growth in success towards lower temperature indicates that a lower t than 12 may be closer to the results shown in the first table. However, the growth does not appear to be progressive enough to be competitive. That is especially the case when considering the low results on large CFMs, A point worth mentioning, though, is that while the tests with few iterations seem to benefit from a large temperature, the findings in the second table indicates that a large level of iterations demands a low temperature.

m	**60**	**90**	**120**	**Avg**
0.01	**.798**	.721	.609	.710
0.03	**.800**	**.742**	**.634**	**.726**
0.06	.786	.560	.056	.467
m	**S1**	**S2**	**S3**	**Avg**
0.01	.694	**.736**	.699	.710
0.03	**.706**	**.737**	**.734**	**.726**
0.06	.451	.491	.458	.467

Table 6.3: Amount of successes from running GA with three different mutation probabilities

The results represented are combined from different sets of results where m is set to 0.01, 0.03, and 0.06, respectively. The other parameters were bp: set to 1, 5, and 12; $popSize$: set to 128, 256, and 512; and r: set to 0.08

6.1.2 Results from separate tests on Genetic Algorithm

Genetic Algorithm is the algorithm with the most parameters and possibilities of customisation. Because of all the possibilities, there are also pitfalls, and a setting that works well for one set of CFMs may work bad for another one, or a setting that works well in combination with a second setting may be ineffective when the second setting change. The focus here is therefore not to find the best settings in all situations but to locate settings that generally works well over different sets.

The tests were run with all combinations of the four parameters that are shown in Table 5.3. However, in the first run, the parameter r was set to 0.08. This parameter was tested separately with a subset of the parameter settings used in the first run. This section presents the results for each of the parameters.

Table 6.3 shows the amount of successes when running GA with three different mutation probabilities on the dataset. The results are first shown when the CFMs are grouped by their sizes. Below that, the results is displayed when the CFMs are grouped by their trait group. In both instances, the middle choice, 0.03, seems to be the better choice among the possibilities. The results in the table combine the results from the different values of bp with the three different population sizes. The introduction rate r was set to 0.08.

The different mutation probabilities all work fairly well for CFMs of size 60, as shown in the topmost of the two tables. However, as the CFM size increases to 60, the tests where m was set to 0.06 becomes considerably less successful compared to the other two mutation probabilities, and for size 120 this m performs quite bad. This is an example of a pitfall when a set of parameters that works well in one instance may perform poorly in another.

The second table shows the same results, but with the sets of CFMs

		Successes				Time (milliseconds)			
Trait group	*bp*	**60**	**90**	**120**	**Avg**	**60**	**90**	**120**	**Avg**
S1	1	**.840**	**.793**	**.663**	**.766**	1257	3266	7217	3914
	5	.797	.723	.603	.708	1284	3124	6405	3604
	12	.740	.643	.547	.643	1359	3441	7218	4006
S2	1	**.850**	**.790**	**.700**	**.780**	964	2069	4842	2625
	5	.790	.760	.690	.747	940	2289	4825	2685
	12	.737	.673	.640	.683	1051	2521	5666	3079
S3	1	.803	.770	**.627**	.733	991	3786	5561	3446
	5	**.833**	**.773**	**.627**	**.744**	1116	3100	5281	3166
	12	.813	.753	.610	.726	1233	2867	5352	3151
Combined	1	**.831**	**.784**	**.663**	**.760**	**1071**	3040	5873	3328
	5	.807	.752	.640	.733	1113	**2838**	**5504**	**3152**
	12	.763	.690	.599	.684	1214	2943	6079	3412

Table 6.4: Amount of successes and time consumption when running GA with three different levels of cross-over breakpoints

The results represented are combined from different sets of results where *bp* is set to 1, 5, and 12, respectively. The other parameters were *popSize*: set to 128, 256, and 512; *m*: set to 0.03; and *r*: set to 0.08

being grouped by their traits instead of their sizes. These results confirm that, based on the CFMs in the dataset used here, the middle mutation probability performs best overall. Both in group S1 and group S2, the low *m* of 0.01, comes quite close to the results when *m* is 0.03. This is not the case for group S3 which has a large amount of CTCs and a big amount of Alternative groups,

In general, the results here indicates that a mutation probability of 0.03 seems to work fairly well for different sets of CFMs. For the rest of this chapter, the results that are presented are from tests with *m* being 0.03.

The next parameter is the number of random breakpoints *bp* that is made during the cross-over procedure in Genetic Algorithm. Note that the positions of the breakpoints, and consequently the length of each partition, is decided randomly, while *bp* determines in how many partitions the candidates are to be split in order to create new and better candidates. In contrast to *m* and *r* which are parameters that increase or decreases the overall randomness and diversity of the population, *bp* is a parameter that is more related to the structure of the problem and should improve the general fitness with basis in the candidates already represented in the population.

To get an idea of how much the number of breakpoints impacts the results, the parameter was tested with a high number of breakpoints, a

r	60	90	120	Avg
0.01	**.820**	.760	.657	.746
0.04	.813	**.797**	.657	**.756**
0.08	.817	.780	**.663**	**.753**
0.12	**.823**	.773	.647	.748

Table 6.5: Amount of successes when running GA with four different levels of random introduction rate

The results represents tests where r is set to 0.01, 0.04, 0.08, and 0.12 respectively. The other parameters are: $m = 0.03$, $bp = 1$, and $popSize = 256$

mid-range and the minimum number, respectively 12, 5, and 1. Table 6.4 shows the performance of the algorithm with the CFMs grouped by size and by their trait group. In addition to showing the level of success, the table also shows how much time was spent by the algorithm to reach the optimal solutions.

The general tendency is that making 1 breakpoint provides better results than making 5 or 12. Considering group S1, the minimal number of breakpoints was 6 percentage points more likely to be successful. Also in group S2 the settings with 1 breakpoints provided more optimal solutions, although with less difference than in group S1. However, in group S3 and especially with smaller CFMs, the tests where bp was set to 5 did locate more optimal configurations than the tests with the lower bp did.

Another interesting finding was that setting bp to 5, generally provided optimal results slightly faster than the case was when bp was set to 1 or 12. Although, in group S2 the tests where bp was set to 1 and when it was set to 5 did, on average, require an equal amount of time. A reason for this could be that fewer steps are needed, when there are a bit more breakpoints, to shuffle the partitions around, ultimately resulting in better candidates in fewer steps.

Generally, the overall success is higher when the level of breakpoints is low. It might be a fair assessment that a higher bp parameter makes the algorithm more likely to overlook certain candidates, or candidate traits, and consequently more susceptible to falling into local optima. The reason for that could be because of how a candidate is structured with the root at the beginning and leafs towards the end. This is because the order in the configuration is based on a level-ordering of the CFM tree structure. A breakpoint will therefore cut through relations in several sub-trees. If the configuration was ordered based on a pre-order of the tree structure, crossover with more breakpoints might be more sufficient. With a pre-order, the partitions from a crossover would presumably preserve larger sub-trees.

The next parameter, the introduction rate r, is the portion of the population that is replaced with new randomly generated candidates for each generation. Based on the results previously described in this chapter,

	Successes				Time (milliseconds)			
pop size	**60**	**90**	**120**	**Avg**	**60**	**90**	**120**	**Avg**
128	.803	.743	.610	.719	521	1267	2878	**1555**
256	.817	.780	.663	.753	936	2348	5210	2832
512	.873	.830	.717	**.807**	1755	5505	9532	5597

Table 6.6: Amount of successes and time consumption when running GA with three different population sizes

The results represent tests where *popSize* was set to 128, 256, and 512. The other parameters are: $m = 0.03$, $bp = 1$, and $r = 0.08$

the parameter r was tested along with mutation probability of 0.03 and 1 cross-over breakpoint. Since r is relative to the population size, it was assumed that variations in the population size wouldn't have a great impact, so the test was only performed with the population size being 256. The different levels of r was 0.01, 0.04, 0.08 and 0.12.

The introduction rate is the parameter that applies the most randomness to the algorithm since it generates new candidates. That is perhaps why the results vary greatly between the different datasets, as shown in Table 6.5. The clue is to have a high enough r so that a sufficient amount of diversity is obtained, but not have it too high, so that too much is left to chance.

Based on the datasets used in this test, there is no strong pattern to what level of introduction rate is better. The set of CFMs of size 60 seems to be more or less unaffected by the change in the introduction rate, while the larger CFMs seem to slightly favour lower rates. In some specific cases, with the large CFMs, significantly worse results are obtained when the rate is set as high as 0.4, which indicates that it should be held reasonably low. For the main tests, this parameter was selected to be set to 0.08.

As previously mentioned for Hill-Climbing, the number of successes along with the average running time increases when the two parameters controlling times of execution and allowed plateau iterations are increased. Similarly for Genetic Algorithm, the larger the population size is, the better equipped the algorithm is to locate optimal solutions. As shown in Table 6.6, when the population size is doubled, the success rate increases by roughly 3-6 percentage points, when applied on the dataset used in this test. At the same time, the average time spent on locating a valid configuration is almost doubled. In addition, comes the increased time on unsuccessful executions of the algorithm.

It is worth noting that when the population size is increased, adjusting some of the other parameters might be necessary to get optimal results. The results here were obtained with a mutation probability set to 0.03, the random introduction was set to 0.08 and 1 breakpoint was set for the cross-over procedure.

	HC		SA			GA			
param:	*e*	*p*	*e*	*K*	*t*	*m*	*bp*	*r*	*popSize*
val:	5	10	10	1000	500	0.03	1	0.08	128

Table 6.7: The parameters selected for testing the metaheuristic algorithms on the main dataset

The parameter *e* means number of executions, *p* indicates allowed plateau iterations, *K* is the maximum number of iterations in SA, *t* is initial temperature, *m* is mutation probability, *bp* is number of breakpoints, *r* is the random introduction rate, and *popSize* is the size of the population.

6.2 Results from testing the metaheuristic algorithms together on the main dataset

The section above showed some of the challenges in selecting the right settings for Simulated Annealing and Genetic Algorithm. For the tests on the main dataset, a set of parameters were selected for the algorithms. The parameters that were selected were the ones that generally proved to work well in terms of successes and time performance. In this section, the results from these tests will be presented.

The individual tests were conducted on Context-dependant Feature Models, representing just a small subset of possible trait compositions and only small ranges of parameters were tested. Consequently, when comparing the metaheuristic algorithms to one another, and to HyVarRec, they are not with certainty configured optimally, even though they are configured so that they may work well in general, assuming that the CFMs in the initial dataset are fairly representative. The parameters selected for testing the metaheuristic algorithms on the main dataset are displayed in Table 6.7.

The CFMs in the main datasets represents 12 different sets of traits, as presented in Table 5.4, and three different sizes, 60, 90, and 150. For each combination of size and a set of traits, the three metaheuristic algorithms were run on 400 non-void CFMs. In total there were 36 datasets with 400 non-void CFMs in each.

This section will first present and discuss the general performance of the three metaheuristic algorithms. Then, it will take a closer look and compare results related to certain subsets of the main dataset. First the section will discuss the tests on CFMs with different levels of CTCs, VFs, and context size, as represented by trait sets A to F. Then, the tests on CFMs with varying levels of the four main relation types, as represented by trait sets G to L is discussed.

6.2.1 The general performance of the metaheuristic algorithms

This section presents the results that show the general performances of the three metaheuristic algorithms as well as the combinations of them. In all cases, the average results from the tests with CFMs of

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	.724	.930	.772	.963	.830	.968	.970
90	.558	.799	.634	.862	.677	.876	.880
150	.374	.514	.046	.604	.381	.526	.607
Avg.	.552	**.748**	.484	**.810**	.629	.790	**.819**

Table 6.8: Summary of successes in the main test

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	10	72	347	65	80	107	73
90	12	111	818	141	225	241	179
150	20	136	**2424**	319	78	214	334
Avg.	**14**	**107**	1196	**175**	128	187	195

Table 6.9: Summary of measured time for successful executions in the main test

In this and in the coming tables that presents time measurements, values represent milliseconds

different sizes are shown separately to indicate how the performance could be expected to develop when the CFMs sizes increase. The combinations of the metaheuristics are named, for instance, HC-SA for the execution of Hill-Climbing followed by Simulated Annealing. The column named All displays the performance of executing Hill-Climbing, Simulated Annealing, and Genetic Algorithm, in that order. The performance here refers to the amount of successes, the time measured in achieving successful reconfigurations, as well as how much time was demanded before termination when the execution wasn't successful. Each of these performance metrics are displayed in separate tables along with the cost of execution which combines the two time performance metrics.

Table 6.8 displays the average amount of successes achieved with the main dataset. The average score shows that among the three different metaheuristic algorithms, Simulated Annealing was the algorithm that achieved the most successes, with 74.8% of all the CFMs in the dataset being reconfigured. The average success rates also show that Hill-Climbing and Genetic Algorithm both were able to reconfigure roughly half of the CFMs, with HC achieving 55.2% successes, and GA achieving 48.4%.

Although HC seems to outperform GA on average GA did, in fact, reconfigure considerably more CFMs of sizes 60 and 90 than HC did. The low average score of GA is that, given the parameters set for this test, the algorithm's performance concerning the large group of CFMs was significantly low. This indicates that the levels set for the parameters of GA did not scale well to the largest CFMs in the set. One of the parameters that have previously been proven to have a potentially big impact on GA's

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	97	621	921	717	1018	1542	1639
90	192	752	1516	944	1708	2268	2460
150	591	1219	2627	1811	3219	3847	4438
Avg.	**293**	864	1688	**1157**	1981	2552	2845

Table 6.10: Summary of measured time in unsuccessful executions in the main test

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	34	111	478	89	240	153	119
90	91	240	1074	252	705	493	453
150	378	662	2618	909	2023	1934	1948
Avg.	**139**	**298**	1450	**362**	815	684	**675**

Table 6.11: Average cost of execution

abilities is the mutation probability. When set to 0.06, it caused the success rate to greatly incline when GA was applied to large CFMs, compared to lower mutation probabilities. It is possible that a mutation probability of 0.03 is too high when applying GA on CFM of size 150.

Another parameter that potentially could have increased GA's success rate, if adjusted, is the population size. However, as previously stated, an increase in population size will increase the time consumption accordingly. Table 6.9 shows how much time that was needed on average to obtain successful results. Time is measured in milliseconds. Here GA requires 2.4 seconds to locate valid results for CFMs of size 150, which indicates that increasing the population size would not be a sufficient alteration in order to increase the algorithm's overall performance.

In contrast to Genetic Algorithm, both Simulated Annealing and Hill-Climbing show quite good time performance with an average of 107 and 14 milliseconds for successful executions. However, the average time of successful executions only shows part of the cost of running a metaheuristic algorithm. In Table 6.10 the average running time of executions on CFMs that was not reconfigured is displayed.

When comparing the two different time metrics for HC and SA it becomes apparent that most valid configurations were obtained quite early during the execution. On average SA spent about 8 times as long on an unsuccessful execution as a successful one, while the same difference for HC is more than 20. This seems to indicate that there is a spectre of complexity for CFMs where some are easily reconfigured by metaheuristics, while other need significantly more time, and may often not be possible to reconfigure within a reasonable time. However, for most of the CFMs in the dataset, the metaheuristic algorithms were able to locate

valid configurations.

In Table 6.11 the two time metrics are combined to a total average cost of execution. This metric show the average running time when both successful and unsuccessful executions are considered. Compared to the running time for non-successful cases, these costs are generally quite low. This is especially the case for HC and SA, which can be attributed to SA's high success rate and HC's high time performance concerning successful executions.

In addition to their individual results, the algorithms were also combined in order to investigate the performance of running them sequentially. The success rates of these combinations, compared to the individual results show how large of an overlap there is between the algorithms' sets of reconfigured CFMs. These results can be viewed as a union of the two sets. For instance, the success rate of executing the combination of Hill-Climbing and Simulated Annealing (displayed as HC-SA) on CFMs, of size 60 shows that approximately 24% of the CFMs that HC could not reconfigure was solved by SA. In the other direction, the case was 3.3% improvement from SA's individual result. The benefit of applying HC and SA together is low, considering the group of small CFMs. However, on the larger models the union of the two performs considerably better than they do separately, with an average of 81% probability of success in 175 milliseconds. The total cost of 362 milliseconds is also fairly good, although the cases without success do require above the second to execute on average, and almost two seconds for the large CFMs.

Hill-Climbing and Genetic Algorithm both seem to benefit from being executed together, in terms of success rate. The success rate of combining Simulated Annealing and GA is also slightly improved by 4%, compared to the individual score of SA. On the other hand, the overlap between the set of CFMs solved by HC and SA together on one side and the ones solved by GA on the other is almost complete. In fact, the improvement obtained by adding GA to the sequence is on average less than 1%. This is apparent when comparing the amount of successes from running HC and SA together, 81%, with the results of running all three, 81,9%. Additionally, the cost increases by almost twice as much, on average.

Based on the parameters set for the three algorithms, and based on the main dataset, Simulated Annealing outperforms the other two metaheuristics on successes, and has a good time performance. The combination of Hill-Climbing and Simulated Annealing provide the best result in the least amount of time. In general there seem to be little to gain from applying Genetic Algorithm instead of, or in combination with Hill–Climbing and Simulated Annealing, as they cover most of the CFMs that GA does with a better time performance. However, these findings are based on results combined with tests done on several subsets of the dataset and does not exclude the possibility that Genetic Algorithm might be suitable in some specific cases. The next two parts of this section represents the results for specific subsets of the main dataset.

CFM size	CTCs	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	15	.749	.940	.796	.967	.857	.973	.975
	35	.671	.904	.726	.952	.778	.959	.960
90	15	**.553**	.833	.654	**.876**	.690	.889	**.892**
	35	**.511**	.744	.586	**.815**	.634	.833	**.839**
150	15	.384	.568	.072	.642	.397	.581	.643
	35	.331	.469	.029	.544	.335	.483	.548
Avg.	15	.562	**.780**	.507	.828	.648	.814	.837
	35	.504	**.706**	.447	.770	.582	.758	.782

Table 6.12: Results with different levels of CTCs

6.2.2 Performance on CFMs with different levels of CTCs, VFs, and context size

Table 6.12 shows the performances on CFMs with two different levels of Cross-tree-constraints. The CTC levels, 15% and 35% are relative to the size of the CFMs. The average results show that an increased number of CTCs has a significant impact on the metaheuristics' ability to locate solutions. The biggest impact is for Simulated Annealing. SA locates solutions for 78% percent of the CFMs with the low level of CTCs, and has a decrease by 7.4 percentage points, for the CFMs with the high level of CTCs.

The difference between SA's performance on the first and second group of CFMs is not as large for the small models, but it increases drastically as the size increases. For CFMs of size 150, the difference is close to 10 percentage points. This is not the case for HC and GA, where the difference develops differently. In fact, HC has a score on CFMs with 35% CTCs that is closer to the score of CFMs with fewer CTCs when the size is 90, compared to the case with CFMs of size 60. For GA the difference is somewhat similar for both the 60 group and the 90 group, with a difference of around 7 percentage points.

What this indicates is that all the three different algorithms are sensitive to an increased number of CTCs, but in different ways. Hill-Climbing seems to suffer most from a large number of CTCs on smaller models. For Genetic Algorithm, the impact of increasing the level of CTCs is not as influenced by the CFM's size. On the other hand, based on these results, Simulated Annealing is more sensitive to an increased number of CTCs, the larger the models are than the fact is for the other two metaheuristics.

When again, comparing HC with SA against the case of executing all three, it is clear that GA does not solve a great number of CFMs that was unsolved by HC-SA. For both CFMs with a maximum of 15% CTCs and with 35% CTCs, Genetic Algorithm increased the set of solutions by approximately 1 percentage point. However, when considering the mid-sized CFMs the contribution from GA is increased from 1.6% to 2.4% of the

CFM size	VFs, Csize	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	17, 13	.775	.926	.782	.961	.852	.964	.967
	33, 13	.698	.927	.761	.964	.817	.971	.973
	33, 27	.658	.913	.742	.954	.784	.963	.964
90	17, 13	.599	.802	.659	.870	.707	.880	.889
	33, 13	.534	.798	.624	.850	.660	.863	.864
	33, 27	.464	.768	.577	.817	.619	.842	.843
150	17, 13	.426	.543	.071	.631	.434	.564	.634
	33, 13	.353	.528	.044	.591	.362	.537	.593
	33, 27	.294	.484	.038	.558	.303	.496	.559
Avg.	17, 13	**.600**	**.757**	.504	.820	.664	.803	.830
	33, 13	**.528**	**.751**	.476	.802	.613	.790	.810
	33, 27	**.472**	**.722**	.452	.776	.569	.767	.789

Table 6.13: Results with different levels of VFs and context sizes

total number of CFMs, between the models with a low level of CTCs and the ones with a high level of CTCs. Even though GA generally is impacted by an increased number of CTCs, this could indicate that it might be a good assistance to HC and SA, on solving CFMs with a high level of CTCs. That would require a better set of parameters set for GA than was the case for large CFMs here.

Table 6.13 shows performances on CFMs grouped by their levels of VFs and size of the context. Both variables vary between two levels. The values presented here are relative to the sizes of the CFMs. These traits also impact the different metaheuristic algorithms in different ways.

When comparing the average results of HC and SA there is a significant difference in how the two algorithms seem to be impacted by an increase in the level of VFs. When the number of VFs increase from 17 to 33 percent, while the context size stays the same, the average success rate of HC is reduced by more than 7% of the dataset, while SA is almost unaffected. When the size of the context is increased proportionately to the increase in VFs the success rate of SA does decrease slightly. However not as much as the score for HC does. The average difference between the low levels of VFs and context size and the high levels is at almost 13 percentage points for HC, 3.5 percentage points for SA and about 5 percentage points for GA. The change in GA's performance, when the number of VFs increase while context size does not, lies somewhere between SA and HC.

These tendencies are reflected in the results displayed for the CFMs when grouped by their sizes as well. The great impact that the level of VFs and context size have on HC does not increase by a lot as the model increases in size. In fact, between the set of CFMs of size 90 and the set where size is 150, the difference is very similar. For SA the moderate

average decrease in success rate between CFMs with a low and a high level of VFs, when the context size stays the same, seems to stem from the set of large models.

A possible explanation for why SA and HC are impacted so differently may lie in what type of CFM comes from increasing the level of VFs. Similar to CTCs, by increasing the level of VFs the general complexity of the CFMs increases too and the number of valid configurations is likely to be reduced. However, in contrast to CTCs, the VFs are dependent on non-variable settings in the context, which the configurations have no impact on. So, while the complexity does increase, VFs allows for more room to make mistakes than CTCs does. This is because selecting a feature that is included in a CTC impacts the configuration of a different sub-tree. Considering that Hill-Climbing is much more vulnerable to being trapped in local optima, than Simulated Annealing is, these findings seems to indicate that VF's creates local optima.

The implementation of HC for this thesis is not instructed to prioritise fsatisfying some rules before others. If HC has a configuration that satisfies a large set of the rules, and a VF, which still is false, can only be made true by making the score worse, the HC have fallen into local optima. In contrast, if the VF was switched out with a CTC in this situation, the algorithm would have the option to change both sides of the implication and may be less likely to fall into local optima. This could be an explanation to why VFs seem to have a greater impact on HC than CTCs have. SA, on the other hand, is not as sensitive to local optima, which may be the reason why it is not as sensitive to the increased number of VFs.

Furthermore, the ratio between the number of VFs and the size of the context may also be a factor. It is possible that the more VFs there are for one context variable, the higher the possibility is for dead features, which could lead to dead sub-trees in certain instances of the context. Under this assumption, the results here are evidence that Simulated Annealing is better suited to handle dead features and sub-trees, induced by a specific context, than the two other algorithms.

6.2.3 Performance on CFMs with different levels of Mandatory, Optional, Or, and Alternative relations

The results on specific trait sets that has been discussed so far have been concerning relations that usually are added to a basic Feature Model tree structure, to add extra meaning. Such relations might be reduced if the wish is to remove some complexity from the model. Here the CFMs are grouped based on their levels of structural relations, such as Mandatory, Optional, Or, and Alternative. Since these relations are structural the total number of them can not be reduced without removing features. However, by adding CTCs, Feature Models, and consequently CFMs, can be restructured without losing their meaning, and by restructuring, the balance of the difference relations may change.

Here, the one-to-one relations Mandatory and Optional will be investigated first, to see what impact their levels may have on the performances

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	.733	.940	.765	.970	.820	.970	.973
90	**.600**	**.820**	**.633**	.895	.688	.900	.905
150	.365	.525	.030	.620	.373	.530	.623
Avg.	**.566**	.762	.476	.828	.627	.800	**.834**

Table 6.14: Summary of successes on CFMs with an even distribution of relation types

CFM size	Trait set	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	H	.670	.928	.715	.958	.768	.958	.960
	I	.850	.973	.950	.993	.973	**1.0**	1.0
90	H	.445	.680	**.523**	.750	.548	.763	.770
	I	.750	.913	.825	.973	.870	.985	.985
150	H	**.278**	**.400**	.033	.468	.285	.405	.468
	I	.615	.640	.068	.835	.623	.665	.840
Avg.	H	**.464**	.669	.424	.725	.534	.709	**.733**
	I	.738	.842	.614	.934	.822	.883	.942

Table 6.15: Comparison between successes on CFMs with a predominance of Mandatory relations and CFMs with a predominance of Optional relations

CFM size	Trait set	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	J	.723	.930	.765	.950	.818	.955	.955
	K	.780	.940	.793	.980	.883	.980	.983
90	J	**.610**	.785	**.668**	.875	.718	.890	.895
	K	**.638**	**.848**	**.703**	.920	.748	.945	.948
150	J	.400	.525	.038	.620	.403	.535	.623
	K	.415	.558	.065	.648	.418	.570	.650
Avg.	J	.578	.747	.490	.815	.646	.793	**.824**
	K	.611	.782	.520	.849	.683	.832	**.860**

Table 6.16: Comparison between successes on CFMs with a predominance of Alternative relations and CFMs with a predominance of Or relations

of the algorithms. Table 6.15 displays the success rates with the CFMs representing the trait groups H and I. These are, respectively, the group where Mandatory is set to cover 50% of the structural relations while Optional is set to 0, and the group where Optional is the equivalently dominant relation type. Table 6.14 will work as a reference point, as it represents the results from running the tests on the CFMs where all four structural relations are set to have equal coverage.

From the results, it is clear that Mandatory relations, more than Optional relation, makes a CFM more complex for metaheuristic algorithms. SA together with GA achieves a perfect score on the CFMs of size 60 when Optional is the predominant relation. On the other hand, the average success rate from applying all algorithms on the models in group H is lower than any of the corresponding results seen so far, in this chapter. However, it is rarely practical or even possible to create CFMs without the use of Mandatory relations, but it does illustrate how much of an impact the mandatory relations has on the metaheuristic algorithms' performances. One reason is that selecting the parent feature in an Or relation have no direct impact on the child, while in a Mandatory relation the selection of the parent implies the child, which may lead to further entailments down in the tree.

The algorithm that seems to be most vulnerable to CFMs with many Mandatory relations is HC, which has significantly fewer successes than the results in Table 6.14. SA and GA are not as affected for the small CFMs. The situation is different when the sizes increas. On the two groups with larger model sizes, all three show a similar vulnerability.

The balance between the two group-relations Or and Alternative have a slightly less impact on the results than the one-to-one relations, showed above. Table 6.17 displays the different performances on CFMs where Alternative relations cover 50% of the structural relations with Or relations covering none, set J, as well as the opposite case, set K. Compared to the control set in Table 6.14, the set of many Alternative relations show, in the case of all three algorithms applied, on average a low reduction in the success rate. The same result with a predominance of Or relations is increased by 2.6 percentage points.

These variations from the control set are most apparent with the groups of small and mid-sized CFMs. For CFMs of size 150, and partly CFMs of size 90, the results show a positive effect from both trait sets. This tendency is particularly the case for Hill-Climbing and Genetic Algorithm. On CFMs of size 90, HC performs better by respectively 1 and 3.8 percentage points when the Mandatory and the Or relations are given predominance, compared to the case of them covering an equal number of relations.

For CFMs of size 150, the increase in results is even greater. GA show the same tendency, while for Simulated Annealing, there is no improvement in increasing Alternative groups and reducing the level of Or groups. However, in the case of large models, SA performs equally well on CFMs with a large number of Alternative relations, as it does on CFMs in the control set.

While there seems to be a general improvement in performance on

CFM size	Trait set	HC	SA	GA	HC-SA	HC-GA	SA-GA	All
60	G-K	.751	.942	.798	.970	.852	.973	.974
	L	.665	.918	.705	.953	.793	.958	.965
90	G-K	.609	.809	.670	.883	.714	.897	.901
	L	.460	.808	.535	.855	.575	.858	.863
150	G-K	.415	.530	.047	.638	.420	.541	.641
	L	.268	.413	.013	.505	.270	.420	.505
Avg.	G-K	.591	.760	.505	.830	.662	.803	.839
	L	**.464**	**.713**	.418	.771	.546	.745	.778

Table 6.17: Comparison between successes on CFMs with different balance of group- and one-to-one relations

CFMs that have a predominance of one group relation type over the other, the performances decrease on CFMs with an overall increase in group relations. Table 6.17 show the difference in performance on CFMs. The CFMs are grouped by the case where one-to-one relation types and the group relation types cover 50% each, and trait set L, where the group relations cover 80% of the structural relations. Most vulnerable on CFMs with a large number of group relations are HC and GA. On average, HC solves almost 13 percentage points less CFMs with the high level of group relations, compared to the even level. In contrast, the same difference in SA's performances is below 5 percentage points.

The different structural relations have different levels of impact on the metaheuristic algorithms' performances. Most impact is made by Mandatory relations and the overall level of group-relations compared to the one-to-one relations. For large CFMs, the algorithms may perform better on CFMs where one of the two group relations is represented more than the other, compared to CFMs with an equal amount of Or and Alternative relations. Furthermore, HC and GA are more vulnerable than SA, on CFMs that have a structural relation composition that increases the complexity. However, a high proportion of Mandatory relations is a factor that reduces all three algorithms' performances greatly, in terms of success rate.

6.3 Comparison between the metaheuristic algorithms and HyVarRec

So far, the metaheuristic algorithms and different combinations of them have been compared to one another, based on their performances. This section compares their performances with HyVarRec (HVR). In contrast to the metaheuristics, HVR guarantees a solution. In comparing the metaheuristics against HVR their probability of success together with

	HC		SA		GA		HVR
CFM size	**time**	**succ.**	**time**	**succ.**	**time**	**succ.**	**time**
60	10	.724	**72**	**.930**	347	.772	1962
90	12	.558	111	.799	818	.634	2103
150	20	.374	136	.514	2424	.046	2640
Avg.	14	.552	107	.748	1196	.484	2235

Table 6.18: The performance of metaheuristics compared to HyVarRec

	HC-SA		HC-GA		SA-GA		All		HVR
CFM size	**time**	**succ.**	**time**	**succ.**	**time**	**succ.**	**time**	**succ.**	**time**
60	**65**	**.963**	80	.830	107	.968	73	.970	**1962**
90	141	.862	225	.677	241	.876	179	.880	2103
150	319	**.604**	78	.381	214	.526	334	.607	2640
Avg	175	.810	128	.629	187	.790	**195**	**.819**	**2235**

Table 6.19: The performance of metaheuristics combined compared to HyVarRec

execution time will be shown. Then the average cost of each metaheuristic approach is compared with the running time of HyVarRec. Cost is defined as the average total running time, combining the time of successful and unsuccessful cases. The algorithms are evaluated on their performance on the CFMs in the main dataset.

Table 6.18 displays the time needed by the metaheuristic algorithms to locate a solution, together with the probability that they will succeed compared to the average running time of HyVarRec. In Table 6.19 the corresponding results for the four different combinations of the algorithms are shown. The average results show that a combination of all the metaheuristic algorithms, based on all the CFMs in the main dataset, locates a solution within 195 milliseconds, with a probability of success at 81.9%. The corresponding running time for HyVarRec is at roughly 2.2 seconds.

Considering the executions of the metaheuristics separately, Simulated Annealing is the algorithm with the highest probability of success, and should, therefore, be considered as the isolated approach that is the most competitive in comparison with HyVarRec. On CFMs of size 60, the algorithm locates a solution in 72 milliseconds on average, with a probability of 93%. However, SA's probability of success on CFMs of size 150 is just slightly above 50%, which can not be considered a particularly competitive result.

The execution of different algorithms combined, shows more promising results in comparison to HVR. Considering the small CFMs again, by first running HC and then SA a solution is on average obtained within 63

CFM size	HC	SA	GA	HC-SA	HC-GA	SA-GA	All	HVR
60	34	111	478	89	240	153	119	1962
90	91	240	1074	252	705	493	453	2103
150	378	662	2618	909	2023	1934	**1948**	**2640**
Avg.	139	298	1450	362	815	684	675	2235

Table 6.20: The cost of executions compared to HyVarRec's running time

milliseconds with a probability of success at 96.3%. If Genetic Algorithm is used as well, the probability increases slightly, while the average time is similar to the case of just running SA. In contrast, HVR spends 1.96 seconds on average on the small CFMs.

Also on the mid-sized CFMs in the dataset, the combined executions perform with a fairly high probability of success. Compared to HVR's running time of 2.1 seconds, the combination of all three metaheuristics solved 88% of the CFMs in 179 milliseconds on average. However, these performance metrics does not take into account the running time of unsuccessful cases. Table 6.20 displays the expected cost of execution an approach, which combines the running time of successes and the running time of unsuccessful executions. These measurements show that the cost of running all three metaheuristics on the large CFMs is twice as high as the cost of running HC and SA. Furthermore, it is close to the running time of HyVarRec.

The expensive running time of Genetic Algorithm on large CFMs could be reduced. In the implementations of Hill-Climbing and Simulated Annealing in this thesis, a technique for calculating a configuration's score was used that greatly reduced their running time. The same technique was not feasible for Genetic Algorithm. However, alternative optimisation of GA, related to the scoring function or to other parts of it could be applied to enhance the running time.

Another reason for the high cost on large CFMs is the low success rate. It is quite likely that a different set of parameters would increase the algorithm's success rate, and that a better set of parameters would lead to successes in a shorter time. The cost of running all three metaheuristics on the mid-sized CFMs is still quite high, compared to running only HC and SA. On the other hand, In this case, the success rate is increased by a decent margin, and the cost of execution is still only a quarter of the running time of HVR. Genetic Algorithm should therefore not be fully dismissed as an assisting approach to running a combination of HC and SA, but it requires the correct set of parameters to function optimally.

Table 6.19 shows that, while still being quite a time efficient approach, the combined execution of HC and SA only succeeds with a probability of 60% for the CFMs of size 150. However, if they are allowed to run for longer, their general probability of success increases. To show this, HC and SA were run again with three different sets of parameters, on three groups

Alg.	Sizes	Success rate main	lvl 1	lvl 2	lvl 3	Time successes lvl 1	lvl 2	lvl 3	Cost lvl 1	lvl 2	lvl 3
	60	**.682**	**.764**	**.849**	**.879**	32	68	89	106	147	177
HC	90	.496	.527	.546	.575	52	81	143	392	595	805
	150	**.303**	.318	.320	**.328**	34	41	76	1545	2501	**3458**
	60	.924	.961	**.983**	**.985**	148	170	204	202	203	242
SA	90	**.742**	.813	.873	**.894**	304	433	453	629	732	**803**
	150	.459	.497	.534	.565	290	507	669	1639	2243	2805
HC	60	.961	.983	.996	.994	175	204	231	205	214	252
-SA	90	.807	.861	.905	.925	516	825	1026	835	1126	1360
	150	.539	.564	.598	.627	1186	2117	2884	2946	4435	5792

Table 6.21: The performance of HC, SA, and the two combined when allowed to run longer

These results were obtained by setting three different levels of allowed executions for Hill-Climbing and Simulated Annealing. In level 1 HC had 15 and SA 20 executions, in level 2 HC had 25 and SA 30, and in level 3 HC had 35 and SA 40 executions. For comparison, the success rates on these CFMs from the main test, where HC had 5 and SA 10 executions, are shown under main

of CFMs. The CFMs were selected from the two trait groups where the algorithms showed the lowest number of successes, H and F, as well as a group in the mid range, G. HC and SA were set to allow a greater number of executions in each test, with the level of executions being incremented by 10.

The success rates, running time and cost of each test is displayed in Table 6.21. The results show that allowing the two algorithms to run for longer on the group of small CFMs give a clear increase in the number of successes without gaining too much in cost. Especially HC show great improvement on the smaller models by an increase in successes of almost 20 percentage points on 35 executions compared to the results of the main test. Also on the mid-sized models, HC proves to be able to locate more solutions, however with a large increase in the cost. For the largest CFMs in the dataset, there is very little to gain from allowing repeating executions of HC. The few extra solutions that HC obtains by allowing 10 more executions is countered by a greater average cost, by one second. This indicates that there are CFMs which HC is unable to solve within a competitive running time.

SA also show a significant increase in successes without a high increase in cost for the small and mid-sized CFMs. However, as the success rate increases towards a full coverage, the up-scaling seems to provide a weaker increase in the results. This indicates that there might be a few CFMs in this dataset that SA is unable to reconfigure unless it is lucky. However, the general increase in results when the number of executions increase,

suggests that SA has the ability to locate configurations for hard CFMs if it is allowed to try again.

The average running time for SA on CFMs of size 150 does not increase as fast as the case was with HC and the increase in successes is steady for these models. This indicates that SA has the ability to solve more large CFMs when given more time. However, for complex CFMs, the cost might be too high compared to the probability of success, when matching the cost of execution against the running time of HyVarRec.

Chapter 7

Conclusion

Metaheuristics and particularly Hill-Climbing and Simulated Annealing are good and efficient algorithms for reconfiguring Context-dependant Feature Models. While Hill-Climbing and Simulated Annealing show most promise, Genetic Algorithm may be a good addition for some CFMs and have the potential to work better with different parameters than the ones used here.

In general, Hill-Climbing is a very low-cost algorithm that often provides solutions. On average the probability of success for CFMs of sizes 60, 90, and 150 is only slightly above 50%, but running the algorithm is quite cheap. In best case, Hill-Climbing locates a solution in a few milliseconds, while in worst case the algorithm terminates fairly quick with no solution. However, HC has a low probability of success on very large CFMs. In the results presented in this thesis, Hill-Climbing solved, on average 37.5% of the CFMs of size 150, which is almost half of the result it got on CFMs of size 60.

Of the three algorithms tested in this project SA generally performed better, considering the probability of success and average running time. The results show that SA has a significant vulnerability to CFMs with a high level of CTCs, especially for large CFMs. On the other hand, compared to the two other algorithms SA was very little affected by an increased number of VFs. In contrast, HC showed a significant decrease in success rate when the amount of VFs and the size of the context increased, but it was considerately less affected by an increased number of CTCs than SA was. Furthermore, even though, SA, in general, have a considerably higher probability of success than HC, there are a good number of cases where Simulated Annealing does not locate a solution, while Hill-Climbing does. Considering the generally low cost of running HC, the approach that provides the best results in addition to being efficient is to run both HC and SA, either in sequence, which was done in this project or in parallel.

Simulated Annealing, especially when combined with Hill-Climbing, is a time-efficient alternative for reconfiguring Context-dependant Feature Models. For small CFMs (with 90 features or less) this approach can locate a solution in less than 150 milliseconds with an average probability of success at 86%. For instances of size 60 or less these two metaheuristics can locate

a solution in 65 milliseconds with a probability of more than 96%.

Genetic Algorithm's performance in this project was generally poor. GA is an algorithm that requires more resources than the other two and shows significantly greater running time. From the results, it seems like GA is quite ineffective on large CFMs. This is probably more of an indication on the fact that the parameters of GA need adjusting according to the CFM, than it is of GA's general abilities. These results probably also affect the general running time of successful executions, since a better set of parameters might lead to the solutions appearing in earlier generations.

In some special cases, GA may find solutions to CFMs where SA and HC are unsuccessful, considering the parameters that were set for them in the main test. For instance when there is a high level of CTCs. The techniques for enhancing speed in HC and SA can not be directly implemented in GA. However, the implementation of GA used in this project might be improved on to reduce the running time. Furthermore, it must be further tested with different parameters to fully get an understanding of its optimal capabilities.

Compared to the CP-solver HyVarRec, Hill-Climbing and Simulated Annealing are in general able to locate configurations in a considerably shorter time but are more uncertain as they don't guarantee a solution. The larger the CFMs are, the more time is needed to obtain a high probability of success. The findings in this thesis indicate that there are CFMs that HC is not able to reconfigure, except by chance. This is particularly the case for large models. SA, on the other hand, seems to be able to find solutions, even for large CFMs, when given the opportunity to try again from a new starting point.

In future research, there are different adjustments that can be made in order to improve the performances of the three metaheuristic algorithms when reconfiguring CFMs. Since Hill-Climbing is more vulnerable to VFs than to other constraints, it is possible to design it so that it is instructed to prioritise configurations that fulfil VFs. For instance, in a subset of a neighbourhood where different candidates have the same probability of being selected, an addition could be added to the scoring of a candidate that favours candidates which fulfil a VF.

One of the reasons why GA might have been insufficient in this project could lie in the way the configurations are constructed. Since a configuration is a sequence of references to the features and attributes following a level-ordering of the CFM tree structure, the cross-over procedure is forced to split configurations across several constraints. Consequently, the left-hand side and right-hand side of these constraints end up in two different child configurations. If instead, the configuration followed a pre-order, the number of constraints that was split by crossover would be minimised, as more constraints would have been contained in the partitions created by the crossover. This improvement will perhaps allow, among other things for the Genetic Algorithm to work better with more crossover breakpoints. Furthermore, the parameters used for Genetic Algorithm in this thesis was sub-optimal for large CFMs. Further testing of the parameters in GA could provide insight into its ability to reconfigure

large Context-dependant Feature Models.

The tests in this thesis were conducted on sets of CFMs representing a range of sizes and traits. However, there are many possibilities of variations in CFMs. Further research on metaheuristic algorithms with different sets of CFMs could uncover weaknesses and strengths that were not covered here. Furthermore, there are several other metaheuristics, as well as variants of the three that was used in this project. This thesis tested three different metaheuristics and shows that a combination of Hill-Climbing and Simulated Annealing can be a good strategy for CFM reconfiguration.

Bibliography

[1] David Benavides, Pablo Trinidad and Antonio Ruiz-Cortés. 'Automated reasoning on feature models'. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2005, pp. 491–503.

[2] *BeTTy*. URL: http://www.isa.us.es/betty/ (visited on 09/05/2016).

[3] RW Eglese. 'Simulated annealing: a tool for operational research'. In: *European journal of operational research* 46.3 (1990), pp. 271–281.

[4] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[5] Bruce Hajek. 'Cooling Schedules for Optimal Annealing'. In: *Mathematics of Operations Research* 13.2 (1988), pp. 311–329. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689827.

[6] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[7] Roberto E Lopez-Herrejon et al. 'An assessment of search-based techniques for reverse engineering feature models'. In: *Journal of Systems and Software* 103 (2015), pp. 353–369.

[8] Roberto E Lopez-Herrejon et al. 'Evolutionary computation for software product line testing: an overview and open challenges'. In: *Computational Intelligence and Quantitative Software Engineering*. Springer, 2016, pp. 59–87.

[9] Sean Luke. *Essentials of Metaheuristics*. second. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/. Lulu, 2013.

[10] Jacopo Mauro. *HyVar-Rec*. https://github.com/HyVar/hyvar-rec. 2017.

[11] Jacopo Mauro et al. 'Context Aware Reconfiguration in Software Product Lines'. In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM. 2016, pp. 41–48.

[12] Francesca Rossi, Peter Van Beek and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[13] Sergio Segura et al. 'BeTTy: benchmarking and testing on the automated analysis of feature models'. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. ACM. 2012, pp. 63–71.

[14] Steven S. Skiena. 'Combinatorial Search and Heuristic Methods'. In: *The Algorithm Design Manual*. London: Springer London, 2008, pp. 230–272. ISBN: 978-1-84800-070-4. DOI: 10.1007/978-1-84800-070-4_7. URL: http://dx.doi.org/10.1007/978-1-84800-070-4_7.

[15] Stefan Voß. 'Encyclopedia of Optimization'. In: ed. by A. Christodoulos Floudas and M. Panos Pardalos. Boston, MA: Springer US, 2009. Chap. Metaheuristics Metaheuristics, pp. 2061–2075. ISBN: 978-0-387-74759-0. DOI: 10.1007/978-0-387-74759-0_367. URL: http://dx.doi.org/10.1007/978-0-387-74759-0_367.