University of Passau

Department of Informatics and Mathematics

Master's Thesis

A Comparison Study of Domain Constraint Solver for Model Counting

Author:

Adrian Marten

March 28, 2018

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Prof. Christian Lengauer, Ph.D.

Chair for Programming

Marten, Adrian:
A Comparison Study of Domain Constraint Solver for Model Counting
Master's Thesis, University of Passau, 2018.

Abstract

In the context of configurable systems, feature models are a common approach to manage and model commonalities and variabilities. Domain constraint solver can be used to explore different characteristics of feature models, e.g. #SAT. Considering that, there exist different approaches which can be used.
In this work we present a comparison of three constraint solver approaches. We will compare the different solver with artificial feature models to get a deeper insight into the behavior of the different solvers, therefore we vary the attributes of the feature models, these are the number of configuration options, the configuration option types, the feature tree depth and the number of cross tree constraints. As a proof, we will transfer our results to real configurable systems afterwards. At the end we conclude with concrete statements of the drawbacks for each domain constraint solver.

Acknowledgements

I would like to express my gratitude to my supervisors Christian Kaltenecker and Alexander Grebhahn for their comments, remarks and guidance to accomplish this master thesis.

Also, i want to address my thanks to Prof. Dr.-Ing. Sven Apel, for the opportunity and chance to contribute to his research, and also for the interesting topic of this thesis.

Finally, i thank my wife and daughter for their support, encouragement and endurance, all the years to reach this point.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Within the last years, the number of configurable systems increased, independent from industrial branches. By using configurable systems, commonalities of products are reused, together with a wide range of different varieties, to define a set of related products and reduce their time to market. Configurable systems can also be used to adjust products for different ethnic fields or geological regions. Popular examples of configurable systems are automobiles, chip sets and software families.
To manage all this variability on configurable systems, feature models are a widely used method [ABKS16]. Feature model encapsulate configuration options and constraints among them. By using this information, all possible products can be derived. There are different approaches for visual or textual purposes [Knü16] to represent feature models. Additionally, researchers found a huge set of different metrics to analyse feature models [BSRC10].
One evaluation metric for feature models is to count all valid configurations. The task of counting all possible valid configurations is also called model counting or #SAT. Whereas #SAT is not only performable on feature models, it can also be used on constraint satisfaction problems [Sch99, RNI95]. Therefore, researchers use domain constraint solver to perform #SAT on feature models [BSTRC06b]. These constraint solver exist for different domains of variables. The different domains for these variables are the types of values such a variable can represent.
In this thesis, we evaluate the three different constraint solver CSP, SAT and BDD by their ability to solve the #SAT problem. Additionally to the ability to solve the #SAT problem, we investigate how the three different solver iterate over certain number of configurations and return them.
To this end, we will run a set of tests to evaluate the behavior of different domain constraint solvers. Therefore, we generate artificial feature models to evaluate the influences of the different properties of feature models on the solvers. On this artificial feature models we perform sampling and #SAT operations to get a insight how the domain constraint solvers react on different problems. The insights will then be used to create hypothesises, which are then validated by models of real configurable systems.
Our results show, that for #SAT a BDD is more preferable than a CSP or SAT

solver. For sampling we will show that the CSP needs the most time to produce a valid set of configurations, in return it produces the sets of valid configurations with the broadest and variable use of configuration options. The remainder of the thesis is structured as follows: In Chapter 2 we introduce some background knowledge. In Chapter 3 we state our research questions and explain our test setup. In Chapter 4 we present our results. At last, in Chapter 5 we state some other works targeting on the usage of domain constraint solver within feature models.

2. Background

In this section, we introduce general terms used in this work: First, we give a brief overview on *configurable systems*, followed by a definition of *feature models* which are a way to describe and store the properties of configurable systems or visualize them. We will then illustrate the *constraint satisfaction problem* which applies to the generalized form of feature models and is an own class of mathematical problems. Last, we describe the three types of domain constraint solver we use in this work.

2.1 Configurable Systems

Every software able to adjust to user needs, with configuration files, roles, or even at compile time, can be called *configurable system*. These systems offer a set of *configuration options* which represent a functionality (feature) of a software or hardware system. The idea is that configuration options can be switched on or off, which is called binary configuration option. Other configuration options can even have a numerical value, maybe configuration options representing the pagesize of a configurable database system.
By adjusting the set of enabled and disabled configuration options, the resulting software or hardware system varies in its functionality. Such a set of assigned configuration options is therefore called *configuration*. The assignment of values to configuration options is not completely random and based on several constraints. These *constraints* describe relations among the configurations options and create a description of the properties of the configurable system.
Constraints can be described by following format "$< configurationOption_1 >< dependency > (< configurationOption_2 > | < value >)$" (example: "$co_1 = true$"), and also combined multiple times (example: "$(co_1 \vee co_2) \wedge \neg(co_1 \wedge co_2)$") to more complex expressions. A valid configuration is therefore an assignment of all configuration options such that all constraints are fulfilled. All possible valid configurations are the *whole population* containing every possible product/derivative of a configurable system. If we take one random configuration, we call this a *sample*. A set of samples are therefore called *sampling set*, which is a subset of the whole population. More detailed explanations of configurable systems can be found in Tartler et

al. [TLSSP11] and Liebig [Lie15], whereas both works point on the problems arising with configurable systems especially finding bugs in configurations.

2.2 Feature Models

Feature models (FMs) [KK02, HT08] are a visual or textual representation of configurable systems, including all configuration options and their constraints.
In general, a *feature model* uses the constraints modelling the parent-child relationship and represents them as a feature tree, as shown in Figure 2.1.
As shown in Figure 2.1, the *feature tree* defines the general dependencies between the configuration options.

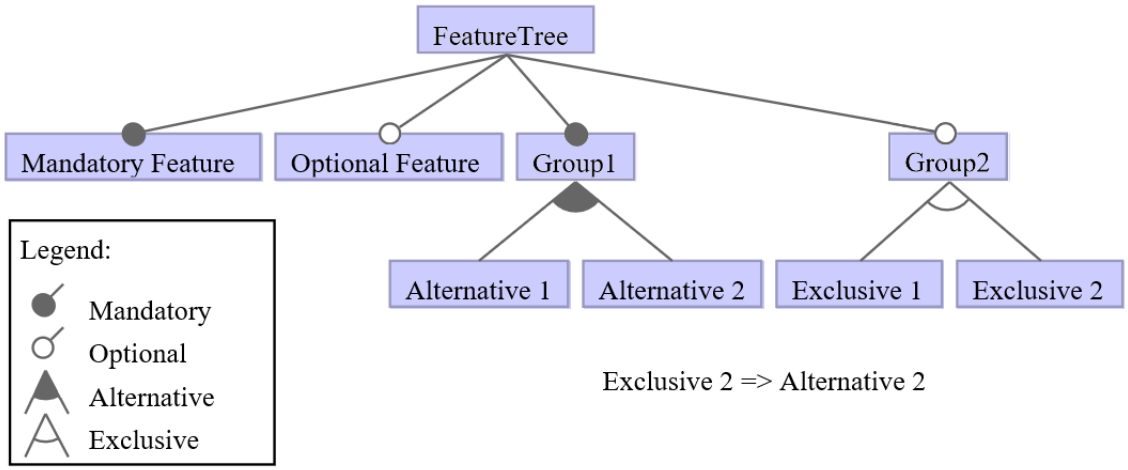


Figure 2.1: Example of a Feature Model, showing a feature tree with the 4 different types of configuration options with their parent-child relationship and a single cross tree constraint.

The parent-child relationships are typed with the following definitions:

- a configuration option is *Mandatory* if it must selected within all configurations
- a configuration option is *Optional* if the validity of a configuration does not depend on the fact whether the option is selected or not
- a configuration option is *Exclusive* if we have a parent as group where only be one child can be selected at a time
- a configuration option is *Alternative* if we have a parent as group where all children can be selected alone or combined within the other group members

The feature tree models the parent-child relationship, stacks up the configuration options on top of each other and groups them. Besides, there are complex constraints that can not be presented by the tree. These are shown in Figure 2.1 below the feature tree. This constraint is called *cross tree constraint* (CTC), which represents a constraint that can not be modelled inside the feature tree as part of the parent-child relationship.
While we showed one visual representation of a feature model, there exist a lot

```
c 1 Group1
c 2 Group2
c 3 Exclusive 1
c 4 FeatureTree
c 5 Alternative 2
c 6 Alternative 1
c 7 Optional Feature
c 8 Exclusive 2
c 9 Mandatory Feature
p cnf 9 15
4 0
1 -6 0
1 -5 0
6 5 -1 0
2 -3 0
2 -8 0
3 8 -2 0
-3 -8 0
4 -9 0
4 -7 0
4 -1 0
4 -2 0
9 -4 0
1 -4 0
-8 5 0
```

(a)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <feature_model name="FeatureIDE model">
        <feature_tree>
            :r FeatureTree(FeatureTree)
                :m Mandatory Feature(Mandatory Feature)
                :o Optional Feature(Optional Feature)
                :m Group1(Group1)
                    :g [1,*]
                        : Alternative 1(Alternative 1)
                        : Alternative 2(Alternative 2)
                :o Group2(Group2)
                    :g [1,1]
                        : Exclusive 1(Exclusive 1)
                        : Exclusive 2(Exclusive 2)
        </feature_tree>
        <constraints>
            C1:~Exclusive 2  or  Alternative 2
        </constraints>
    </feature_model>
```

(b)

Figure 2.2: The feature model from Figure 2.1 as textual representation in the formats DIMACS 2.2a and SXFM [MBC09] format.

more different visual and textual representations. Textual representations are DIMACS [Cha93] and Simple XML Feature Model(SXFM) [MBC09] for example.

As shown in Figure 2.2a, DIMACS is defined by a linebased structure. `c` denotes a comment which is used to place explanation texts inside the feature model. The comment is often used as a way to assign an ID to a configuration option name. `p` is the required problem line giving all needed information: the encoding, the number of configuration options, and the number of constraints used for this feature model. All lines after the problem line are the constraints. Depending on the encoding CNF [GK99, PSW10] (DNF) the ids are connected by an OR-relation (AND-relation) per line and the lines representing a AND-relation (OR-relation). Important is the '0' at each line ending, it is used as a line-ending symbol to ensure that the constraint are explicitly separated. This is also the reason why '0' is never used as an ID inside a DIMACS feature model. Important is that the feature model does not separate between the feature tree and the CTCs, all constraints are modelled in the same way.
In Figure 2.2b, we show the feature model using SXFM. Different from the DIMACS representation SXFM has a visual and textual separation between the feature tree and CTCs. Because XML is used as base structure, the feature tree is encoded directly inside the <feature_tree> tags and the CTCs are encoded inside the <constraints> tags. Besides, the parental dependencies are modelled by indents rebuilding the tree structure. Following tags are used inside the feature tree: **:r** the root configuration option, **:m** a mandatory configuration option, **:o** a optional configuration option and **:g** a group configuration option followed by a identifier for the type: **[1,*]** marks an alternative type for each configuration option within this group and **[1,1]** marks an exclusive type for each configuration option within this group. Each configuration option is written by its name followed by an ID-tag enclosed between two braces.
The three types of models already described are only a small overview on the differences among different representations of the one feature model. Knüppel [Knü16] presents an extensive comparison of different feature models with textual and visual types.

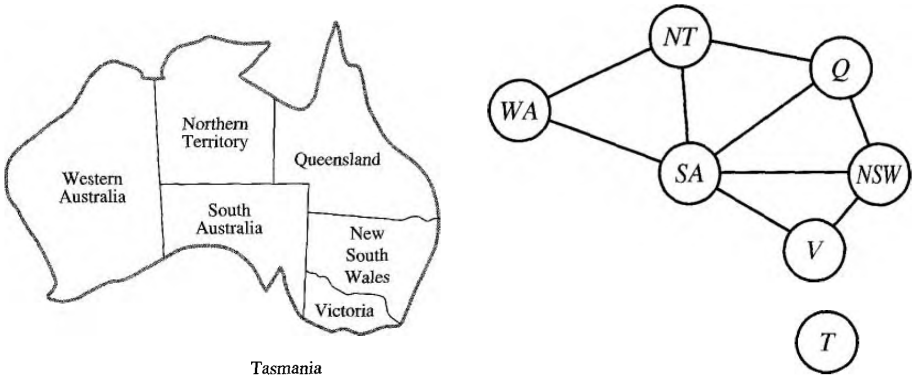
2.3 Constraint Satisfaction Problem

The Constraint Satisfaction Problem (CSP) is a general term of a class of problems in NP-hard. Real world problems, such as scheduling or resource management, can be seen or transformed into a CSP. Schöning [Sch99] shows that the complexity of the constraint satisfaction problems are generally NP-complete, if not NP-hard depending on the specific problem itself.
A CSP consists of a set of variables (Υ), domains of this variables (X), and a set of constraints (Ω).
The variables out of Υ are a description and placeholder for the assignment of descrete values. Whereas, the domain X of variables describes their set of valid values. In the following we use variables which have only the values true or false (0 or 1), also called boolean.
The constraints are constructed out of Υ and model the dependencies between the variables or restrictions on them. As an example, if we have a configuration option as described at Section 2.1 and say it shouldn't be true, the constraint is written as $v_1 \neq true$. If we have more configuration options and want to define a complexer constraint we can take '$(v_1 \vee v_2) \wedge \neg v_3$'. For a more detailed explanation we refer to, Russel et al. [RNI95].
A subclass of CSPs are the class of k-CNF problems. A CNF is a conjunction of constraints, whereas the constraints are a disjunction of variables. The k denotes the number of variables used at each disjunction. Russel et al. [RNI95] explains also how to transform a general formulated CSP into a k-CNF problem.
Whereas we use only boolean variables, we have to state that a k-CNF problem constructed only out of boolean variables is called a *boolean satisfiability problem* (SAT problem).
A very popular example for the CSP is the k-Coloring Problem [RNI95, DH98]. In the map-coloring problem, a certain number of different colors should be assigned to the nodes or edges of a graph without having two neighboring edges/nodes the same color. Figure 2.3 shows an example of such a CSP from Russel et al. [RNI95]. It shows the 3 coloring problem on the territories and principal states of Australia.

The varibles are the abbreviations of the territories and the domains defined on them are the 3 possible colors on them. The constraint graph of Figure 2.3b represents the constraints defined over the variables. This problem can then defined as all territories are out of the set of variables Υ = (WA, NT, Q, SA, NSW, V, T) and their domain X, which contains 3 different arbitrary colors. The nine edges inside the constraint graph shown in 2.3b formulate our set of constraints Ω, containing 9 pairwise negations one, for each edge inside the constraint graph.

2.4 Model Counting

Finding one valid configuration or proof a certain configuration as valid are not the only tasks in the field of CSPs. One interest is to find the number of all valid configurations for a given CSP. This task is known as *Model Counting* or *#SAT* [SBB+04, WS05, GSS06, GHSS07, GSS08]. We will use the term #SAT during this work, it defines the search of the count for all valid configurations within a given constraint satisfaction problem.



(a) (b)

Figure 2.3: The example for the 3-colouring-problem from Russel et al. [RNI95], taking the principal states and territories of Australia2.3a. The target is to colourize each territory with one of 3 colours, without having two connected territories the same color. 2.3b shows a constraint graph for this coloring problem.

While finding one configuration can be difficult for the constraint satisfaction problem, because CSP are based at the class of NP-complete problems [Sch99], counting of all possible configurations is even harder. For a more detailed explanation of the #SAT problem, we refer to Gomes et al. *Handbook of Satisfiability* [GSS08].
To solve the #SAT problem, there exists two different types of approaches: approximate methods and exact methods. Approximate methods (see the work of [CMV16]) uses heuristics to estimate the number of configurations within a short time. Exact methods instead focused on delivering an exact count of the number of configurations for a given CSP. At this work we focus on the exact methods for the #SAT problem.
As an example, *7z* the popular compression tool consists of 42 configuration options and 210 constraints encoded as CNF. This feature model leads to an exact count of 82368 possible configurations. Another example can be the *Berkeley Database in C* with 19 configuration options and 39 constraints, which results in 2560 possible configurations. These two feature models are small models related to the feature model of *buildroot* with 14910 configuration options and 45603 constraints or the popular feature model of the Linux 2.6.28 kernel, which consists of 6888 configuration options and 343944 constrains.

2.5 Domain Constraint Solver for the Constraint Satisfaction Problem

At this section we give an overview of different approaches to solve the constraint satisfaction problem. For this we roughly explain the ideas behind the solving process of the different domain constraint solver.
First we introduce the CSP-Solver, the most general solving approach. This is followed by two Boolean Satisfiability Solver the SAT-Solver and the BDD-Solver, both work on the same class of problem but use different strategies. For possible solver implementations have a look at Table A.1

2.5.1 Constraint Satisfaction Problem-Solver

As already mentioned a Constraint Satisfaction Problem-Solver (CSP-Solver) is the most general solution to solve CSPs, because it can solve CSPs independent from the variable domains. This makes the CSP-Solver practicable at the most possible problems.
It uses technics like Depth First Search, together with Backtracking [FW74, PW94, RNI95] and Forward Search [RNI95] to find configurations for a given CSP. Russel et al. [RNI95] describes the most popular techniques used by such CSP-Solvers.
However, the #SAT problem is not directly (within a single solving call) solvable by a CSP-Solver. To circumvent this problem, the CSP Solver implementation we use [PFL17], combines this techniques with a learning-approach. Listing 2.1 shows such a simple iterative learning approach.

Listing 2.1: Pseudocode of the learning loop back approach used by SAT and CSP solvers. Taking a solver which is connected to a CSP problem and returns the number of solutions found. To prevent the solver to find duplicates it adds the found solution to the set of original constraints.

```
1  input: solver
2  output: number of solutions
3  begin
4    count ← 0
5    while solver.hasSolution = true
6      singleSolution ← solver.getSolution
7      negated ← ¬singleSolution
8      solver.addClause ← negated
9      cound ← count + 1
10   end
11   return count
12 end
```

As shown in Listing 2.1 the solver starts by the initial problem solves it until no configuration can be found any more, and for each configuration we found we increase the counter for the found configurations. During the counting process it attaches the identified configurations of the CSP, negated to the constraints to ensure that the solver won't find the same configuration twice.

2.5.2 Boolean Satisfiability Solver

The Boolean Satisfiability Solver (SAT-Solver) [ES03, MMZ+01] is a more special Solver compared to the CSP-Solver. It can only solve problems defined over boolean constraints and thus, the solver is just able to use boolean variables. With the information from Section 2.3 we know, we are able to transform nearly all CSPs into a CNF which is represented only by boolean values. This enables the SAT-Solver also to work with CSPs containing variables beside the domain of boolean values.
Because the SAT-Solver is a more special and works on a subset of the CSP, the algorithms used are more special for the problems. The most popular algorithm is the David-Putnam (DP) algorithm. If the set of constraints Ω is empty the whole problem is *satisfiable* and if it contains an empty constraint it's *unsatisfiable*. The main procedure is done after the the check of the set Ω, by taking a variable out of Υ and assigns this value a truth value which satisfies this one. After this assignment the simplified formula is again used inside the DP algorithm. If it is *satisfiable* then

the DP algorithm returns *"satisfiable"*, otherwise the value from the last variable is set to the opposite value and DP is called again. A more detailed explanation is given by Mitchell et al. [MSL92] and Selman et al. [SLM+92] and implementations are done by Eén and Sörensson [ES03] and by Le Berre and Parrain [LBP10].
To solve the #SAT problem developers adapted their SAT-Solvers with a learning approach. This method uses the same iterable learning approach (see Listing 2.1) used by the CSP-Solver.

2.5.3 Binary Decision Diagram-Solver

Like the SAT-Solver the Binary Decision Diagram (BDD) [Bry86, Bry92, Som99, Jan06] works on SAT-Problems. Nevertheless, the BDD uses a completely different approach compared with the SAT-Solver and the CSP-Solver.
The BDD creates a Directed Acyclic Graph (DAG) [KU02, KB07], containing variables and constraints. After the BDD is completely generated the DAG of the BDD has to groups of terminal-nodes '1' satisfiable and '0' for not satisfiable. Also shown at Figure 2.4 the BDD uses two types of edges, dashed for a negative assignment of the parent node, straight line for a true assignment of the parent node.
However, for complex systems the DAG can become very complex. Therefore, exists two different specializations of the BDD, the Ordered-BDD (OBDD) and the Reduced-OBDD (ROBDD). Whereas the OBDD simply add an ordering to the variables the ROBDD is based on this order to reduce the number of nodes and edges within the BDD.
The strict ordering of nodes by their ID itself doesn't affect the performance or size of the BDD significantly. It ensures that each node n_i is followed by a node n_j within the parent-child relationship, if we have a strict order $n_i < n_j$.
The reduction is repeatedly performed by two rules, until both can't be applied any more.

1. Terminal nodes, or inner nodes having the same children, will be merged.
2. Inner nodes having the same children on both outgoing edges, will be removed. The parent of this inner node is then redirected to the children.

In Figure 2.4 we show a small example how the reduction affects the BDD. At 2.4a we see the first rule applied to the BDD, the two leaf nodes are merged and all inner nodes are just pointing to them. 2.4b shows the result of applying the second rule the first time. This merges the x_3 nodes together and removes the half of the edges from the terminal nodes to the x_3 nodes. 2.4c shows the result of continuing application of rule two on the BDD. At this stage we can't apply one of the two rules any more. As a result, it finished with five nodes left, out of fifthteen (be aware that 2.4a is the result after rule 1 was already applied), which is the fifth of the original size.
For more detailed information about the reduction and BDD manipulation, see Somenzi [Som99]. In the reduction nodes will be removed or merged, which deallocates memory and therefore reduce the complete memory consumption. Additionally it increases the performance, because we have to process less nodes during the configuration finding, and counting. Because of memory and performance constraints

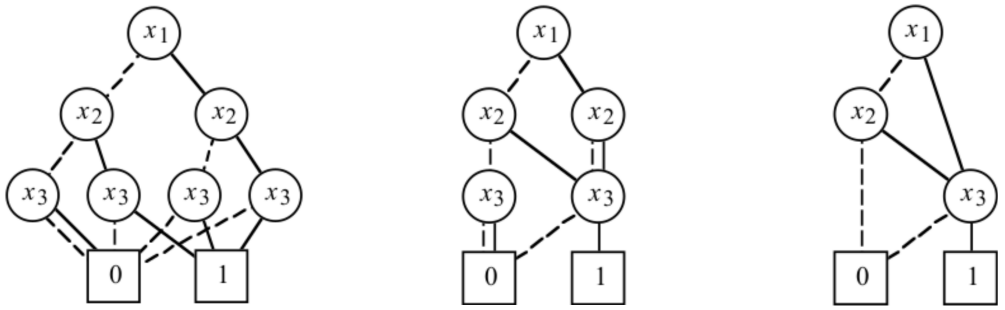


Figure 2.4: Example how reduction rules affect the BDD structure [Bry92]). Starting by eliminating duplicate terminal nodes 2.4a, followed by successively eliminating duplicate non-terminals 2.4b.
The most right picture shows the final BDD structure 2.4c.

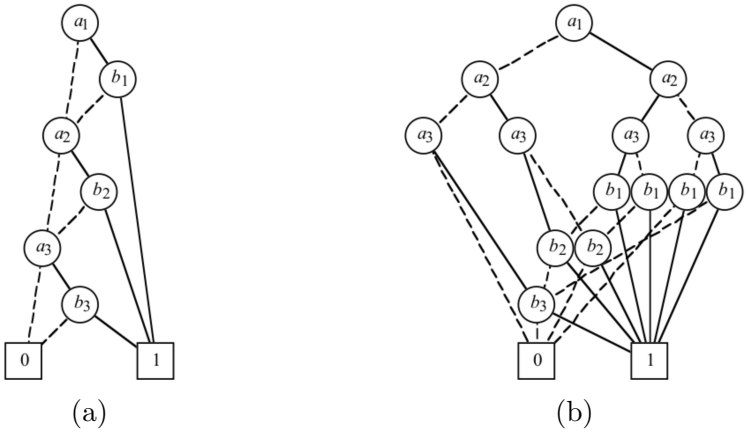


Figure 2.5: Example how variable ordering affects the BDD size [Bry92]. The example uses the same boolean expression: $(a_1 \wedge b_1 \vee a_2 \wedge b_2 \vee a_3 \wedge b_3)$ with a changing variable ordering. The effect is that we get a small and thin BDD (2.5a) for the variable order: $(a_1 < b_1 < a_2 < b_2 < a_3 < b_3)$ and a thick and puffy BDD (2.5b) for the variable order: $(a_1 < a_2 < a_3 < b_1 < b_2 < b_3)$, both with the same logic significance.

the ROBDD is the most commonly used BDD representation and in the following we denote ROBDD as BDD.
However, reduction itself is not every time effective like this. In Figure 2.5 we show an example how the variable ordering affects the reduction of a BDD. While both BDDs are generated out of the same boolean function, two different variable ordering are applied. In Figure 2.5a we can see the expected result of the BDD reduction. The original BDD with 127 nodes, including all inner nodes and leaf nodes is reduced to 8 final nodes. In Figure 2.5b we can see the same BDD with a different order of the configuration options. The result is a much broader BDD with 16 nodes left over, twice as big as the one of 2.5a.

Solving the #SAT problem with an BDD can be done by using the information hold by each node or edge of the BDD. The Figure 2.6 shows an example from Oh et al. [OBMS16], a BDD with the necessary information hold by the edges of the BDD.

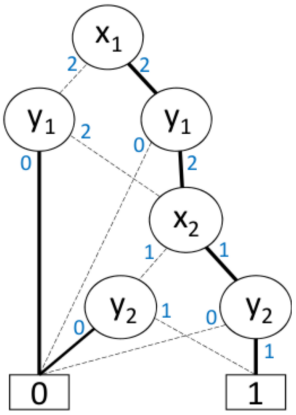


Figure 2.6: An example of an Counting BDD from Oh et al. [OBMS16]. The possible number of configurations starting by a certain node is attached as information to the edges.

A BDD fitted up with this counting information is called *Counting BDD* (CBDD). Let's say we start with x_1, using an false assignment, points us towards y_1. This can be described with a *configuration path* by using 0 for false and 1 for true, as common for boolean values. If we take the false assignment from x_1 to y_1 the path is written as (0,-,-,-). Whereas the '-' denotes the empty assignment for the 3 remaining assignments possible. Back to node y_1 we have the path (0,-,-,-) for the false assignment for x_1 and now summing up the counts of the outgoing edges from y_1 we know, we have 2 possible configurations left $->$ (0,0,0,0) and (0,0,1,1).

To add this counting information to the BDD a suitable way is to go buttom up from the '1' leaf node upwards to the root. In Figure 2.6 we can see the nodes for y_2, have one connection to the '1' leaf node, this assigns them a count 1 one for the possible configurations, the other edges not pointing to the '1' leaf node are directed to the '0' leaf node and get a configuration count of 0. Each parent not directly connected to the leaf nodes retrieves the possible configuration count from it's child nodes. Therefore each child gives the sum of it's own count up to the parent. This is done until we reach the root of the BDD. After fitting up the CBDD by this approach, we can compute at every node at the CBDD the number of remaining configurations, just by sum up the outgoing edges.

3. Experiments

At this section we introduce our research questions. For each research question we state their origin and intended target. After the research questions we summarize all experimental variables influencing our experiments together with the intended outcome. The last objective targeted by this section is a summary of used methodologies to setup our experiments.

3.1 Research Questions

To address the characteristics of the domain constraint solver, we set up four research questions (RQs). Each of them consider different behavior of domain constraint solver.
First, we target the solver creation behavior, followed by the behavior a solver shows for the #SAT problem. We end with the behavior of the different domain constraint solver types, while sampling snapshots of the whole configuration space by each solver. The sampling behavior is separated into runtime, memory and sampling set quality.

RQ1: What is the memory and time consumption to create a solver?

As explained at Section 2.5, the different solver types have different algorithms and memory structures. To create a solver instance we have to insert all variables and constraints into the solver object, ready to apply any algorithmic solving process, possible by the certain solver type.
Due to the differences in variable or constraint storage and the used algorithms it will be interesting to ascertain the differences during the solver creation, due to the DAG of the BDD solver type and the differences between the variable domains of the SAT and CSP solver.
By this question we want to investigate the differences between the different solver types. At the end we have information about the memory consumption of each solver and also how much time it takes to get it ready. Maybe this leads to informations

about possible pre optimizations inside a certain solver type.
Due to the construction of the DAG, we assume that the BDD solver needs more runtime and memory, compared to the CSP or SAT solver for the initialization. We assume not high differences comparing the CSP and SAT solver.

RQ2: How high is the memory and time consumption to solve the #SAT problem?

We showed different ways how the #SAT problem can be solved at Section 2.5. Whereas, the differences between CSP and SAT are smaller, the BDD approach is completely different. The question here is, if the CSP and SAT approach needs additional memory due to the loop-back approach. Does the BDD also request additional memory during the counting?
And most interesting is, how much differ the three approaches in runtime. How is the runtime behaviour for each solver type? And which approach takes the most time to solve the #SAT problem.
By answering this question we want to state which solver performes best for the #SAT problem. Additionally we also want to know if possible memory savings at the creation time (Section 3.1) are neutralized during the solving process of #SAT.
For #SAT we assume that the BDD has the best performance compared to the CSP or SAT solver, because of the information the BDD holds already at the DAG structure after the solver initialization. For the memory consumption we think, that the BDD uses only slightly more memory as used after the solver initialization. For the CSP and SAT solver, think of an increasing amount of memory during the solving process, due to the loop-back approach (Listing 2.1), but not more than the BDD uses in complete.

RQ3: How high is the memory and time consumption to sample a number of configurations?

This question is similar to RQ2, but it differs within the number of calculated configurations. The question by this task is, differ the solver types within their strategy for sampling? We know how the #SAT problem can be solved by a certain solver type, but can the same strategy used for sampling? Or do the different solver types handle sampling differently? Both questions can be pointed to the memory and time consumption as indicators for the solver behaviour.
The solver behavior should be nearly the same for the sampling of valid configurations as for #SAT, for the different solver types. The memory consumption could higher, because of the enumeration and storing of the configurations.

RQ4: How is the quality of a sampling set?

Due to RQ3 the performance of a domain constraint solver for sampling can not only measured by time and memory consumption. We can also measure the quality of a sampling set given by a solver. With quality we point on the similarity of the given sampling set in relation to the whole population. By this we defined to subtopics to answer RQ4:

- **RQ4.1** How much differs the number of configuration options per sample in relation to the whole population?
- **RQ4.2** Is each configuration option as often present as in the whole population?

With RQ4.1 we want to know how the ”outer” shape of a sample set is equal to the whole population. Together with RQ4.2, which can be seen as ”inner” shape of the sampled set we will be able to measure how similar the selection of configuration options is by a certain solver type.
Both properties can be useful to reduce the sampling sets if random sampling is used for research, like Oh et al. [OBMS16, OBMS17]

3.2 Experimental Dependencies

In this section, we define the empirical variables, we consider in our experiments. Firstly, we start with the independent variables related to the tough about from the already explained RQs. Resulting into the dependent variables, which are the derived values from the RQs. We end with the confounding factors and our solutions to negate them.
A complete list of all variables can be found at Table 3.1, together with their relation to the research questions.

	RQ1	RQ2	RQ3	RQ4
Independent Variables				
Feature Model	X	X	X	X
Domain Constraint Solver	X	X	X	X
Sampling Size			X	X
Dependent Variables				
Runtime	X	X	X	
Memory	X	X	X	
Number of configuration options per sample				X
Presence of configuration options within samples				X

Table 3.1: Table of the independent and dependent variables, with their relation to the four research questions.

3.2.1 Independent Variables

Domain Constraint Solver

As shown at Table 3.1 the domain constraint solver are also stated as independent variables. Whereas we want to compare them, it is necessary to classify them as independent variables. Each solver has it's own behaviour and structures as stated at Section 2.5. We choose three different solver implementations, whereas all three libraries are implemented in native Java®.

First the Sat4j [LBP10][1] library, which is the most prominent SAT-library for Java applications. Second the CHOCO [PFL17][2] library, which is a popular CSP-Solver library used at different commercial and non-commercial applications, at a wide range of different domains. Last, the JDD [Vah15][3] library, which implements the BDD and is also used by Jeho Oh [OBMS16, OBMS17] to find product line configurations with high performance or near-optimal configurations in product lines.
Most difficulty is the input structure each solver uses. This means each library uses its own input structure for the configuration options as well as the constraints. To unify the usage of each solver implementation we created an own implementation covering the method calls to make use of the solver implementation. Whereas this implementation is used to instantiate and use the three different solver implementations within the same methods in the same manner. Both parts helps to focus on the differences between the domain constraint solver.

Feature Model

At Section 2.2 we gave a short introduction to feature models an their different characteristics. Beside the textual and visual differences shown also by Knüppel [Knü16] there exists characteristics individual for each single feature model itself:

- The number of configuration options used at the feature model
- The number of constraints to define relations between the configuration options
- The ratio of configuration option types, which means the number of each type of configuration option in relation to the overall number of configuration options (see Section 2.2)
- The Branching Factor, which means the childs per node within the parent-child relation of the feature model, related to the maximum depth of the feature tree

All these differences comes into account while designing the experimental setup and predestinate the Feature Model as one of the independent variables.
During our experiments we used the DIMACS [Cha93] and SXFM [MBC09] feature models. To focus onto the individual characteristics of each used feature model, besides the textual or visual representations, we created a wrapper implementation covering the different input formats and unify it for each experiment.
The wrapper implementation uses as internal representation a mapping of ID and variable name given by the textual representation All constraints are stored by the instances of the variables or negations of them in a CNF format. We choose the CNF format because the SAT and BDD implementation uses it already for there constraint handling. This means each input has to be mapped into the CNF format, regardless of whether the model supports this or not. As an example the SXFM model is not directly written in CNF, and so we have to convert all expressions into the CNF format. The chosen internal representation is also used to transform the output given by any solver implementation into the correct variable name and id.

[1] http://www.sat4j.org/ (accessed on 2018/27/03)
[2] http://www.choco-solver.org/ (accessed on 2018/27/03)
[3] https://bitbucket.org/vahidi/jdd/wiki/Home (accessed on 2018/27/03)

Sampling Sizes

With research question three and four, we point on the behaviour of the domain constraint solver to return different sets of valid configurations. To get a better knowledge of the solver behavior we scale the size of the sampling sets in relation to the number of possible configurations, this leads to the sampling sizes as one of the independent variables. By scaling the Sampling sizes we get different runtime and memory consumptions by each scale used. Additionally we are able to investigate the solver sampling behavior due to the size of the sampled set of configurations.

3.2.2 Dependent Variables

Needed runtime

Because we tackle problems out of the class of NP-hard problems, the runtime is a direct indicator for effectiveness of the approaches used by the domain constraint solver. Additionally the runtime is directly measureable without any drawbacks for the benchmarks itself. So by this dependent variable we can directly answer the first part of the research questions 1-3.

Needed memory

To get a more comprehensive overview of the performance delivered by the domain constraint solvers, we also need the memory as a dependent variable. Together with the runtime, the memory is a factor how effective a constraint solver works on a given constraint problem. Therefore, we are able to directly answer the research question 1-3.

Number of configuration options per sample

With research question 4 we have the quality of sampling sets as a research point. Due to the experiments for research question 3 we will get the needed samples to investigate this aspect. By analysing the samples, we will be directly able to enumerate the amount of active configuration options per sample.

Presence of configuration options within samples

Together with the number of configuration options per sample we give the presence of configuration options within samples as one of two quality parts and sub research question. Together with the number of configuration options per samples, we will be directly able to enumerate the presence of configuration options within the samples.

3.2.3 Confounding Factors

For our experiments we identified severall confounding factors, which are listed below. During the section we will note them and state which actions we took to negotiate them.

Hardware and Software specifics

As the first confounding factor, we identified hardware specifics used to run the experiments as a set of different factors of confounding factors. Modern computers provide different efficiency improving mechanisms, like speed stepping, multi core setup and thread scheduling, which have a considerable influence on the performance. To negate or reduce the influences of this, we used for every test the same machine configuration and fixed the core speed. At the end we get a fix configuration with an Core i7 from Intel® and 16 GB of RAM.
Additionally, we reduced the number of physical cores and usable threads to one, to reduce influences of thread scheduling and core switching. In the end we also reduced the maximum amount of RAM up to 12GB, to avoid influences by OS depended memory usage like SWAP usage which leads us to the possibility and keep the whole test run inside the RAM.
As operating system we used the Linux™ distribution Ubuntu at version 16.04.

Java Virtual Machine

Second of the confounding factors, is the Java® Virtual Machine (JVM). Related to the usage of Java® as implementation language of the used libraries, we have to take the JVM into account. Benchmarking an application within the JVM can be difficult, even *simple* time measurement. While we need the mean of runtimes over different tests we want to have *Steady State Performance*. With steady state performance we think of normal execution performance, after the JVM made all the internal optimizations to speed up performance. This performance term also describes the performance for long lasting operations repeating the same task multiply. The term steady state performance is used, instead of the Start-Up performance which describes the runtime at the first execution of a Java®-Program, including optimization operations done by the JVM, to increase the performance. For further informations how to perform time benchmarks on the JVM, we refer the reader to the work of Georges et al. [GBE07]. Related to the work of Georges et al. [GBE07], we perform each benchmark 11 times in a row and ignore the first run during the evaluation. We ignore the first run of our benchmarks to exclude time measurements with additional time needed for the inbuilt JVM speed up methods.
For our tests we used a the JRE at version 1.8.0. Additionally the already mentioned implementation is compiled with a 64-bit JDK at version 1.8.0_152 and a Java® language level 8.

Wrapping Implementation

The third confounding factor is the implementation done to wrap the solver libraries and different model representations. While we implemented the application to just cover the responsible solver libraries without influencing the implementation itself, we can't state that the implementation has no influences for runtime and memory. To reduce the influences for the runtime measurement, we decided to implement an own setup. First choice we made was to measure just single method calls, especially the methods of the solver implementations. This should reduce the influences of our own application for the benchmarks. Additionally it enables us to measure the solver creation and problem solving of #SAT independent from each other, without

influences from reading the feature model or writing out the informations. Instead of using time tracing from the batch system which covers all steps included to solve #SAT or perform sampling, we have a better understanding which single task is responsible for the time needed by each solver. The part of possible increased memory consumption is not negated due to this implementation. Due to the dependent variable of needed memory consumption we can assume that the base amount of memory needed is related to the tasks for read in the feature model or write out the results. This is mostly non related to the solver task itself and therefore we can assume that the memory behavior of each solver can be successfully measured.

3.3 Methodology

3.3.1 Artificial Feature Models

As discussed in Section 3.2.3, the characteristics of the feature models have also an influence for our benchmarks. Therefore, we generated a large set of 40 characteristically different feature models of different feature sizes. For the generation we used the *SPLOT Feature Model Generator*[4].

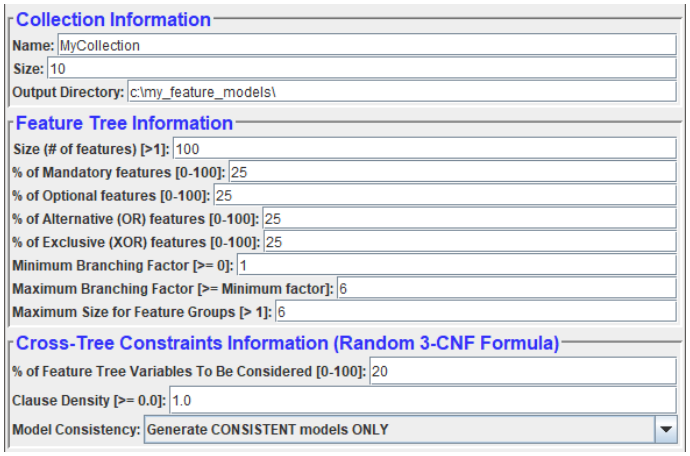


Figure 3.1: Configurations options of the SPLOT Feature Model Generator UI. The Collection Information setup for the saving location and number of models to generate. The feature tree setup to define the shape and properties of the feature tree, followed by the cross tree constraint properties.

As shown at Figure 3.1 the *SPLOT Feature Model Generator* offers advanced settings for the two parts of a feature model, the feature tree and the cross tree constraints. This offers us the possibility in individually studying the influence of the parameters on the runtime of the solver.
As baseline we created artificial feature models with 100% of mandatory features, because this configuration defines a feature model with the smallest whole population possible. The other types of configuration options are varied by 10% steps. All of die feature models have a parent-child relation only from the root node to the leaf nodes, this reduces the possible influences by different tree depths. To evaluate the influences by the feature tree depth we created three different types of feature

[4]http://52.32.1.180:8080/SPLOT/splot_fm_generator.html (accessed on 2018/27/03)

models by using a fix mandatory optional ratio, together with feature tree depths 1-2, 2-3 and 3-4. All of the generated feature models are suited without any cross tree constraints. To evaluate the influences of cross tree constraints we created also feature models with fixed type of configuration option ratio and a increasing number of cross tree constraints. All the different settings results in 40 different feature model configurations, generated for 20 and 30 configuration options each.

3.3.2 Real World Configurable Systems

To transfer the insights from the artificial feature models into to the real world, we use 7 feature models from real configurable systems. To compare this models with

ModelName	FeatureCount	Mandatory	Optional	Alternative	Exclusive	CTC's	Level
7z	42	2 (4,8%)	4 (9,5%)	0 (0,0%)	36 (85,7%)	0	1-2
BDBC	19	3 (15,8%)	7 (36,8%)	0 (0,0%)	9 (47,4%)	0	1-2
HIPACC	53	3 (5,7%)	2 (3,8%)	0 (0,0%)	48 (90,5%)	0	1-2
HSMGP	30	3 (10,0%)	0 (0,0%)	0 (0,0%)	27 (90,0%)	0	1-2
TriMesh	64	3 (4,7%)	1 (1,6%)	0 (0,0%)	60 (93,7%)	138	1-2
clasp	20	1 (1,0%)	5 (25,0%)	0 (0,0%)	14 (70,0%)	0	1-2
curl	14	3 (21,4%)	8 (57,2%)	0 (0,0%)	3 (21,4%)	0	1-2

Table 3.2: Table of the feature models from real world configurable systems, we use within the experiments. All models are listed with their properties used for the classification.

the artificial models, we list their attributes in Table 3.2. As shown in Table 3.2, the most models have a high number of exclusive configuration options, and only the *TriMesh* feature model contains cross tree constraints. In general we are faced with feature models with a maximum depth of 1-2.

7z[5], is an open source file archiver. The feature model we use focus on the *7z format* as part of *7z*, used for the compression of files.

Berkeley DB[6], is a database implementation provided with three different implementation languages C, Java and C++. The feature model we use is the C implementation, therefore we shorten it to *BDBC*.

HIPA^{cc}[7], is image processing framework using an own domain-specific language for high-level descriptions and transfers this coding into low level code for a wide variety of different GPUs.

HSMGP [KGKR13], is a scalable multi-grid solver. *HSMGP*, is used to test different data-structures and algorithms on high-performance computing systems.

TriMesh[8], is a library for the usage and manipulation of 3D triangle meshes.

clasp [GKNS07], is an answer set solver available under the MIT license, from the Potassco project for *Answer Set Programming*. It can be used as a solver approach for several domains (ASP, SAT, PB) or as library for own projects.

[5]https://www.7-zip.org/ (accessed on 2018/27/03)

[6]http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html (accessed on 2018/27/03)

[7]http://hipacc-lang.org/ (accessed on 2018/27/03)

[8]http://gfx.cs.princeton.edu/proj/trimesh2/ (accessed on 2018/27/03)

curl[9], is a open source command line tool and library for network data transfer based on different protocols.

3.3.3 Memory Measurements

At our research questions we mentioned the memory consumption as a performance criteria. Because the memory measurement within the JVM is not that easy, if we don't want to influence our runtimes. To circumvent influences directly within the test setup, we used a python script, called ps_mem[10] to trace the memory consumption of our test process.
Ps_mem can be attached to specific process id to log it's memory consumption. After the process is finished, ps_mem stops the recording.
By this approach we were able to measure the memory consumption without influencing the test process itself.
Due to the additional work by ps_mem we assume a slightly higher runtime for the test process. To reduce this influences we run the tests for the memory measurement seperatly. As a result we get exact to the second memory measurements.

3.3.4 Sampling Quality

At Section 3.1 we stated at research question 4 the quality of sampling sets. Just to recap this term, a sampling set is a subset of valid configurations, out of the whole population, defined by a feature model. At this section we give an explanation how we define this two quality factors for a given sampling set.

Cardinal Distribution

At Section 3.1 we stated research question 4.1 with the first quality measurement: *Number of configuration options per sample*. We also said this defines the "outer" shape of a sample set. The property defined by the number of activated configuration options per sample is also called the cardinality of a sample. If we take the question RQ4.1 we can reformulate it by the comparison of the cardinal distribution for a sampled set and the whole population.
As an example we placed Figure 3.2. Shown in Figure 3.2, is the cardinal distribution of an arbitrary feature model. The x-Axis shows the groups of the cardinal length. While the y-Axis shows the number of samples with this cardinal length. Both together defines the cardinal distribution.
The cardinal distribution will give a view on the rate of feature interactions within a sampling set. If the cardinal distribution has only small cardinalities, we can assume that the rate of feature interactions is small. Otherwise we have a higher grade of interacting features.
The term feature interaction is used to describe configuration options influencing each other while both are activated. The influences can therefore the performance or correctness of the operations inside the final software.

[9] https://curl.haxx.se/ (accessed on 2018/27/03)
[10] https://github.com/pixelb/ps_mem (last visited 2018/05/03)

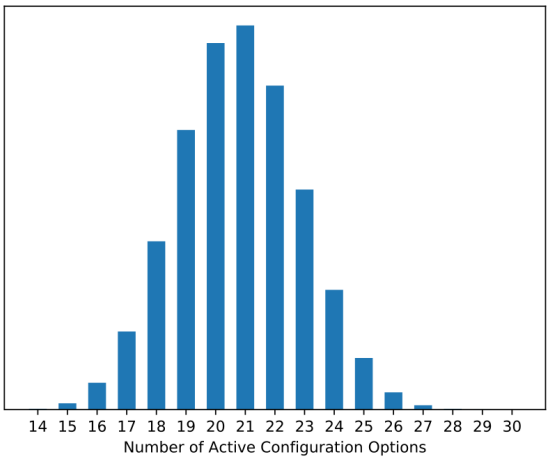


Figure 3.2: Example graph, for cardinal distribution. Each bar shows the number of valid configurations, for a specific number of selected configuration options.

Feature Frequency

The second quality measurement of a sampling set stated at Section 3.1 is the percentage, how often a configuration option is activated within the whole sampling set. We will call this property *Feature Frequency*. The feature frequency can be shown like the cardinal distribution at Figure 3.2. Whereas the x-Axis notates the configuration options ID and the y-Axis shows how often a configuration option is used inside a sampling set.
The difference here is we change the level of view from the raw number of activated features per sample to the overall usage per configuration option, within the whole sampling set. By evaluating the Feature Frequency of a sampling set, we are able to determine how good a solver varies at using the different configuration options within the sampling process.

4. Evaluation

In this section, we focus on two main topics: the performance of the domain constraint solver for #SAT and sampling, and on the quality of sampled subsets.
This section is separated into two steps. In the first step, we consider the performance by using artificial feature model to get a deeper insight into the behavior of the domain constraint solver. In the second step we use the results of the first part to define hypothesises on the performance of the domain constraint solvers on real world models. Last, we discuss the threats to validity that may affect the results of our experiments.

4.1 Experiments on Artificial Feature Models

In this section, we analyse the influence of the properties of feature models on the behavior of the domain constraint solver. For the analysis, we create multiple artificial feature models with different properties, such as the number of mandatory and optional, alternative or exclusive configuration options, the depth of the feature tree or the number of cross tree constraints. In result, we created a huge set of feature models with different properties. We only change one property at the same time, to evaluate the influence of single properties of the feature models.
This section is structured as follows: First, we present the runtime and the memory consumption of the solvers in the initialization. Afterwards, we show how the solver types perform to solve the #SAT problem. Last, we analyse the solver performance for sampling different sizes of sampling sets, followed by the sampling quality.

4.1.1 Solver Initialization

In this section we answer RQ1, the time and memory consumption for the initialization of the domain constraint solver. Therefore we use artificial models and modify their attributes to identify the influence of these attributes on the solver initialization. As stated for RQ1, we assume the highest runtime and memory consumption for the BDD solver, without high differences between the CSP and SAT solver.

Mandatory vs. Optional Ratio

First we start with the analyze of the artificial feature models, changing the number of mandatory and optional configuration options. Additionally we fixed the feature tree level and use no alternative and exclusive configuration options, without any cross tree constraints.
In Figure 4.1, we show the runtime for the solver initialization. Here, we see that all three domain constraint solver indicate a low runtime for the initialization. If we compare the trends from 20 and 30 configuration options in Figure 4.1, the runtime can be seen as nearly the same due to the millisecond runtime. Contrary to our assumption is the BDD not the one having the highest runtime, whereas we assumed the higher runtime because of constructing the reduced DAG. However, the CSP solver yields the highest runtime, which is a consequence of possible internal optimization and translation of the feature model during the initialization. The domain constraint solver with the lowest runtime is the SAT solver, which indicates not such a translation or optimization during the initialization.

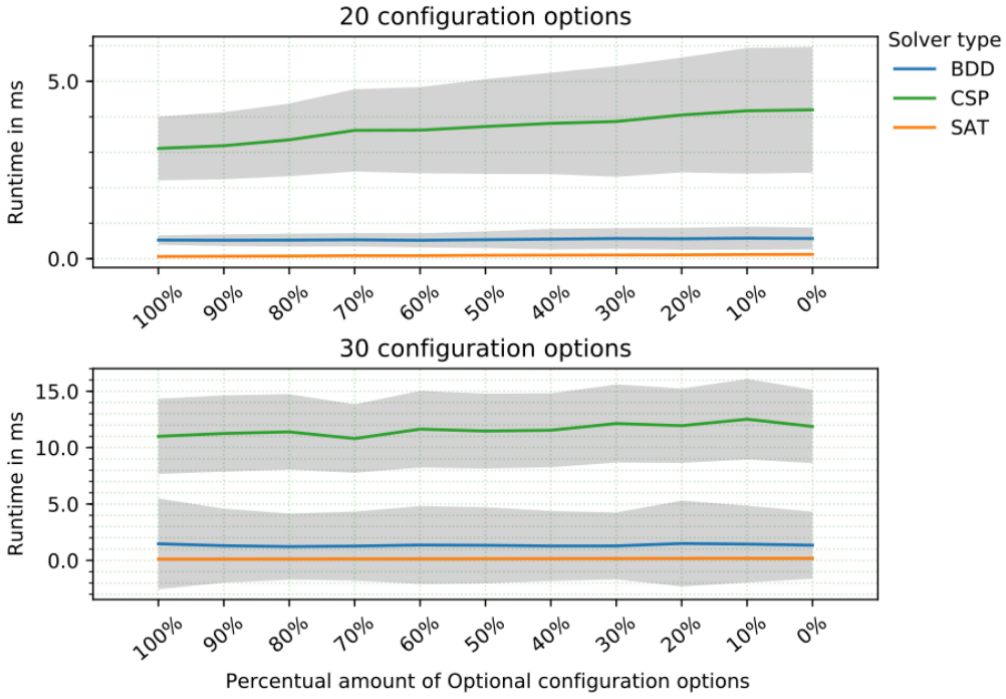


Figure 4.1: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for the initialization. The runtime is shown per solver, from 100% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

As shown in Figure 4.1, the runtime for the CSP solver increases with 10 additional configuration options. Also the BDD has a slightly increase, together with an increase of the surrounding variance shown in gray. This two parts indicates more work by the CSP solver for an increasing number of configuration options to optimize and translate within the initialization. Additionally we have an indicator for

the BDD, and the grown complexity for the internal DAG construction depending on the number of configuration options.
As shown in Figure 4.2, based on the shaded bar the memory for initialization is nearly the same as for all three domain constraint solver. In general, we have a maximal memory consumption of about 300 MB for all three solver types during the initialization. Also the number of mandatory and optional configuration options have no influence for this trend. Therefore, we can not state, that the BDD consumes more memory, compared to the CSP and SAT solver for a different number of optional configuration options.

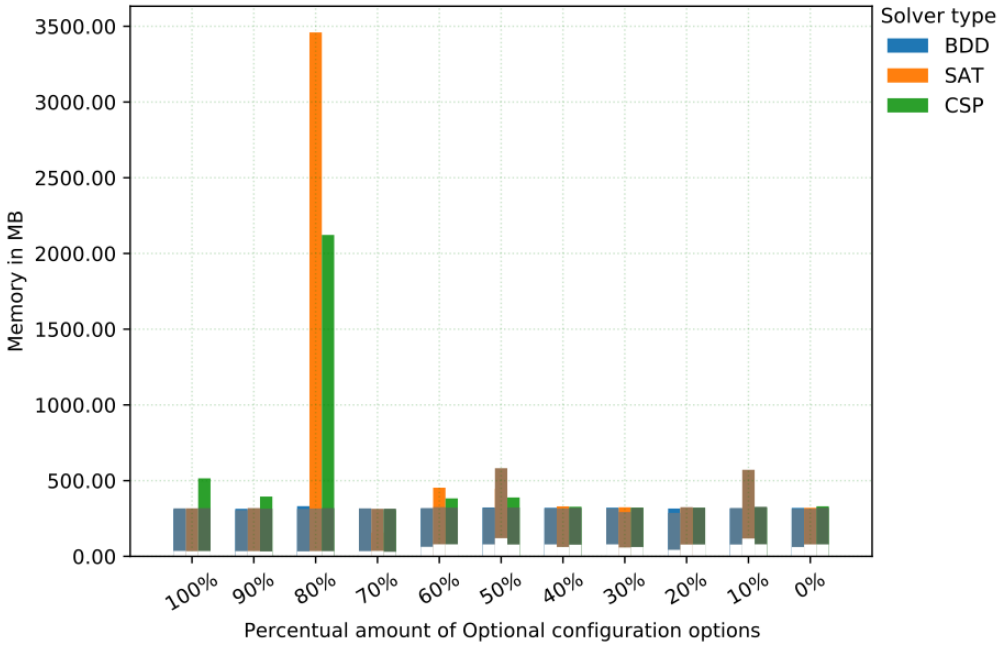


Figure 4.2: Plot of the memory consumption for 20 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 100% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

Mandatory vs. Alternative Ratio

The second set of feature models we use for our evaluation is equal to the setup of the mandatory and optional feature model set, but instead of modifying the ratio of mandatory and optional configuration options, we consider the influence of the percentual number of the mandatory and alternative configuration options.
Shown in Figure A.2, we have the runtime curves for the solver initialization with the alternative and mandatory configuration ratio. Compared to the runtime curves from Figure 4.1 we can see in that the relations created by the alternative configuration options, also have no great influence on the solver initialization. Again the CSP solver needs the most time compared to the SAT and BDD solver. However we see significant differences, in the lower variance (gray surroundings) of each solver

type, shown in Figure A.2, together with the flat runtime curve of the CSP solver. Whereas the increased number of configuration options leads not to a larger runtime, as shown in Figure A.2.
Again we can not state a significant trend for the memory consumption by any of the three solver types, using the information shown in Figure A.3 All three of them, use an equal amount of maximum memory during the solver initialization.

Mandatory vs. Exclusive Ratio

In the first set of experiments, we aim at identifying the influence of exclusive configuration options on the different domain constraint solver. To this end, we generate artificial feature models that differ in the percentual number of mandatory and exclusive configuration options. The rest of the attributes remains constant.
In Figure 4.3, we illustrate that the SAT solver behaves as before for the alternative and optional configuration options, which indicates no internal optimization of the configuration options and constraints. Also, the CSP solver shows the same trend

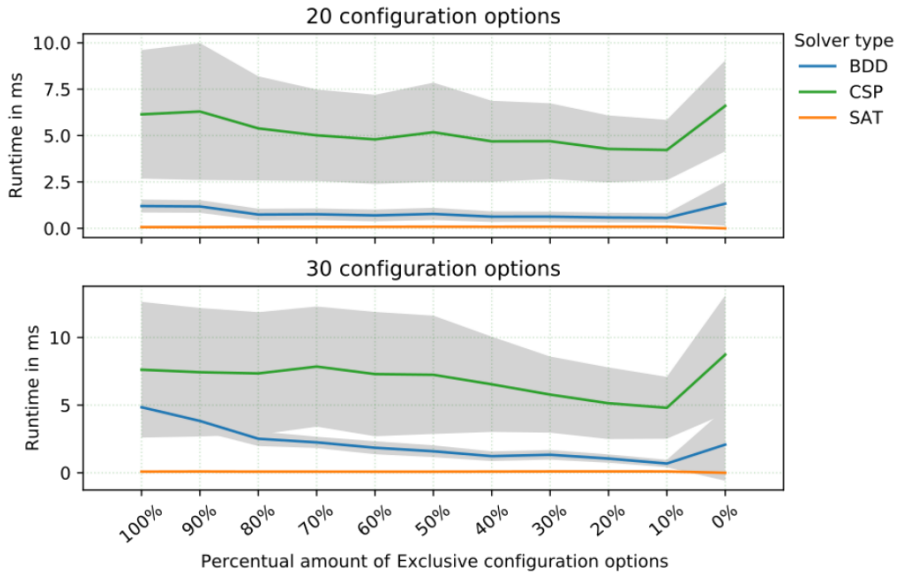


Figure 4.3: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for the initialization. The runtime is shown per solver, from 100% of exclusive configuration options down to 0% exclusive configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

as before for the changing number of alternative or optional configuration options. In contrast to the observations for the alternative and optional configuration options, we observe in Figure 4.3 an effect on the BDD solver with an growing number of exclusive configuration options. The runtime needed becomes smaller from left to right, which indicates a higher cost for the DAG, which is used in the BDD, to integrate exclusive configuration options. This behavior is best shown at the 0% mandatory and 100% exclusive ratio and shows also that an increase of 10 additional configuration options, increases the runtime for the initialization of the BDD solver.

If we compare the runtimes with the time needed for 100% mandatory configuration options the overall time is nearly the same as the runtimes for the optional runtimes from Figure 4.1. As shown in Figure A.5, we can see the memory consumption of the solver for the initialization. As before, we can not state a higher memory consumption for the BDD solver, or a concrete trend depending on the number of exclusive configuration options. Also, the CSP and Solver are not directly influenced by the number of exclusive configuration options.

Feature Tree Depth

Now, we consider the influence of the depth of the feature tree on the initialization of the different solver. Here, we generate a set of feature models, that differ in their maximum depth of the feature tree, while all of them provide 0% alternative or exclusive configuration options and 30% of mandatory and 70% of optional configuration options. At this set we fix the number to 30% mandatory and 70% optional configuration options, without any alternative or exclusive configuration options.
As shown in Figure 4.4, the runtimes for the solver initialization are unaffected by the feature tree depth, the variance of The runtime is smaller, than the standard deviation of the runtime for an single experiment. The only remark for the solver

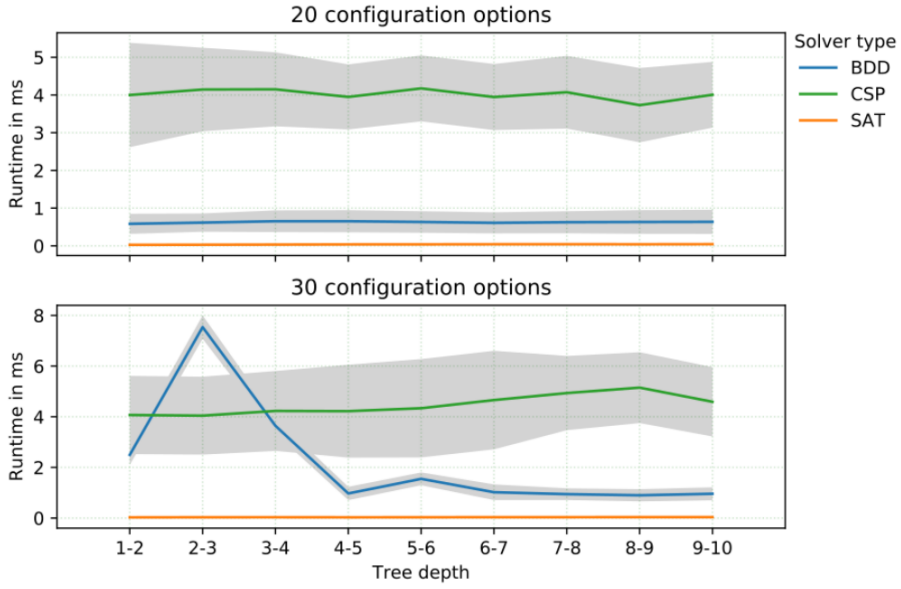


Figure 4.4: Two plots of runtimes needed for solver initialization. The runtimes is shown per solver, from depth 1 (one under root node) up to depth 10 (10 nodes under the root node). Each line of the solver runtimes are surrounded by the variance shown in gray.

initialization can stated for the BDD solver, for 30 configuration options. The BDD solver has an explosive increase in needed runtime for the depth 2-3, which flatten down until the depth 6-7. During multiple test runs, the result remains the same, whereas we are not able to state the reason of this behavior
The trend for the BDD of 30 configuration options shown in Figure 4.4, foreshadows that integrating a feature tree with a more deeper structure requires more runtime than a flat feature tree with all configuration options within one level. In contrast to

the runtime, we have no concrete influence of the feature tree depth for the memory consumption of all three solvers, as shown in Figure A.7.

Cross Tree Constraints

Next, we consider the influence of cross tree constraints on the runtime of the solvers. Here, we generate feature models with no alternative or exclusive configuration options and a flat structure, with all configuration options on the same level. For the experiments we fitted the feature models with a increasing number of cross tree constraints, therefore we reuse the feature models and add the new cross tree constraints. By this, the feature models with a higher number of cross tree constraint also contains the cross tree constraints from the feature models with a smaller number of cross tree constraints.
As shown in Figure A.9, the cross tree constraints have no influence for the time of the solver initialization. On the one side, the direct comparison between 20 and 30 configuration options, shows only a slightly increase of the needed runtime for the solver initialization. On the other side, the increasing number of cross tree constraints has no influence, on the runtime for the solver initialization of the three solvers. Like the evaluation before, the memory consumption shown in Figure A.10 shows no specific influence, of cross tree constraints and the needed memory of the three different solver types.

Summary

As result for the solver initialization, we have the CSP solver showing the highest runtime curves at all experiments compared to the BDD and SAT solver. Whereas, the BDD solver shows a significant increase for the initialization, when the feature model contains a high number of exclusive configuration options. Additionally, equal for each experiment is the smallest runtime of the SAT solver, which is not influenced by any property of the feature models we evaluate. In contrast to our statement for the memory consumption of the BDD solver from RQ1, we found no indicator which proves this. All of the solver types shows an equal amount of needed memory for the initialization.

4.1.2 Solving the #SAT problem

In this section, we answer RQ2 by evaluating the needed runtime for the different solver types, solving the #SAT problem. Therefore we use the same artificial models we also use to evaluate the runtime of the solver initialization.
For solving the #SAT problem, we assume that the BDD solver takes the fewest time of all domain constraint solver. For the BDD solver we assume only a small increase for the memory consumption, because all information for counting is already included in the DAG. Whereas, the SAT and CSP solver will take additional memory, because of the loop-back approach (see Listing 2.1).

Mandatory vs. Optional Ratio

In Figure 4.5, we show that the runtime needed by the three domain constraint solver to solve #SAT. One investigation we made, is the fact that the CSP and

SAT solver took a lot more time than scheduled. Because some runs have exceeded a maximum runtime of 4 days and a maximum memory of 12GB, these runs have been aborted. This leads to missing information from 80% optional configuration options up to 100% optional configuration options in Figure 4.5 and Figure A.12. We therefore added Figure A.13, which shows the reason of missing measures for the missing measures in Figure 4.5 and Figure A.12.
In Figure A.13, we highlighted the number of runs where the domain constraint solvers exceeded the memory limit (straight line) or exceeded the time limit (dashed line). As we illustrate in Figure A.13, the CSP solver exceeds the memory limit more often than the SAT solver. Whereas the SAT solver exceeds more often the time limit.
Beside of this, we can see that the SAT-Solver needs the most time of all of the three solvers (see Figure 4.5), followed by the CSP solver. Both the CSP and SAT solver

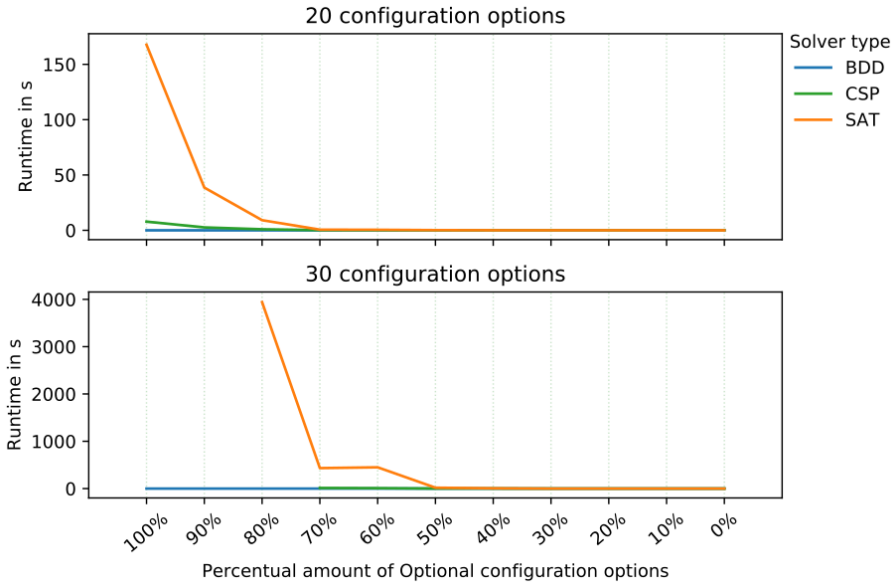


Figure 4.5: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for solving the #SAT problem. The runtime is shown per solver, from 100% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

shows a lower runtime for a lower number of optional configuration options. The reason for this behavior can only be the loop-back approach (Listing 2.1), used by the CSP and SAT solver for solving #SAT, and therefore the additional constraints took more time to validate.
In Figure 4.5 we show, the CSP solver needs less time than the SAT solver, which shows that the CSP solver takes actions during the loop-back to reduce the influence of the additional constraints. Because of the high runtime of the SAT solver, we provide in Figure A.12 the runtime for the BDD and the CSP. Again, increasing the number of configuration options from 20 to 30 provides similar results, but with an increase of the overall runtime for 30 configuration options. The time needed by the BDD-Solver is shown as a straight line, which shows that the percentile of optional

options doesn't affect the BDD for solving the #SAT problem. This observation shows, that the runtime of the BDD solver basically depends on the complexity of the DAG.
As shown in Figure 4.2, all three solvers have nearly the same memory consumption. We can not explain the inconsistencies, shown by the peak of the memory consumption for 80% of optional configuration options. Also performing the experiment multiple times, shows the same results. Taking this into account, we have no specific influence on the memory consumption for the three solver types. For 30 configuration options, we omit the memory measurements for the experiments with more than 60% of optional configuration options due to exceeding runs of the SAT and CSP solver, the result remains the same as for 20 configuration options, as shown in Figure A.1.

Mandatory vs. Alternative Ratio

The results shown in Figure A.14 and Figure A.15, are in line with the results when comparing the results from the mandatory and optional ratio (see Figure 4.5). The CSP and SAT solver have a lower runtime for a smaller number of alternative configuration options. The runtime curve shown can be explained by the loop-back approach the CSP and SAT solver uses to solve the #SAT problem. If we extend the size of configuration options to 30 we can see at Figure A.14 that this trend becomes stronger. Like for the experiments with mandatory and optional configuration options we had the problem that the SAT or CSP solver exceeds the runtime limit we set, or runs out of memory. Therefore, we add Figure A.16 to show the reason for missing measures.
The second match for the runtime trends is shown for the BDD solver. It appears that the BDD solver is not influenced by the number of alternative configuration options. Whereas this props the assumption that the most time the BDD uses is taken at the construction time and the counting itself is then solvable in linear time. As shown in Figure A.3, we have a higher memory consumption with an increasing number of alternative configuration options. The inconsistencies with loss of memory informations for the experiments with more than 70%, can not explained by us. Also, multiple runs of the experiments produce the same result. For 30 configuration options we omit the memory measurements for the experiments with more than 60% of optional configuration options due to exceeding runs of the SAT and CSP solver, nevertheless we have the same result as for 20 configuration options, shown in Figure A.4.

Mandatory vs. Exclusive Ratio

In Figure 4.6, we consider the influence of the number of exclusive configuration options on the time needed to solve #SAT. Here, we see a similar figure as we have already seen in Figure 4.5 and Figure A.14, for the experiments with optional and alternative configuration options. Again the BDD solver is the fastest to solve the #SAT problem, followed by the SAT solver. The runtime for the CSP solver is the highest. Comparing 20 and 30 configuration options (see Figure 4.6), this trend becomes stronger with an increasing number of configuration options.
Shown in Figure A.5, the SAT solver slightly uses more memory to answer #SAT

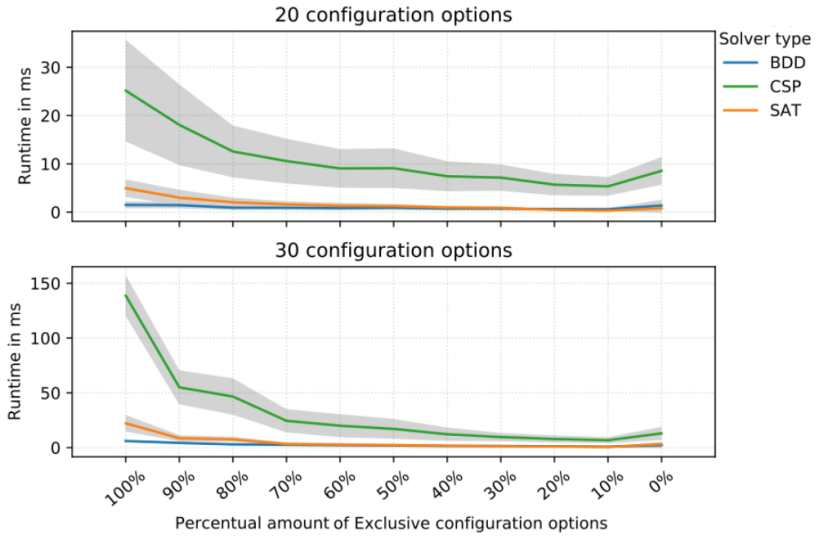


Figure 4.6: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for solving the #SAT problem. The runtime is shown per solver, from 100% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

compared to it's initial memory consumption. Instead, the CSP solver uses significant more memory for the test sets with 100%, 90% and 80% of exclusive configuration options. Together, with the increased runtime, this gives an indicator for a disadvantage of the CSP solver to process exclusive options in solving the #SAT problem. Like the SAT solver, uses the BDD solver no additional memory to solve the #SAT problem.

Feature Tree Depth

To compare the runtime curves of the solver, we show Figure 4.7. We can see, that the runtime of the SAT and CSP solver are influenced by the feature tree depth, other than for the BDD solver. Whereas The runtime is small for 20 configuration options, the runtime increase for 30 configuration options. As shown in Figure 4.7, the runtime becomes smaller with an increasing depth of the feature tree.

The decreasing runtime can only be explained with an decreasing number of valid configurations due to the deeper level of the feature tree. This means, if a parent configuration option is of type optional and not selected, the SAT and CSP solver can dismiss the rest of the feature tree under this configuration option for the solving process. This also explains the lower runtime, shown in Figure 4.7 for the CSP and SAT solver after level 2-3.

In contrast to the runtimes, we can see a higher amount of memory usage for the level 2-3 and the CSP and SAT solver, dropping towards the higher depths, as shown in Figure A.7. Together with the decreasing runtimes we can state that the JVM do not need to run the garbage collector within the test run and therefore we measured a higher memory consumptions.

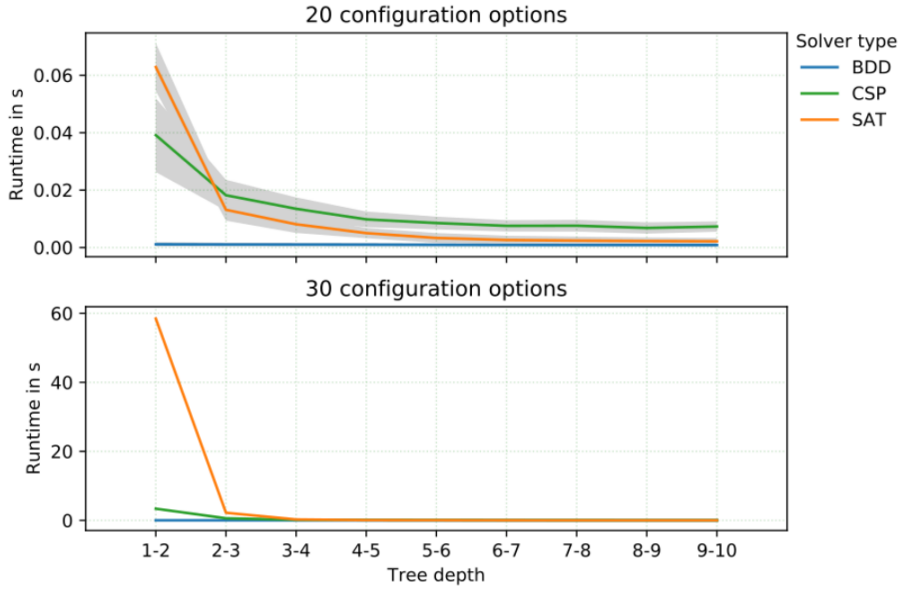


Figure 4.7: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for solving the #SAT problem. The runtime is shown per solver, for a increasing feature tree depth. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

Cross Tree Constraints

As the last experiment for solving #SAT, we have the comparison of the increasing number of cross tree constraints. Here we use the feature models, we also used for the solver initialization experiments. As shown in Figure A.17, we have the same behavior of all solvers, as before for the changing number of optional or alternative options. Therefore, the cross tree constrains do not have any influences on the runtime of the BDD solver. Also, the runtime of the CSP solver shows only small influences by the increasing number of cross tree constraints. Compared to the other to solver types, is the SAT solver the one which is the most influenced by the increasing number of cross tree constraints, this confirms the insight that the SAT solver heavily depends on the actual number of possible valid configurations for the #SAT performance, which decreases with a higher number of CTCs. As shown in Figure A.10, the memory consumption increases for the CSP and SAT while solving the #SAT problem. The BDD solver do not have a significant increase of the memory consumption for solving the #SAT problem. Except the drop for 4 cross tree constraints the memory consumption of the SAT solver decreases, with the increasing number of cross tree constraints. For 30 configuration options, shown in Figure A.11, we have a completely different behavior shown by the three solver types. This result remains the same for multiple runs of the experiments, therefore we can not state the reason for this inconsistency.

Summary

Taking the results in this section, we have the insight, that the CSP and SAT solver heavily depends on the total number of valid configurations defined by the

feature model, for solving the #SAT problem. Which in turn means, the more valid configuration defined by a feature model, the higher the runtime of the CSP and SAT solver to solve the #SAT problem. In contrast to this, the BDD solver always shows a constant runtime behavior within the variation of one property. This indicates that the runtime needed by the BDD for solving #SAT only depends on the complexity of the internal DAG structure.
For the memory consumption of solving the #SAT problem, we have a significant increase for the SAT and CSP solver together with the increased runtime. Whereas, the BDD solver only uses slightly more memory after the initialization, to solve the #SAT problem. Due to the inconsistencies, shown for the memory consumption of solving #SAT we are not able to state a concrete behavior of the different solver. Nevertheless, we are able to say, that the CSP and SAT solver have increased memory consumption for solving the #SAT problem after the initialization, depending on the number of valid configurations defined by the feature model.

4.1.3 Sampling

For the evaluation of the sampling behavior we took feature models with 70% of optional, alternative or exclusive configuration options and 30% of mandatory configuration options, a fixed feature tree depth of 1-2 and without any cross tree constraints. Additionally, we added a feature model with a feature tree depth of 2-3.
We select 1% up to 90% of the whole population of configurations valid for the feature model. As stated for RQ3, we assume a runtime like for #SAT, therefore the SAT solver should have the highest runtime for sampling. For the BDD, we assume that the runtime of the sampling scales linearly with the number of configurations to sample.

Runtimes & Memory

As shown in Figure 4.8 and Figure A.18, we see that selecting more configurations needs more time. Other than our assumption, the CSP solver needs the most time compared to the other solver types. In general, for all of the experiments, we see that the BDD is faster compared to the SAT solver. By this, we have a complete different behavior than for #SAT.
Like the runtime, the memory consumption of the three solver types are related to the number of sampled configurations. In all of our comparisons Section A.2.3, we see that the SAT solver needs less memory compared to the CSP solver. Additionally, the BDD is the solver using less memory than the other two solver types, this is in line with the results for solving the #SAT problem. If we compare the memory consumption from the #SAT solving and sampling, we have a significant increase in maximal memory consumption. This is related to the creation and storing of valid configurations. Therefore, the dropping amount of used memory in relation to the number of produced valid configuration can easily explained by storing the already generated configurations.

Cardinal Distribution

As showed in Figure 4.9 and Figure 4.10 the three solver types create configurations with cardinalities in line with the cardinal distribution defined by the whole population. Whereas the solver not really fit into the curve, the distributions shown in

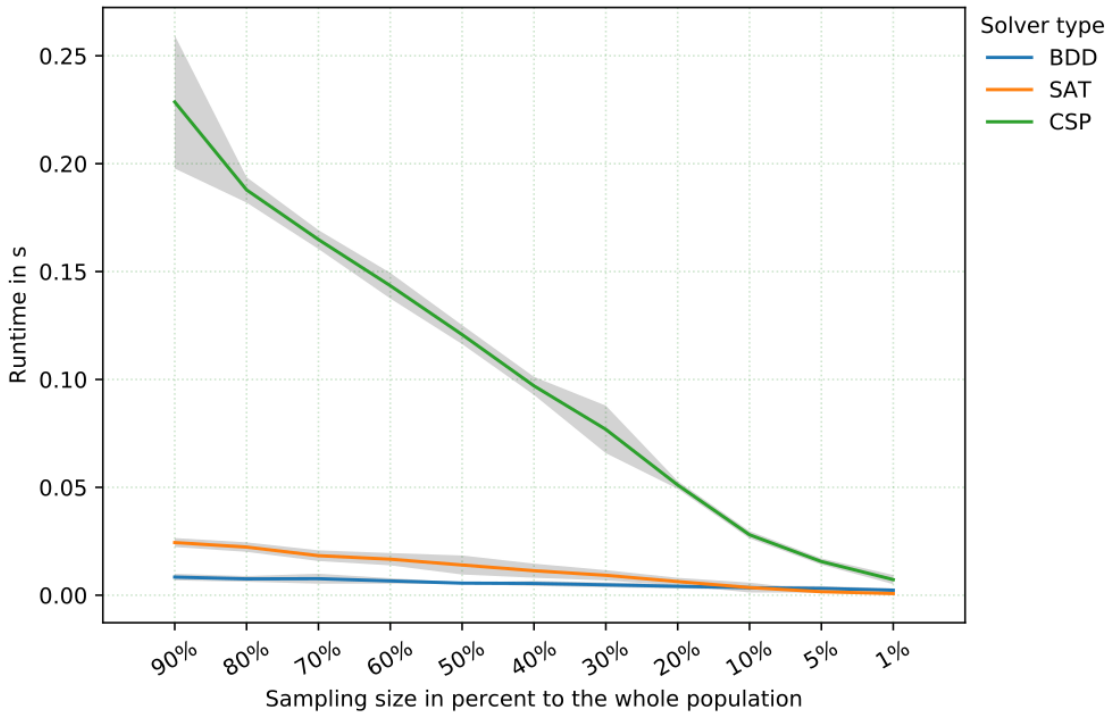


Figure 4.8: Runtime performance comparison, for sampling of configurations from the mandatory and optional artificial feature model, with 20 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

Figure 4.9 for 5% and 10% shows, that the solver already fullfil the correct distribution, but shifted to the left or the right which indicate a tendency to generally create configurations with less or more activated configuration options. The BDD and SAT solver are shifted more to the left, which indicates that both produce more valid configurations with only few activated configuration options. For the SAT solver we assume that this behavior depends on the David-Putnam algorithm, because it stops the solving process after a valid configuration is found by stepwise activating configuration options.
In contrast, the CSP solver generates more configurations with a larger number of activated configuration options.

Feature Frequency

As shown in Figure 4.11 and Figure 4.12, the three solver, hardly met the feature frequency defined by the whole population. As shown in Figure 4.11 for sampling 80% of the whole population, the solver nearly hit the baseline, other plots in Chapter A shows a nearly hit for sampling 20% or 40% of the whole population.
The best results in this experiment are achieved by the CSP solver, which shows the best variation for the usage of configuration options and catches the most different configuration options within the sampling process. Followed by the BDD solver, which shows a also wide variation of his configuration option usage, but this looks

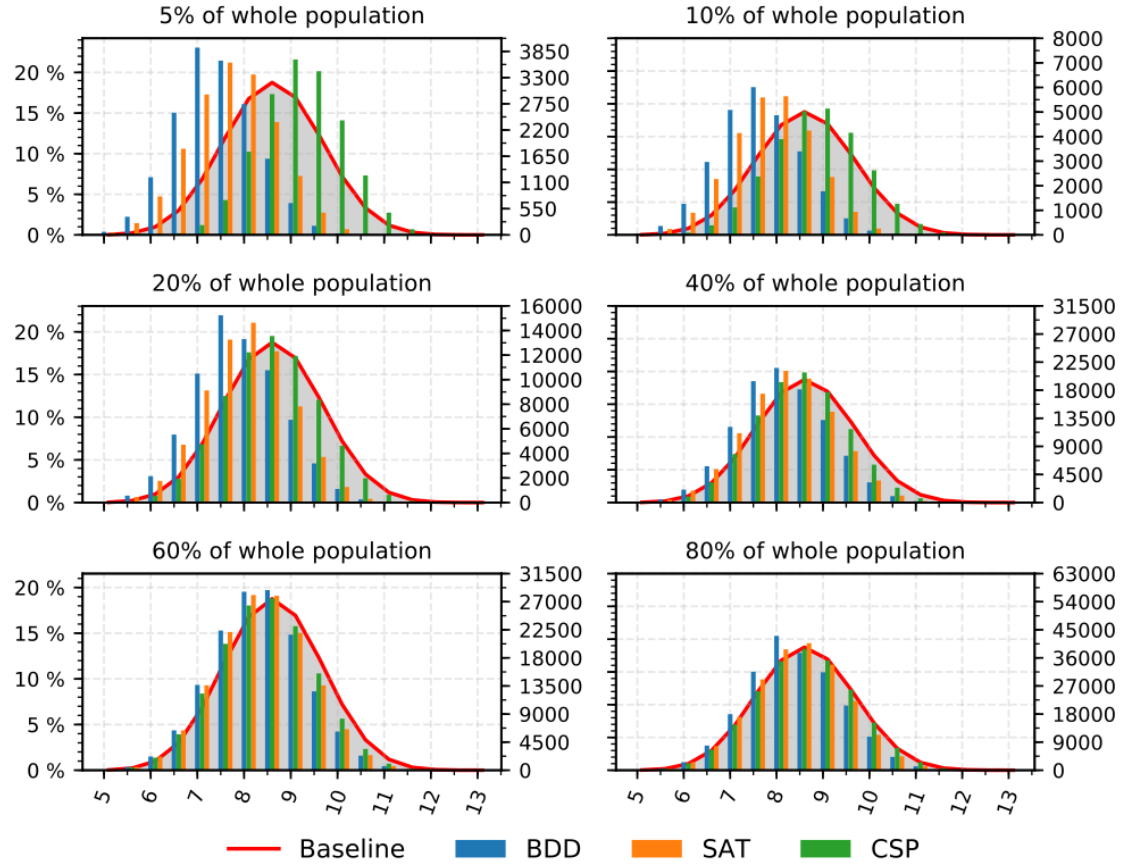


Figure 4.9: Comparison of the cardinal distribution, for the sampling results of the mandatory and alternative feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

like it depends on the overall number of possible valid configurations, defined by the whole population. Is the total number of valid configurations defined by the whole population smaller, it can't hold up to the CSP. In the end, the SAT solver shows the most localized usage of configuration options of all the three solvers. Even for higher numbers of sampling sets, it has higher usage counts for configuration options related to each other, defined by the order given from the feature model used.
As shown in Figure 4.12, all solvers have a problem with the alternative options. As they try to use a each feature as often as possible, this behavior goes completely different to the baseline defined by the whole population.

Summary

As summary for sampling a certain number of valid configurations, we can state that the general runtime needed by each solver depends on the number of configurations itself. Whereas the BDD solver needs less time than the CSP and SAT solver to create a equal number of configurations. In contrast to the runtime needed for

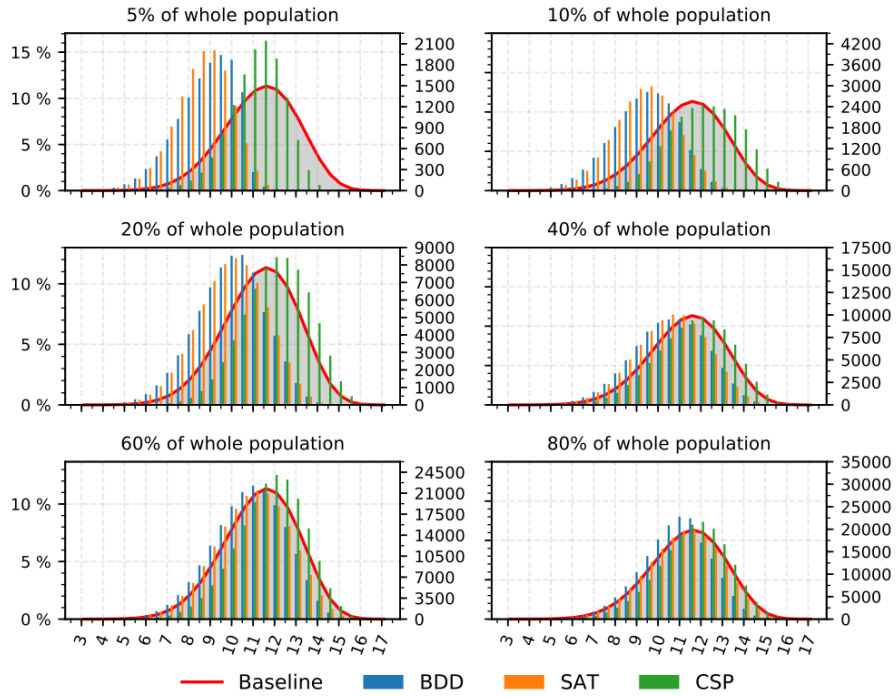


Figure 4.10: Comparison of the cardinal distribution, for the sampling results of the tree depth feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

solving the #SAT problem, the CSP solver is the one with the highest runtime, instead of the SAT solver. Additionally to the runtime, the CSP solver is the one consuming the most memory of all three solver, followed by the SAT solver.
Comparing the quality of the sampled sets of the three solver types, the CSP solver is the one showing the widest usage of configuration options during the sampling process. Additionally, the CSP solver has a tendency to select more configuration options to create a single configuration. In contrast to this, both the BDD and SAT solver show the behavior to select few configuration options within the sampling process. This comes together, with a localized usage of configurations options by the SAT solver. With localized we mean the order of the configuration options, defined by the input order of the feature model during the initialization.

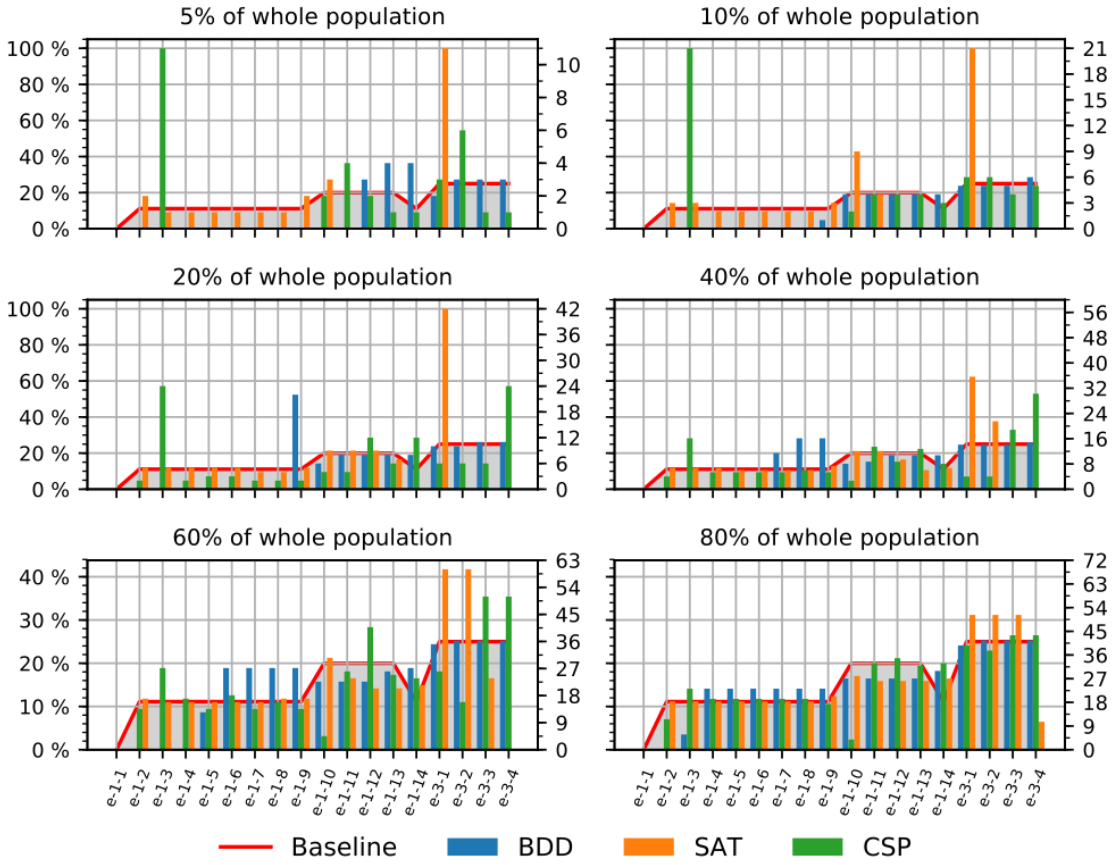


Figure 4.11: Comparison of the feature frequency, for the sampling results of the mandatory and exclusive feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

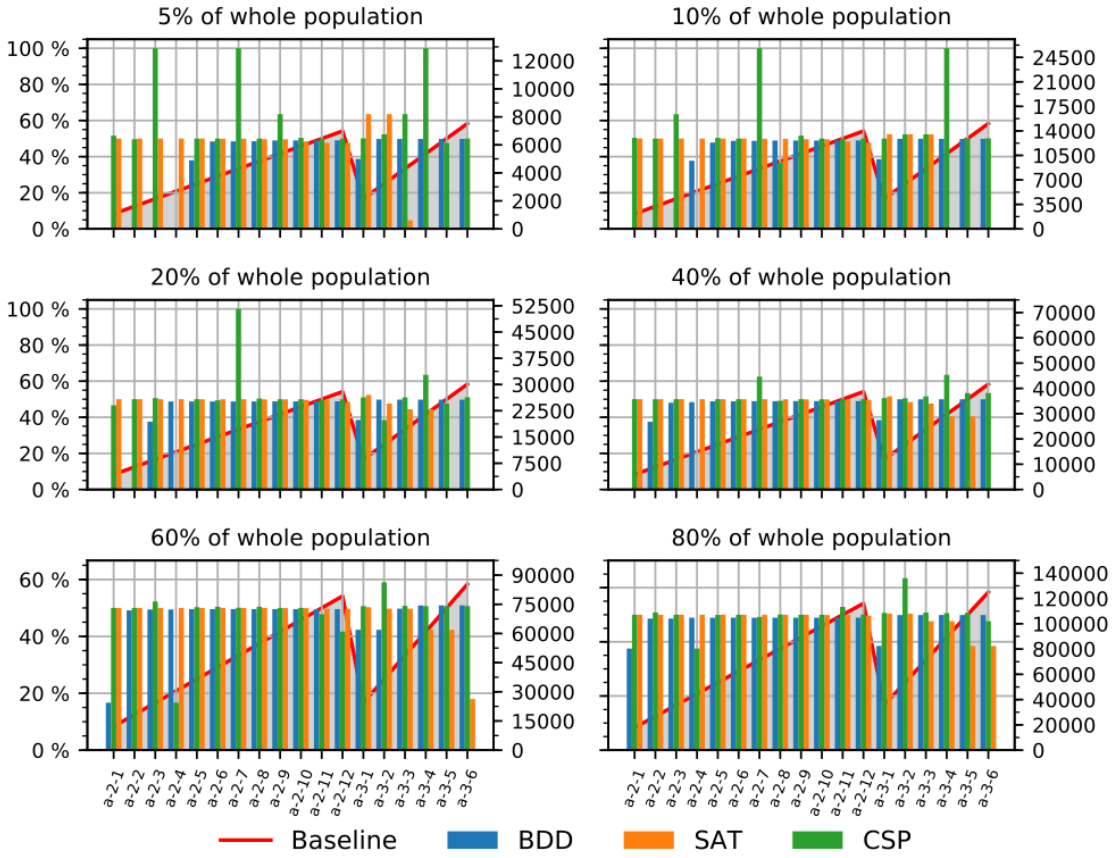


Figure 4.12: Comparison of the feature frequency, for the sampling results of the mandatory and alternative feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

4.2 Real World Feature Models

As proof of the former results, we performed a set of experiments with real configurable systems.
Due to the high number of exclusive configuration options for *TriMesh*, *7z* and *HIPA*^{CC} in combination with an overall number of more than 30 configuration options, we assume the highest runtime for the solver initialization for the BDD solver. By using artificial models, we have encountered that the SAT solver will have the lowest runtime for the initialization and the CSP solver will be slightly above the SAT solver, with a higher variance than the other two solvers. For the runtime to solve the #SAT problem, we expect the same result as for the experiments with the artificial feature models. The CSP and SAT solver will have higher a runtime, depending on the variability defined by the feature model and a nearly linear runtime for the BDD solver. For *TriMesh*, *7z* and *HIPA*^{CC}, we expect higher runtimes due to the higher number of configuration options in relation to the other models. Additionally, we expect a higher memory consumption for this three models, compared to the other four feature models.

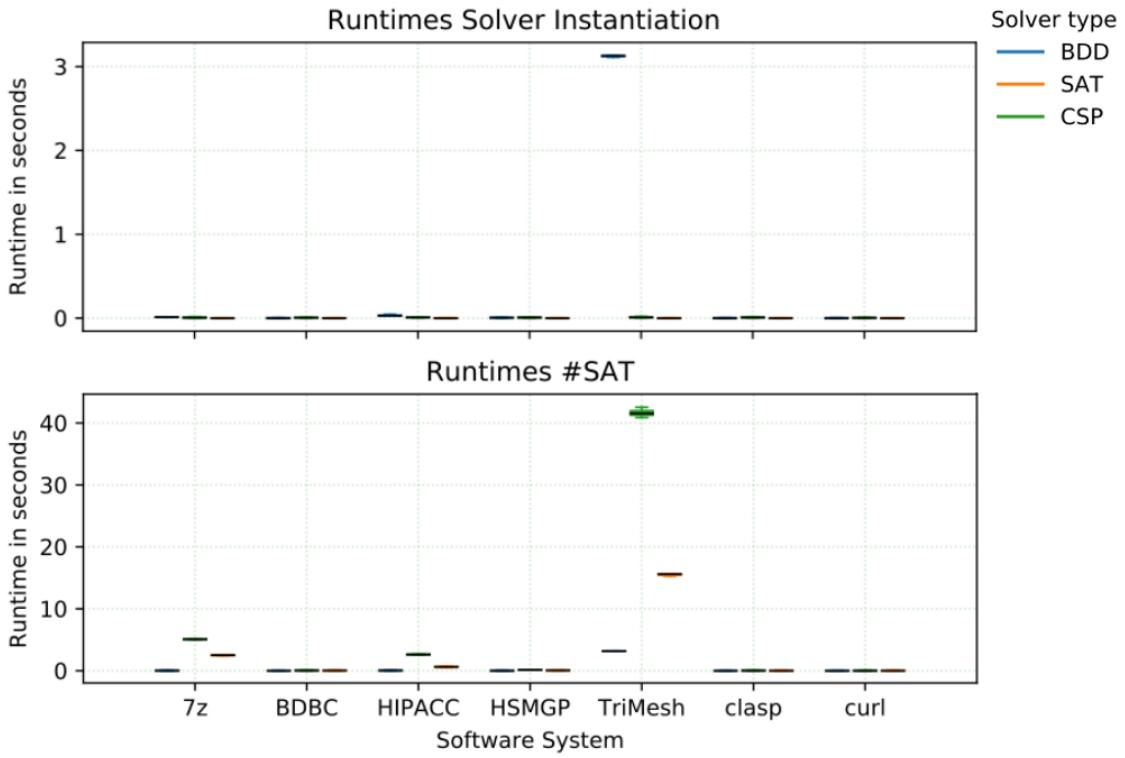


Figure 4.13: Runtime performance comparison, for the solver initialization and solving the #SAT problem for feature models of real world configurable systems. The runtime is shown per solver and model using a box plot. If no runtime variance is shown, it is to small compared to the mean runtime.

4.2.1 Solver Initialization

Shown in Figure 4.13, we can see that the runtime needed by the different solver types for the initialization. As expected, the BDD solver has a higher runtime for

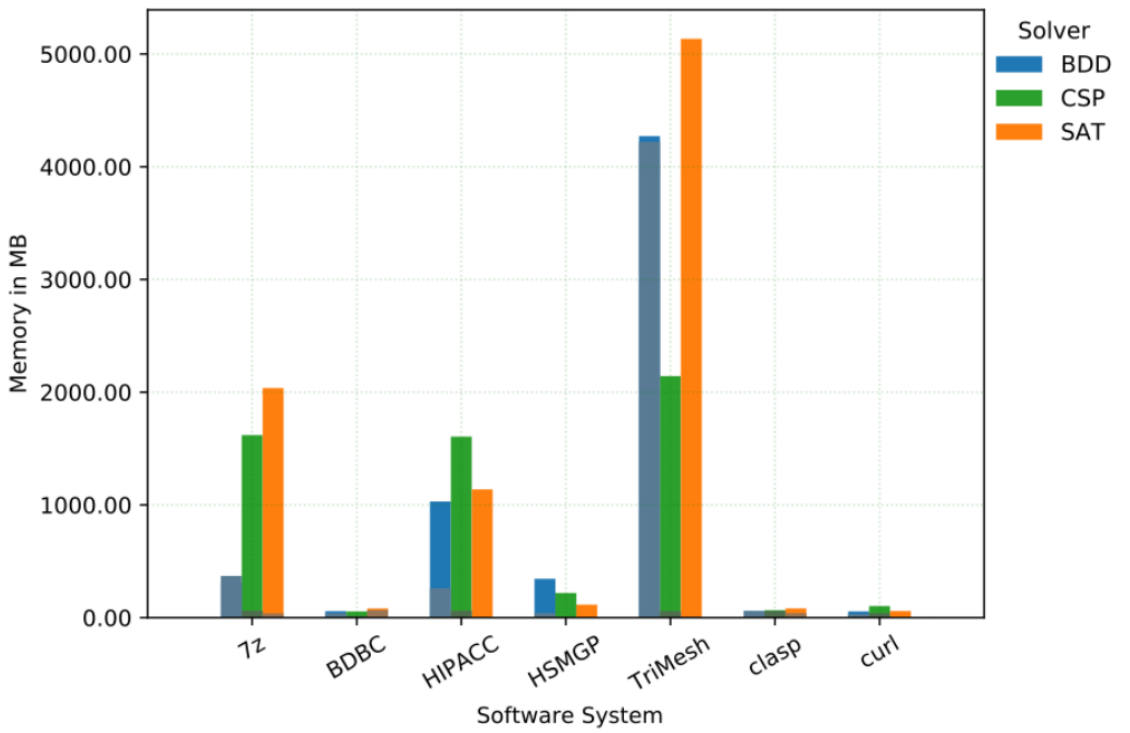


Figure 4.14: Comparison of memory consumption, for solver initialization and solving the #SAT problem. The consumption is shown per solver and model. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for solving the #SAT problem.

TriMesh, *7z* and *HIPA*^{CC} because of the high number of exclusive configuration options. The high runtime for *TriMesh* up to *7z* and *HIPA*^{CC} is related to the number of configuration options as listed at Table 3.2.
Taking the insights from the experiments with artificial feature models, the runtime for the SAT solver are as low as expected, due to the high number of exclusive configuration options and, therefore, reduced variability. Between the SAT and BDD solver, we can find the CSP solver with the usual runtime used for the initialization. As shown in Figure 4.14, the different solvers have a low memory consumption for the solver initialization. Comparing the results for *TriMesh*, *7z* and *HIPA*^{CC}, the BDD solver consumes stepwise more memory for a higher number of exclusive configuration options in combination with a overall high number of configuration options within the feature model. The CSP and SAT solver have a low memory consumption for the solver initialization, as shown in Figure 4.14.

4.2.2 Solving the #SAT problem

In the plot at the bottom of Figure 4.13, we show the runtime for solving the #SAT problem of all three solver types. For the models with a small number of configuration options and higher number of exclusive options we have the smallest runtimes. As an example, for *Clasp* and *Curl* we have the lowest runtimes of all three solvers, compared to the other feature models, because of the small number of configuration

options. Comparing the different solver, the BDD solver is the one with the lowest runtime, followed by the SAT and CSP solver. Taking the models *TriMesh*, *7z* and *HIPA*^{CC}, the CSP solver has a higher runtime than the SAT solver (see Figure 4.13), which only can originate from the high number of exclusive configuration options within the feature models. Additionally, we show the results for the smaller runtimes in Figure A.45, which show the same results.
As shown in Figure 4.14, the BDD solver only slightly consumes additional memory after the initialization, whereas the CSP and SAT solver significant use more memory, depending on the number of valid configurations defined by the feature models.

4.2.3 Sampling

As an example for the performance of sampling with feature models of real configurable systems, we show Figure 4.15. As shown, the CSP solver is the one with the highest runtime, compared to the BDD and SAT solver. Whereas, the BDD solver shows a nearly constant curve for the runtime, in relation to the increasing number of configurations to sample. In addition, as shown in Figure 4.15, the CSP solver heavily depends on the number of configurations to produce.
Together with the runtime, each type of solver needs additionally memory to produce and store valid configurations. Because *TriMesh* and *7z* are the two largest feature models, we have the highest memory consumption compared to the other feature models, shown in Figure 4.16.

As shown in Figure 4.17, all the three solver types start to produce a normal distributed, cardinal distribution (see Figure 4.17 for 5% and 10% of the whole population). The CSP solver starts with sampling configuration with a higher number of selected configuration options. In contrast, the SAT and BDD solver start to sample configurations with a smaller number of selected configuration options. With an increasing number of configurations to sample, the three solver types come closer to the baseline for the cardinal distribution defined by the whole population.
As shown in Figure 4.18, we see the feature frequency for the increasing number of configurations for sampling. For the lower percentages of the whole population, we can see that the solver have a very localized usage of the configuration options defined by the input order of the feature model. The CSP solver is the fastest by using the complete set of configuration options in contrast to the SAT and BDD solver. Both solver, stuck longest on single configuration options as shown in Figure 4.18 for 20% and 40% of the whole population.

Summary

If we compare the results of the feature models originating from real configuration systems together with the results from the artificial feature models, we can state that both sets of experiments produce the same results. The runtime for the CSP and SAT solver for solving the #SAT problem heavily depends on the number of valid configurations defined by the feature model. Additionally, we are able to verify the dependency of exclusive configuration options with the increasing runtime and memory consumption of the BDD solver initialization, as shown in Figure 4.13 and Figure 4.14.

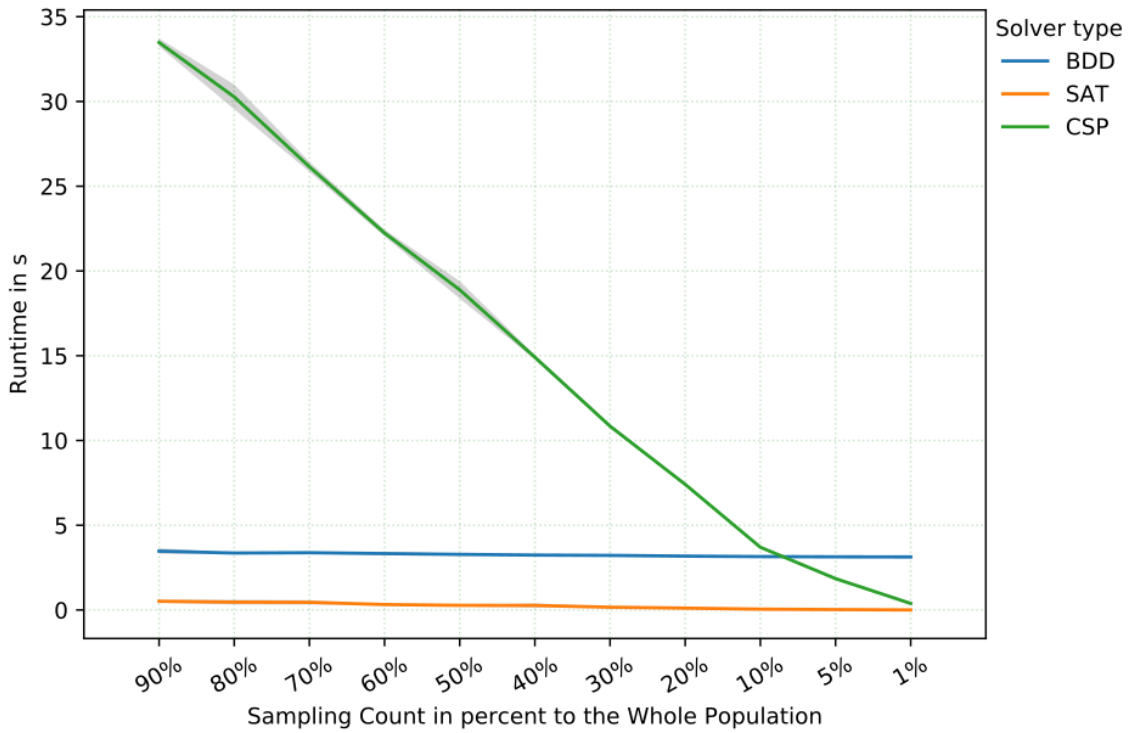


Figure 4.15: Runtime performance comparison, for sampling subsets of the TriMesh feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

For the sampling performance, we made the same observations regarding runtime and memory consumption, as for the artificial models, the runtime and memory consumption increases with number of configurations to sample. For the sampling quality, we found that the solver attempt to sample with a normal distributed cardinality in general. With a increasing number of configurations, this will be complicated by the restrictions of the feature model, defining the distribution of the whole population. This comes together, with the tendency that the solvers tend to sample with a higher or lower number of selected configuration options within a configuration. The results for the feature frequency remains the same as for the artificial feature models, the CSP solver uses more different configuration options within the sampling process. Also, the SAT and BDD solver show the same behavior as for the artificial feature models (i.e. the localized sampling of single configuration options) for the feature frequency.

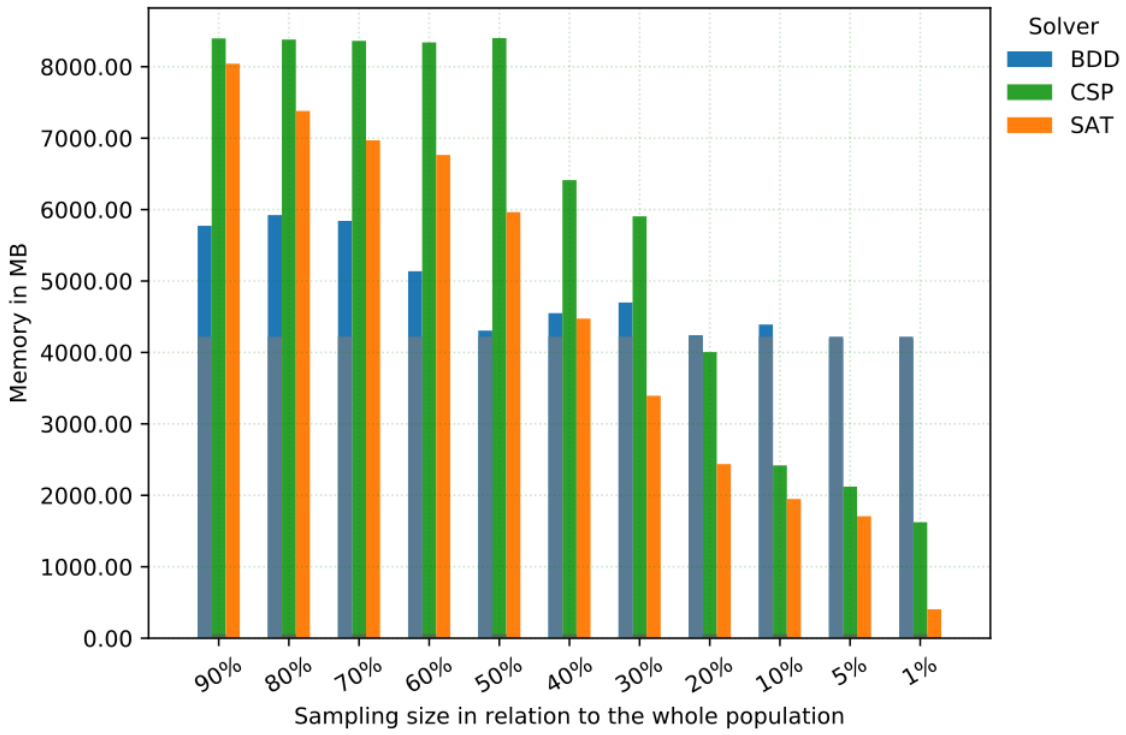


Figure 4.16: Comparison of memory consumption, for sampling subsets of the TriMesh feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

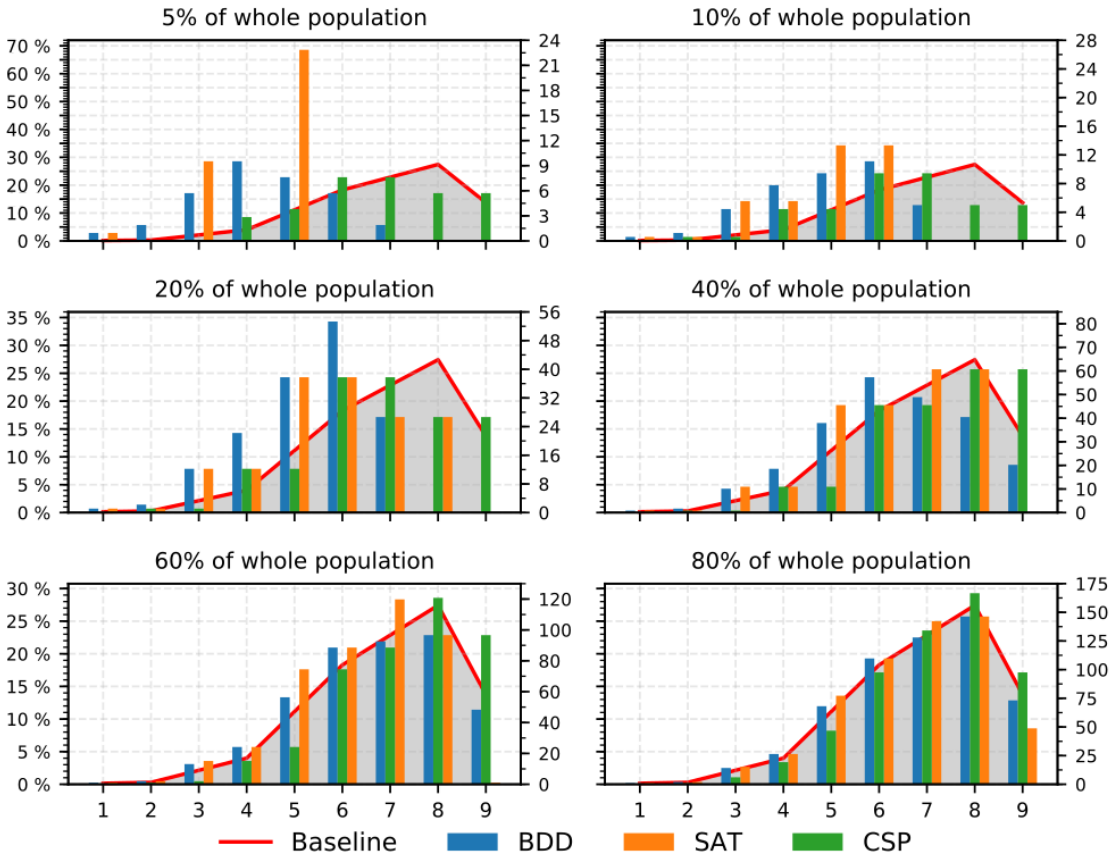


Figure 4.17: Comparison of the cardinal distribution, for the sampling results of the Clasp feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

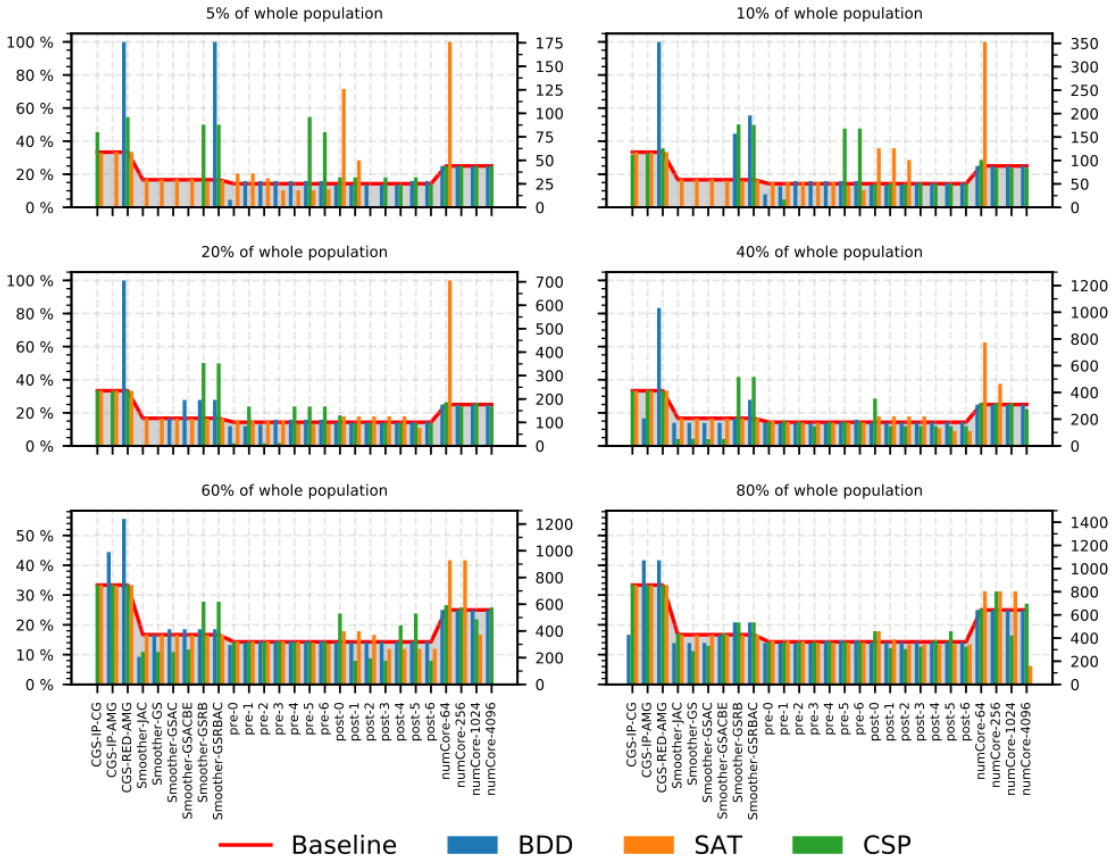


Figure 4.18: Comparison of the feature frequency, for the sampling results of the Hsmgp feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

4.3 Threads To Validity

In this section, we describe the threats to validity that influenced our work. We categorize the threats in internal (i.e., factors that threaten our implementation and evaluation) and in external (i.e., factors that threaten the generalizability of this work) threats.

4.3.1 Internal Validity

Developed Application

The first threat we saw during the thesis, is the application we developed to wrap and compare the different domain constraint solver. Nevertheless we took a lot of effort to ensure the influences by the application are minimal as possible for our measurements. To reduce the influences of the target system, we repeated the tests 11 times, for each feature model.

SPLOT Feature Model Generation

By using the SPLOT feature model generator for the generation of feature models, we observed that SPLOT does not fully respect our configuration. For instance, we give the SPLOT feature model generator the parameter to create a feature model with 30% of mandatory options and 70% of alternative options. All this configuration options should be on the same depth of the feature tree to ensure only to measure the type of configuration option influences. To address this problem, we checked the feature models afterwards and excluded feature models that violated the configuration. Therefore, we opened an issue at the repository of SPLOT[1].

Memory

During the explanation of our experimental setup, we state the use of *ps_mem* as a better approach, compared to the use of programming the memory measurement in combination with triggering the garbage collector. We state that this approach has less influences for the program execution.
One drawback of *ps_mem* is the triggering of the memory measurement, only every second. Therefore, we executed the memory measurement multiple times, to catch also states were the garbage collector was not executed before the memory is measured.

4.3.2 External Validity

Feature Models of Real World Configurable Systems

In our evaluation, we use seven different feature models from different real configurable systems. To increase the internal validity, we used only one version of the configurable systems. While we expect that this configurable systems will evolve after our experiments, the feature models will also change in their number of configuration options and constraints. To increase the external validity, we have chosen the real-world model in such a way that they contain different number of configuration options, and different properties, to get a optimal level of test coverage for real configurable systems.

[1] https://github.com/marcilio/splot/issues/1

Solver Libraries

While we compare three different domain constraint solver, we used only three different libraries one for each approach. This may lead to inconsistencies because of specific implementational details. Because we can't detail which specific optimizations are used within each implementation.
As counterpart to this, we chooses popular implementations, which are widely used. The Sat4j solver is also used by other researcher like Benavides et al. [BTC05, BSTRC06a, BSTRC06b]. Second the Choco solver is also a very popular implementation for the CSP approach and also used by Benavides et al.. The BDD library we used is the only one not such as popular as Choco or Sat4j, but with JDD we choose a pure and native Java implementation also used by Oh et al. [OBMS16, OBMS17]. If we take all this into account, we can state our choice as a valid one for the solver libraries. Additionally, we were limited to time constraints and had to limit the solver implementations within our test setup.

Sampling Results

The sampling results showed above, have a clear statement for the quality of the sampled sets of valid configurations. Important for this, is the knowledge that we can state this results only for the solver implementations we use. If other implementations will be used, the results may vary, due to other algorithms with the changing implementation.

No use of solver implementations

During our experiments, we use the solver implementations without explicitly activating or slot optimizations in ahead. We use the implementations with their default configuration. If the implementations use optimizations internally, this is out of scope within our experiments. We accept this circumstances and decrease of external validity, due to increase the internal validity.

5. Related Work

Combining model counting together with domain constraint solver is not a new use case [GSS08]. But using domain constraint solver to analyse feature models is not a common approach.
Benavides et al. started to use domain constraint solver to reason feature models automatically. Within a literature summary Benavides et al. [BSRC10], summarizes different analysis metrics for feature models, as well as model counting.
The first work of Benavides et al. [BSTRC06b], uses two different CSP solver, Choco and JaCoP[1] for feature models ranging from 15 to 52 configuration options. In their work, they compare the performance of both CSP solvers on finding one single configuration and for solving #SAT. As a result they state that Choco is more performant for #SAT than JaCop, whereas JaCop is more performant for finding one configuration.
In another work, Benavides et al. [BTC05] shows more ways of analysing feature models, additionally to the topics targeted by us, #SAT and sampling. To perform the tests they use the commercial domain constraint solver **OPL Studio**[2]. Within their survey, they use feature models with 15 to 25 configuration options. They mainly target on evaluating changes on features, which can be the adding or removing of configuration options or maybe define a predefined set of configurations options to use. All this changes influence the feature model and they evaluate how strong the solver is affected by this changes. As result they have higher runtimes for feature models with less commonalities than configuration options increasing the number of valid configurations.
Our last view on a related work of Benavides et al. [BSTRC06a], uses 3 different solver, JaCoP as CSP solver implementation, JavaBDD[3] as BDD implementation and last Sat4j as SAT solver. They use 50 different randomly generated feature models, from 50 to 300 configuration options, to evaluate the performance of the three solver implementations for #SAT and finding one valid configurations. Besides,

[1] https://osolpro.atlassian.net/wiki/spaces/JACOP/pages/26279944/JaCoP+-+Java+Constraint+Programming+solver (last visited 2018/05/03)
[2] https://www.ibm.com/de-en/marketplace/ibm-ilog-cplex (last visited 2018/05/03)
[3] http://javabdd.sourceforge.net/ (last visited 2018/05/03)

they add random cross tree constraints to up to 25% of the number of configuration options. For the performance measurement they take the memory consumption as well as the runtime into account. For finding one valid solution, they have the lowest runtime for the BDD solver compared to the CSP and SAT solver. As second result, the state a superiority of the BDD solver for #SAT compared to the CSP and SAT solver. The last result, is the significant higher memory usage compared to the CSP and SAT solver.
At all works of Benavies et al., they use #SAT as a test-case for the domain constraint solver they use, as well as feature models changing in their properties. The differences from Benavides et al. to our work are mainly the differences for the solver implementations. They use JaCop or OPL Studio as CSP solver within two works [BTC05, BSTRC06b], but they did not consider other constraint solver approaches. In contrast to this, in the parallel work [BSTRC06a], they made a comparison of different solvers, whereas the implementations used for CSP and BDD are different to our used implementations. Together with the different solver implementations used, the main difference to our work, is the setup for the feature models. Benavides et al. uses always randomly generated feature models up to 300 configuration options and a increasing number of cross tree constraints from 0% to 25% of the number of configuration options. In contrast to this we used artificial feature models with explicit defined properties for the types of configuration options and cross tree constraints. They use the feature models as the changing factor for their experiments, in contrast to our changing number of configuration option types, together with the changing feature models.
Besides, there is also other work, that aims at identifying the influences of changing properties of feature models on the performance of domain constraint solvers. For example, Segura, who compared three different domain constraint solver (one SAT, one BDD, one CSP solver) [Seg08]. He compared the different domain constraint solver within their performance for finding one valid configuration and #SAT. As feature models he uses models generated with the FAMA [BSTC07] tool set within a range from 50 to 300 configuration options and a increasing number of cross tree constraints up to 25% of used configuration options. The main goal of his work is to evaluate the influences of atomic sets onto the performance of the three solver types, atomic sets are a pre-optimization reducing the overall number of configuration options and constraints, by combining highly related configuration options. As a result, he have a reduction in memory and runtime when using atomic sets on feature models, for the tasks of solving #SAT and finding one valid configuration. Whereas the improvements are smaller for small feature models, he state a significant performance improvement for all three solver types, when using large feature models.
In contrast to our work, Segura uses a set of randomly generated feature models from 50 till 300 configuration options, with a increasing number of cross tree constraints. The properties of each feature model is not taken into account during his evaluation. Additionally, he edit the feature models by applying the atomic set algorithm on each feature model, which also influences the properties of feature models. The second difference is the performance measurement, he compares the solver types by the popular find one valid configuration and #SAT, while we take the performance for the sampling into account and additionally compare the quality of the samples

sets.
In another work, Pohl et al. [PLP11] compare different domain constraint solver within their performance for #SAT and finding one configuration. In sum, they use different solver implementations with a wide variety of Java and C/C++ implementations. They consider feature models of different size of the whole population, defined by the feature models themselves. For this they used 90 different feature models from the set of the SPLOT project [MBC09]. As a result of the work, they state the BDD solver as best performing on larger feature models and in special for the #SAT problem. They say that the other solver types, CSP and SAT are more suitable for smaller models of finding only one valid configuration. As a special point they mark the performance of the BDD solver not as predictable as for the native implementations of the CSP and SAT solvers.
The first difference to our work, is the selection of feature models. They used 90 different feature models chosen how they fit to their time constraints and if they are usable by the chosen domain constraint solver. The second difference are the used domain constraint solver implementations as well as the number of the tested solver implementations, 9 in complete. For this they took 3 different implementations per domain constraint solver approach, implemented in Java and C/C++.

6. Conclusion and Future Work

Feature models are one common way to manage and validate the variability and commonality of configurable systems. Therefore, they are a central point for research on configurable systems. A common question placed together with feature models, is the one, how much valid configurations are defined by them. Although feature models contain configuration options and the relations among them, it is still a difficult task to enumerate all valid configurations, which is known as the #SAT problem. To solve the #SAT problem, several approaches were developed. Most of them origin from the field of constraint programming or boolean satisfiability solving.

In this work, we analyzed three different domain constraint solver for their ability to solve the #SAT problem. In this work, we focus on the influences of different properties of feature models on the performance of the different domain constraint solver. With this thesis, we fulfill a basic research of domain constraint solver for their ability to solve the #SAT problem. Compared to other research in comparing domain constraint solver, we use artificial feature models to identify the influences of different types of configuration options and different relations among them on runtime and memory of the domain constraint solver. Together with the evaluation of the influences of different types of configuration options on the solver performance, we identify the quality of sampling the different solver types reach, for different sampling sizes. This will be helpful for research using sampling sets as base.

We showed that the BDD approach is the best to solve the #SAT problem, because the runtime keep constant for a fixed number of configuration options, independent from the different properties of feature models with the same number of configuration options and therefore, the size of the configuration space defined by the feature model. However, the BDD has a higher runtime and memory consumption during the initialization, when using exclusive configuration options because of the internal representation. As a consequence thereof, the runtime and memory consumption increase during the initialization. Additionally, we were able to evaluate, that the performance of the CSP and SAT solver will not directly depend on explicit types of configuration options, rather than the number of possible valid configurations defined by the feature model itself. Especially for the #SAT problem, the use of the

loop-back approach proves to be critically. Within the field of sampling, the CSP solver found to be the one with the most extensive use of configuration options of all the three solvers.
As future work, we could add further domain constraint solver to the analysis to increase the external validity of this work. Another topic, would be the research on sampling methods based on domain constraint solver, improving the sampling quality to mimic the distribution of the whole population, with the smallest number of configurations possible.

A. Appendix

A.1 Solver Implementation Listing

A.2 Additional Results

Solver Name	Implementation Language	Link
SAT-Solver Libraries		
Dimetheus	C/C++	https://www.gableske.net/dimetheus
Lingeling, Plingeling	C/C++	http://fmv.jku.at/lingeling/
Riss	C/C++	http://tools.computational-logic.org/content/riss.php
pycosat	Python	https://pypi.python.org/pypi/pycosat
soSAT	Python	https://github.com/domoritz/SoSAT
Sat4j	Java	http://www.sat4j.org/
Sugar	Java	http://bach.istc.kobe-u.ac.jp/sugar/
CSP-Solver Libraries		
BTD	C++	http://www.cril.univ-artois.fr/XCSP17/files/BTD.pdf
Cosoco	C++	—
Mistal	C++	https://github.com/ehebrard/Mistral-2.0
Naxos	C++	https://github.com/pothitos/naxos
Concrete	Scala	https://github.com/concrete-cp/concrete
Oscar	Scala	https://bitbucket.org/oscarlib/oscar/wiki/Home
ABSCon-Basic	Java	https://www.cril.univ-artois.fr/∼lecoutre/software.html#
Choco-Solver	Java	http://www.choco-solver.org/
Diet-Sugar	Java	http://kix.istc.kobe-u.ac.jp/∼soh/dsugar/
Sat4j-CSP	Java	http://www.sat4j.org/products.php#csp
BDD-Solver Libraries		
Buddy	C++	http://buddy.sourceforge.net/manual/main.html
miniBDD	C++	http://www.cprover.org/miniBDD/
bdd-for-c	C	https://github.com/grassator/bdd-for-c
The BDD library	C	https://www.cs.cmu.edu/∼modelcheck/bdd.html
bddSharp	C#	https://github.com/sorenjuul/bddsharp
JavaBDD	Java	http://javabdd.sourceforge.net/
jdd	Java	https://bitbucket.org/vahidi/jdd/wiki/Home
lightBDD	Java	https://github.com/SigmaX/LightBDD

Table A.1: Table of different domain constraint solver implementations, we consider during our survey. The library we use, are marked with a gray background. (latest update 2018/13/01)

A.2.1 Solver Initialization

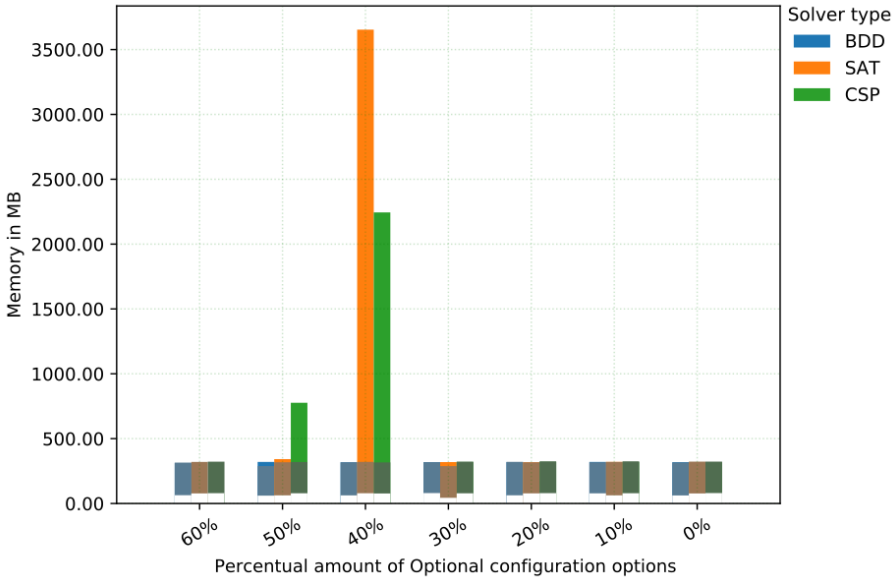


Figure A.1: Plot of the memory consumption for 30 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 60% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

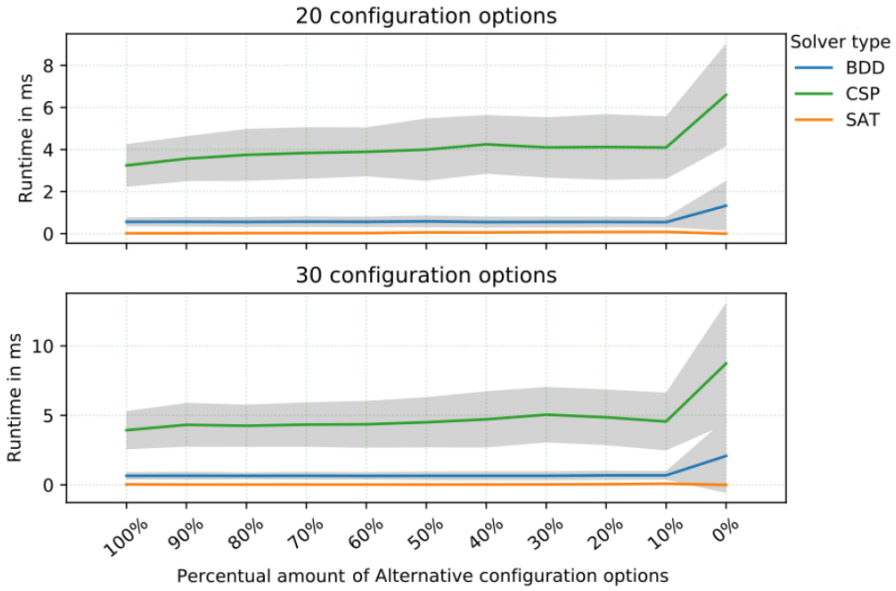


Figure A.2: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for the initialization. The runtime is shown per solver, from 100% of alternative configuration options down to 0% alternative configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

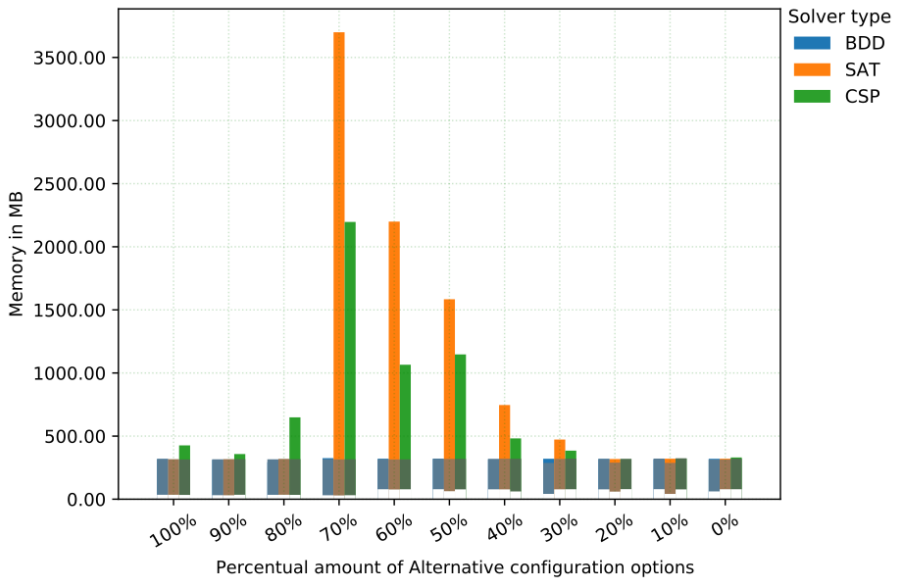


Figure A.3: Plot of the memory consumption for 20 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 100% of alternative configuration options down to 0% alternative configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

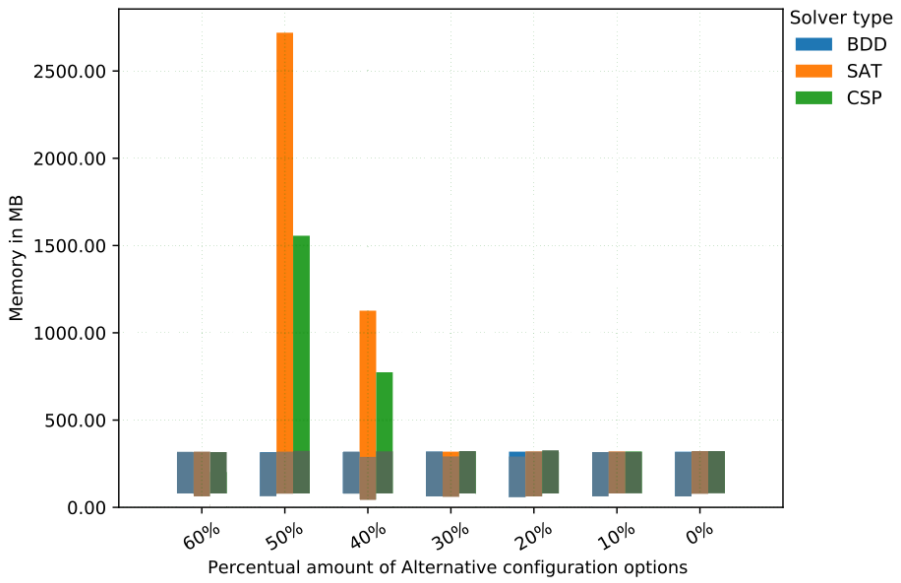


Figure A.4: Plot of the memory consumption for 30 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 60% of alternative configuration options down to 0% alternative configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

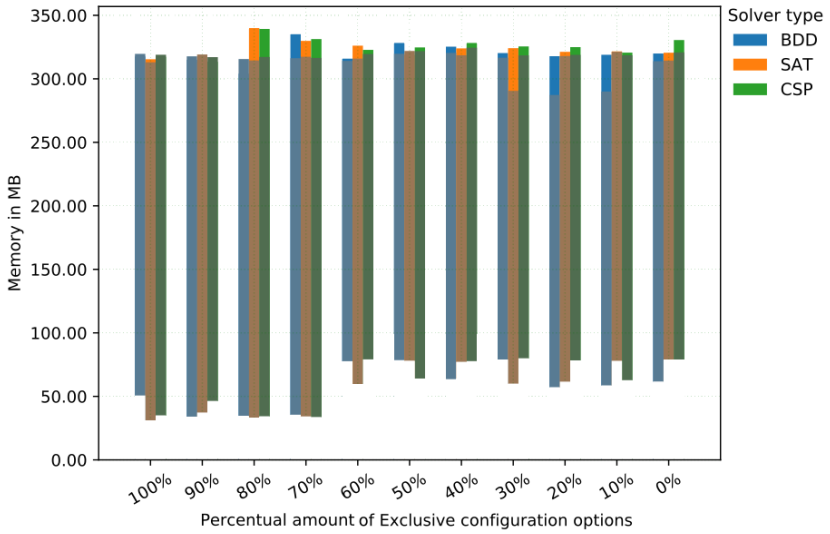


Figure A.5: Plot of the memory consumption for 20 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 100% of exclusive configuration options down to 0% exclusive configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

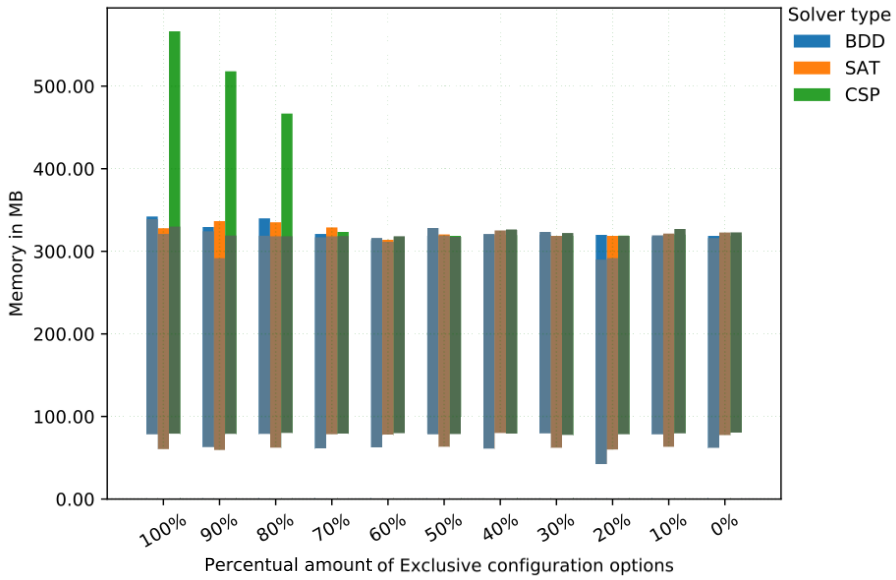


Figure A.6: Plot of the memory consumption for 30 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, from 100% of exclusive configuration options down to 0% exclusive configuration options, in relation to the number of configuration options. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

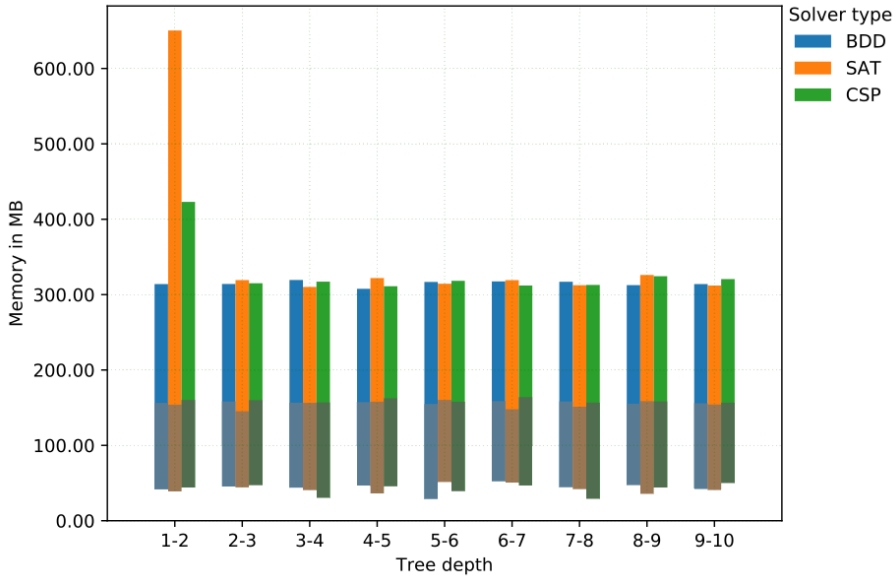


Figure A.7: Plot of the memory consumption for 20 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, for an increasing depth of the feature tree. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

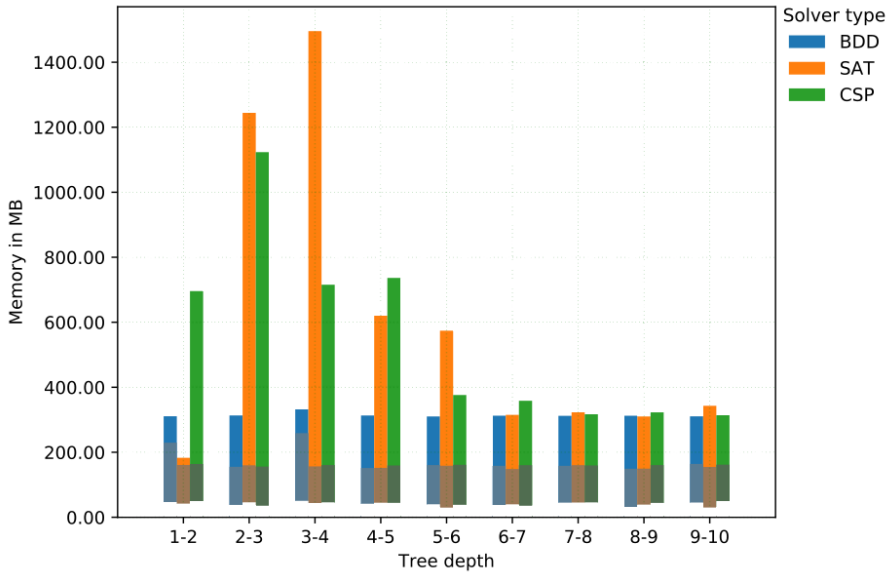


Figure A.8: Plot of the memory consumption for 30 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, for an increasing depth of the feature tree. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

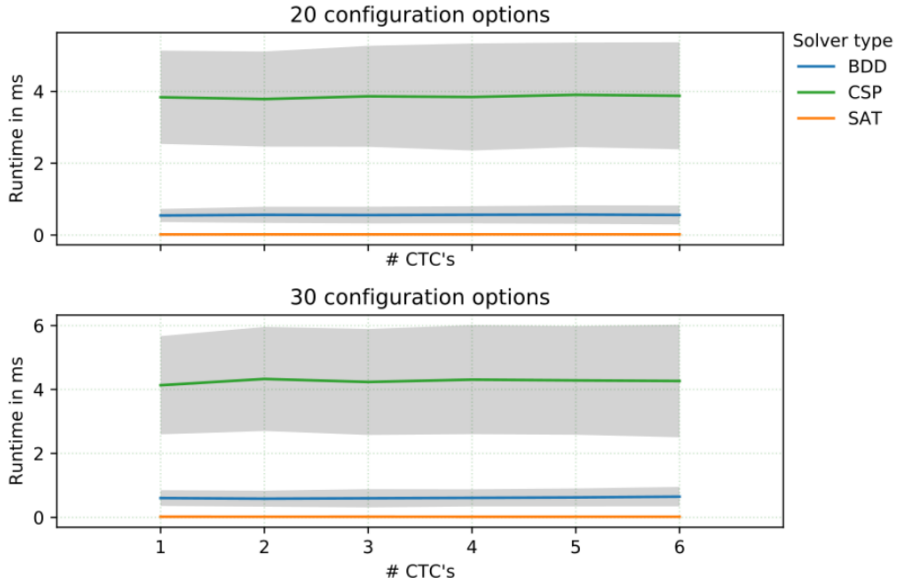


Figure A.9: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for the initialization. The runtime is shown per solver, for a increasing number of cross tree constraints, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

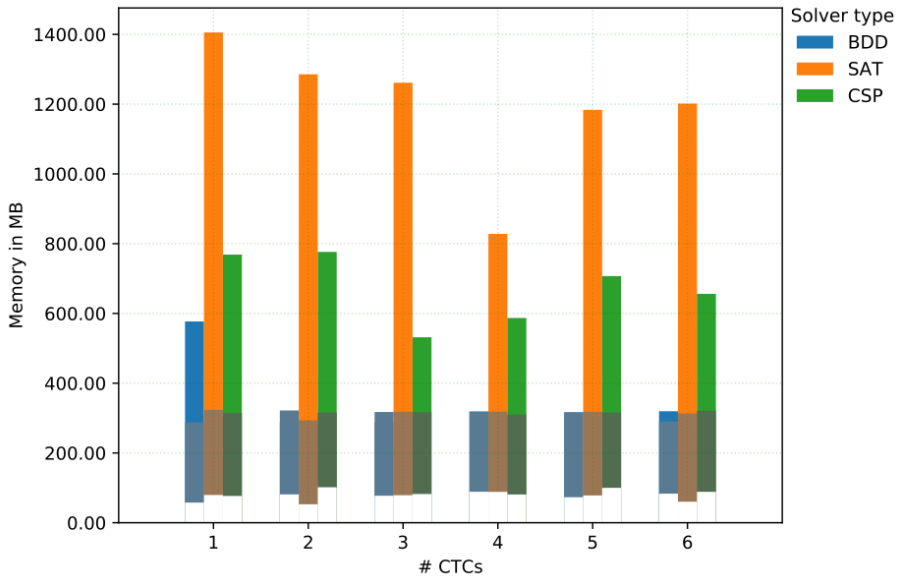


Figure A.10: Plot of the memory consumption for 20 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, for an increasing number of cross tree constraints. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

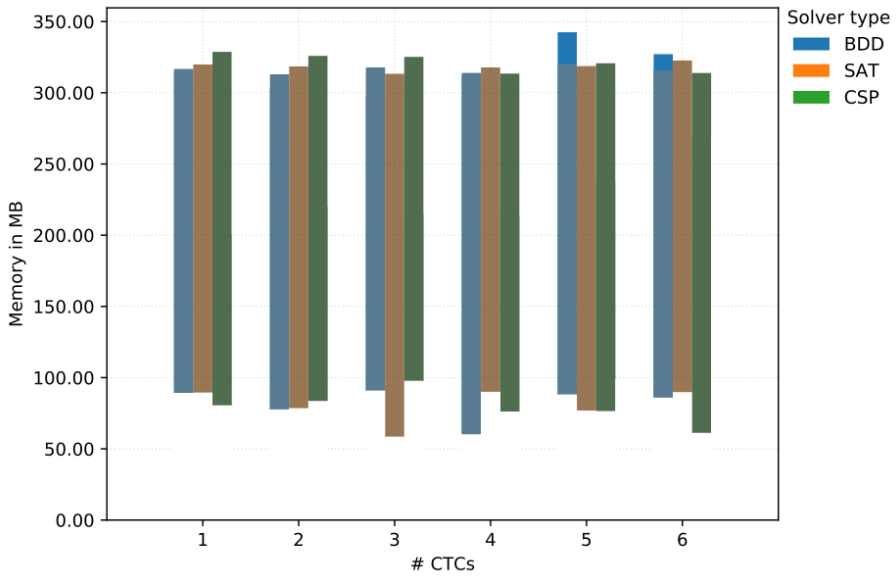


Figure A.11: Plot of the memory consumption for 30 configuration options, needed by solver instances for the initialization and solving the #SAT problem. The needed memory is shown per solver, for an increasing number of cross tree constraints. The memory consumption for initialization is shown with a shaded bar, above we have the memory consumption for solving the #SAT problem.

A.2.2 Solving the #SAT problem

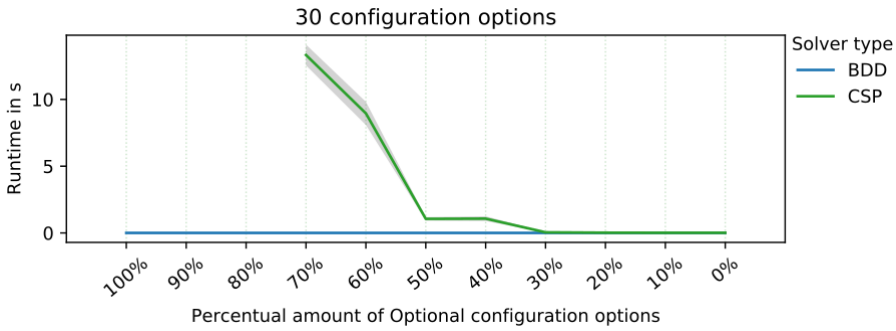


Figure A.12: Runtime performance comparison, needed by the solver instances for solving the #SAT problem, without SAT solver for better CSP and BDD solver comparison. The runtime is shown per solver, from 100% of optional configuration options down to 0% optional configuration options, in relation to the number of configuration options, with 30 configuration options overall. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance is to small to plot.

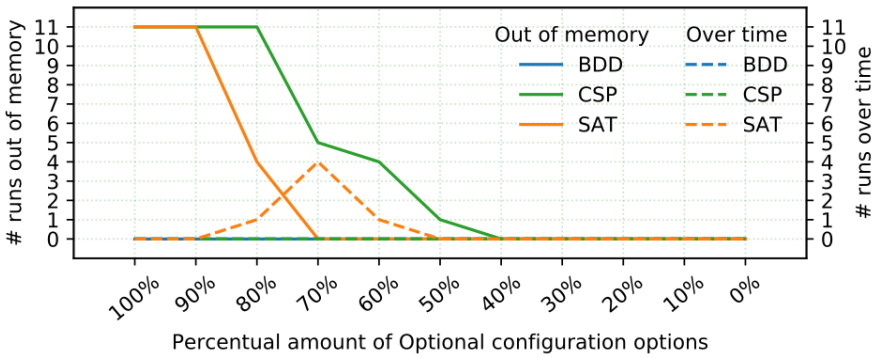


Figure A.13: Failure listing of the three solver types. Straight lines are for the number of runs aborted due to reaching memory constraints. Dashed lines are for the number of runs aborted due to reaching the time limit of 4 days for one model instance.

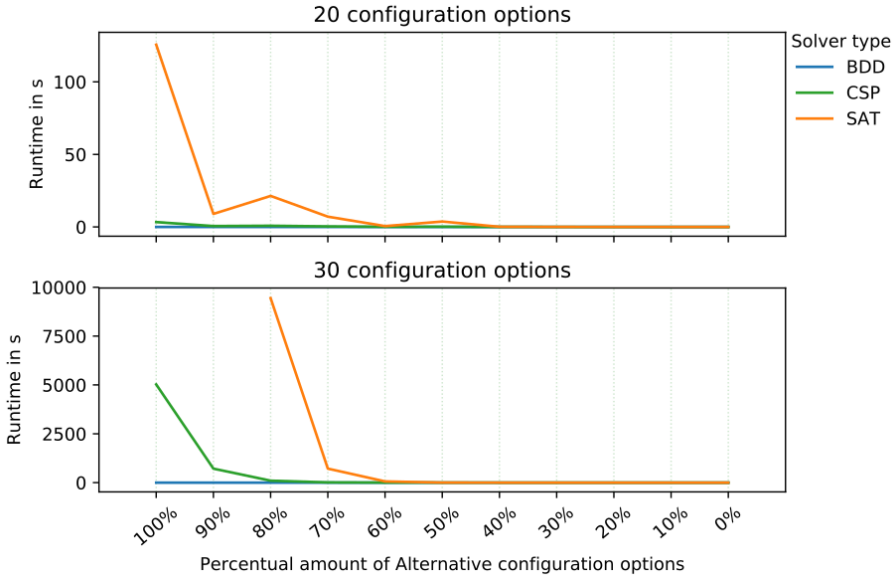


Figure A.14: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for solving the #SAT problem. The runtime is shown per solver, from 100% of alternative configuration options down to 0% alternative configuration options, in relation to the number of configuration options. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

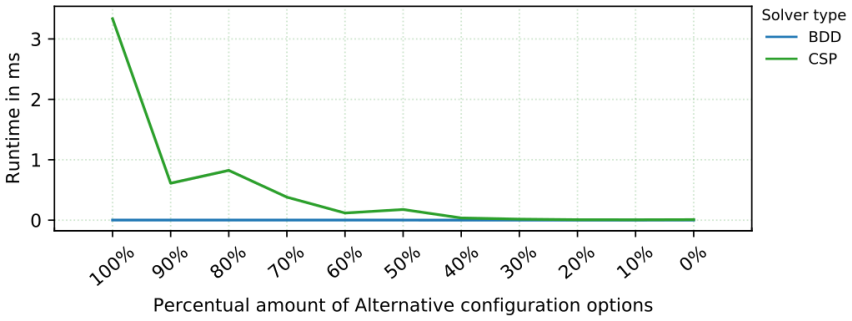


Figure A.15: Runtime performance comparison, for solving the #SAT problem, without SAT solver for better CSP and BDD solver comparison. The runtime is shown per solver, from 100% of alternative configuration options down to 0% alternative configuration options, in relation to the number of configuration options, with 20 configuration options overall. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance is to small to plot.

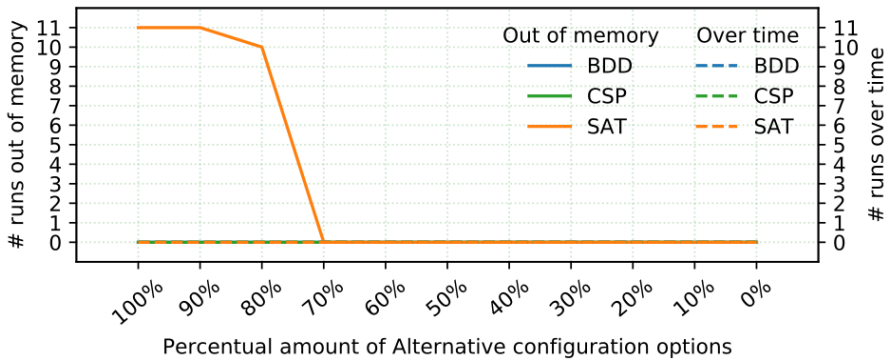


Figure A.16: Failure listing of the three solver types. Straight lines are for the number of runs aborted due to reaching memory constraints. Dashed lines are for the number of runs aborted due to reaching the time limit of 4 days for one model instance.

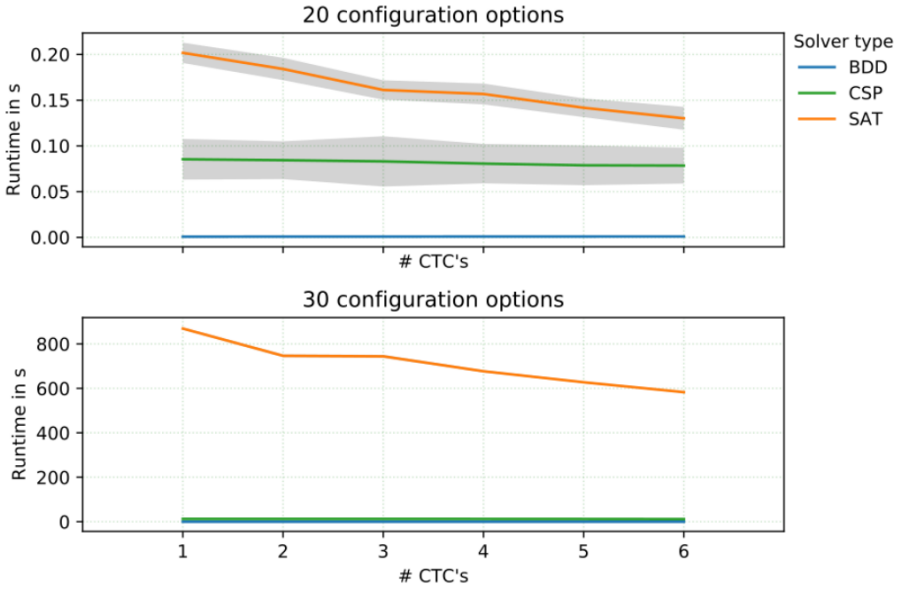


Figure A.17: Two plots of runtimes with 20 and 30 configuration options, needed by the solver instances for solving the #SAT problem. The runtime is shown per solver, for a increasing number of cross tree constraints. Each curve of the solver runtime is surrounded by the runtime variance shown in gray, if omitted the runtime variance are to small to plot.

A.2.3 Sampling

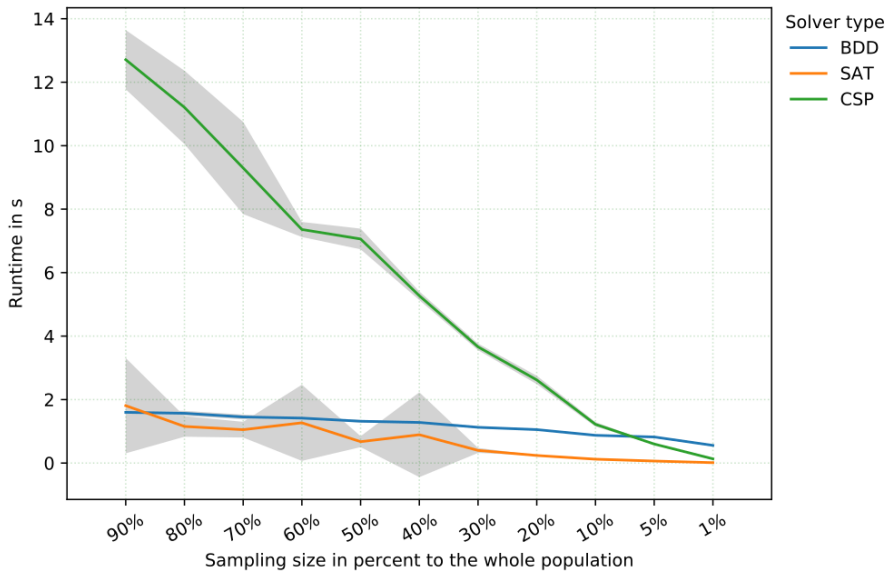


Figure A.18: Runtime performance comparison, for sampling of configurations from the mandatory and optional artificial feature model with 30 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

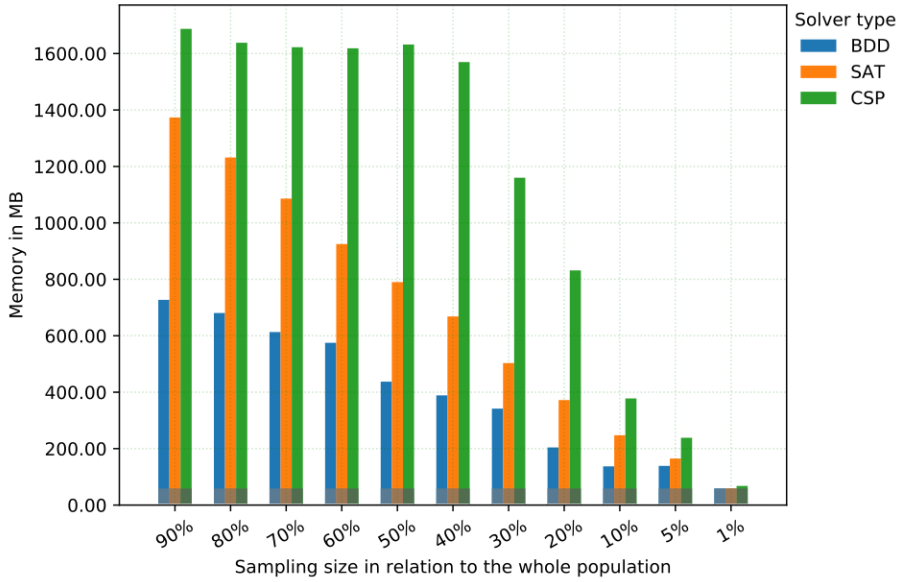


Figure A.19: Comparison of memory consumption, for sampling subsets of the mandatory and optional artificial feature model with 20 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

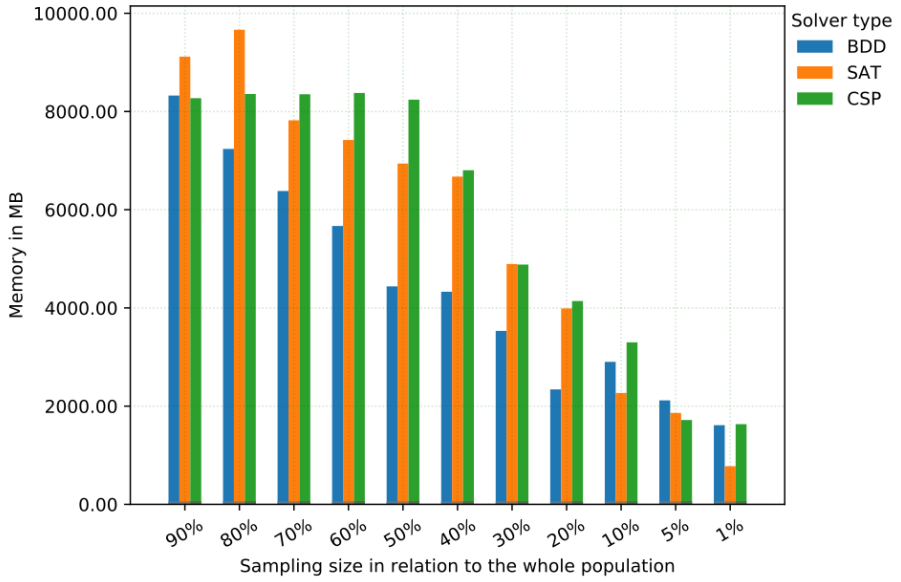


Figure A.20: Comparison of memory consumption, for sampling subsets of the mandatory and optional artificial feature model with 30 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

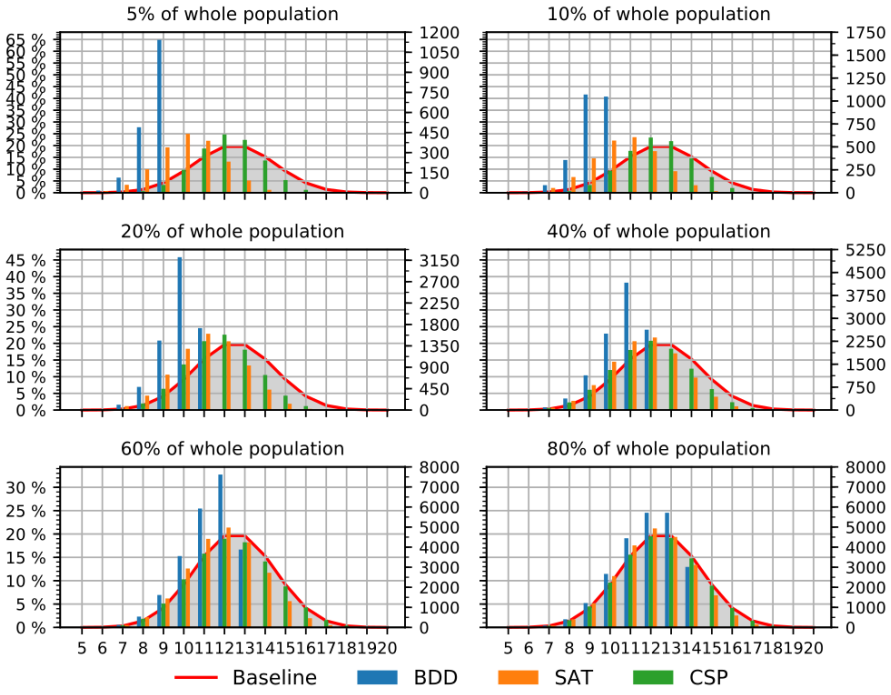


Figure A.21: Comparison of the cardinal distribution, for the sampling results of the mandatory and optional feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

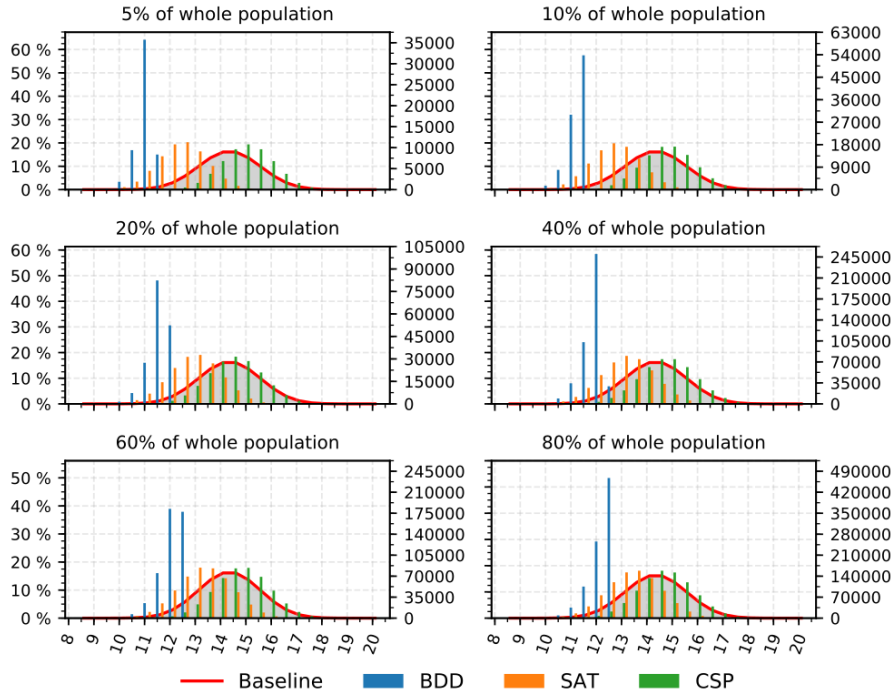


Figure A.22: Comparison of the cardinal distribution, for the sampling results of the mandatory and optional feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

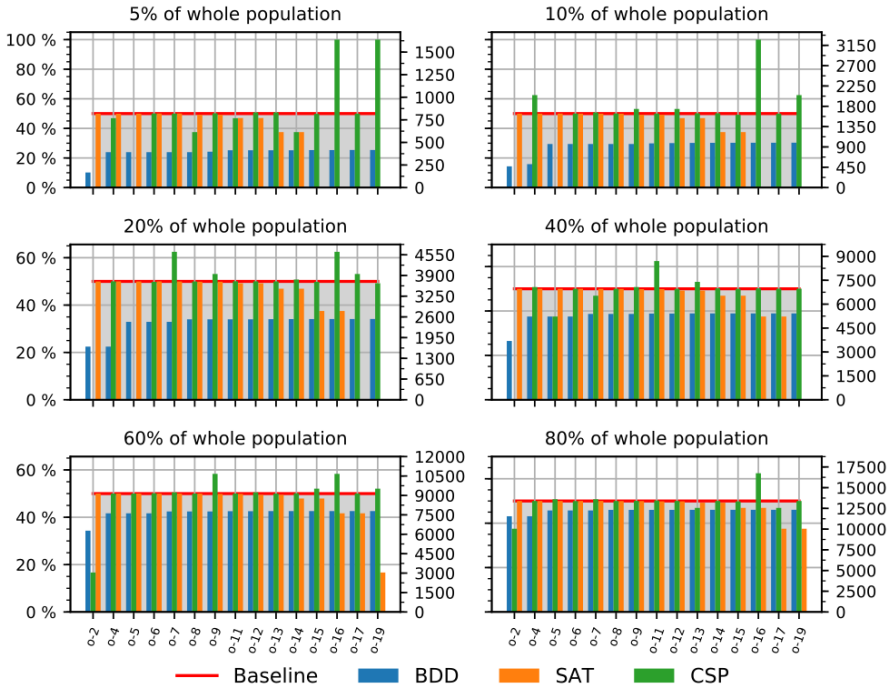


Figure A.23: Comparison of the feature frequency, for the sampling results of the mandatory and optional feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

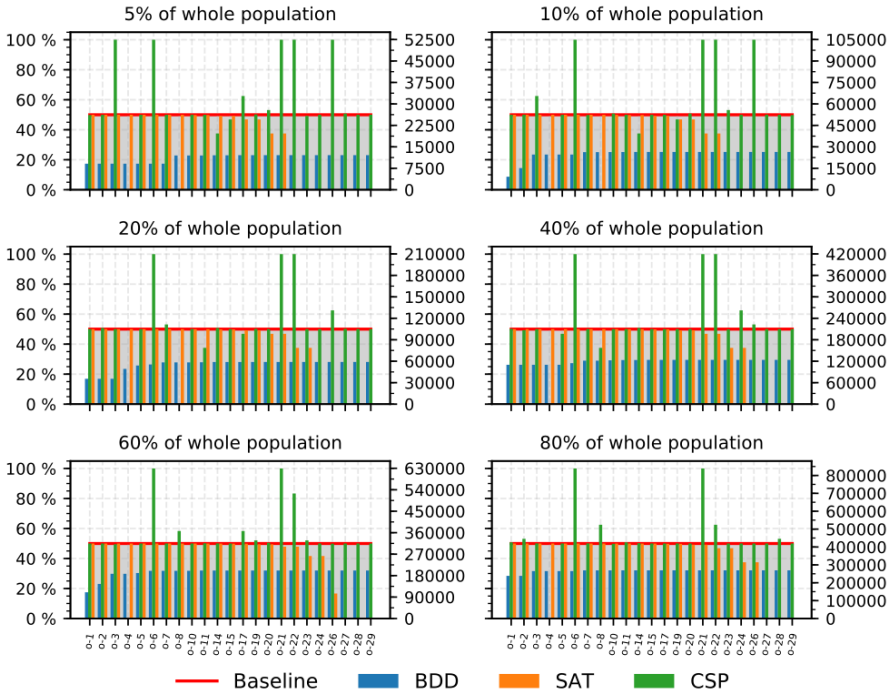


Figure A.24: Comparison of the feature frequency, for the sampling results of the mandatory and optional feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

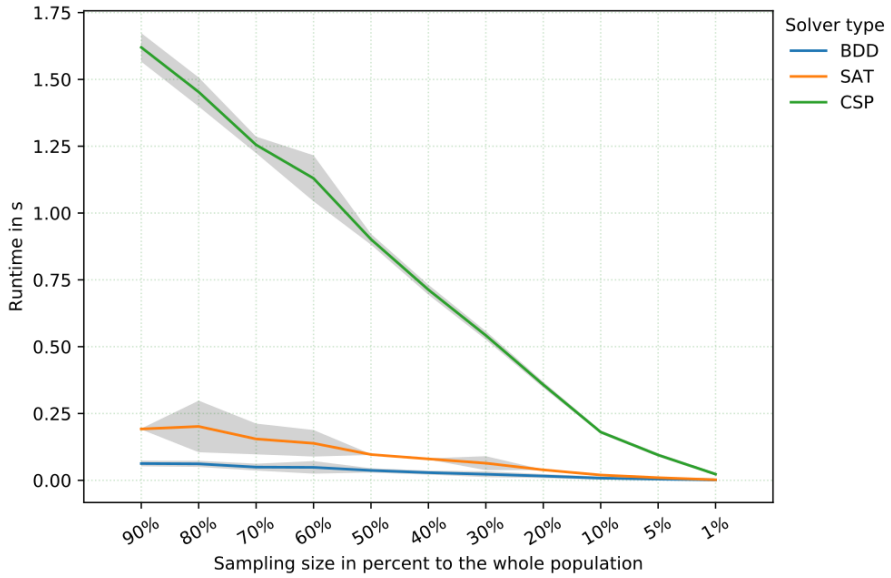


Figure A.25: Runtime performance comparison, for sampling of configurations from the mandatory and alternative artificial feature model with 20 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

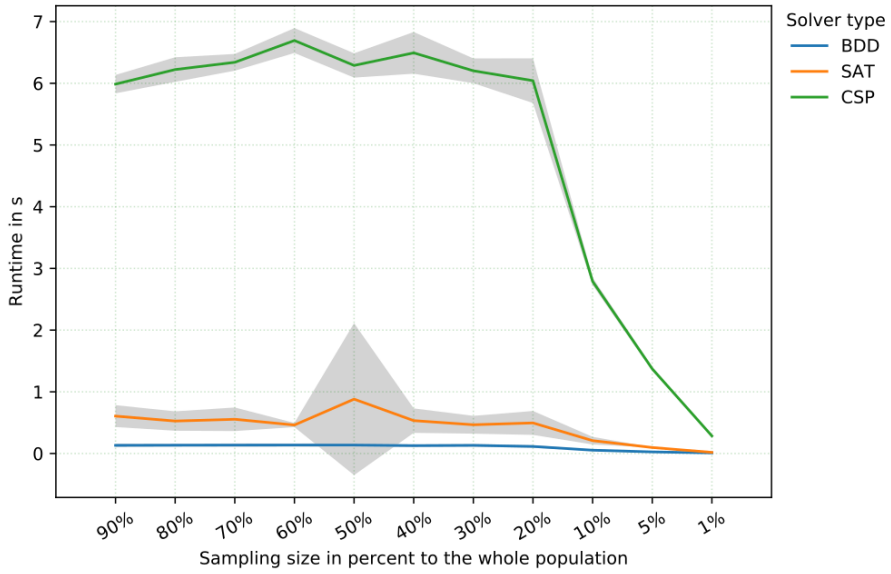


Figure A.26: Runtime performance comparison, for sampling of configurations from the mandatory and alternative artificial feature model with 30 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

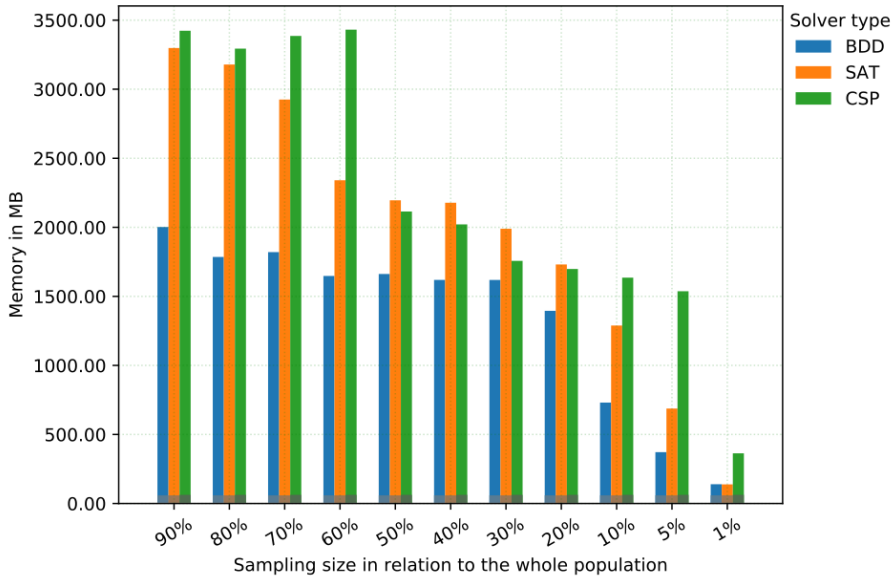


Figure A.27: Comparison of memory consumption, for sampling subsets of the mandatory and alternative artificial feature model with 20 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

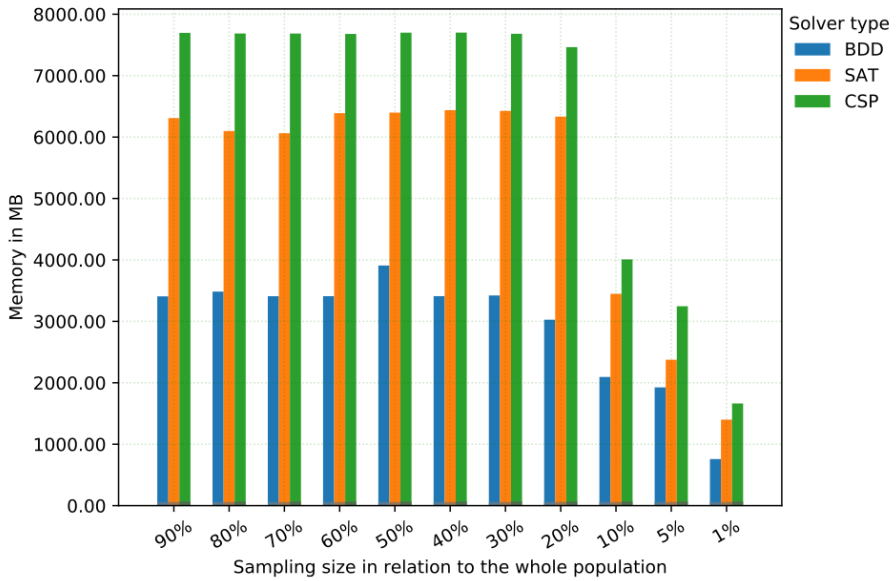


Figure A.28: Comparison of memory consumption, for sampling subsets of the mandatory and alternative artificial feature model with 30 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

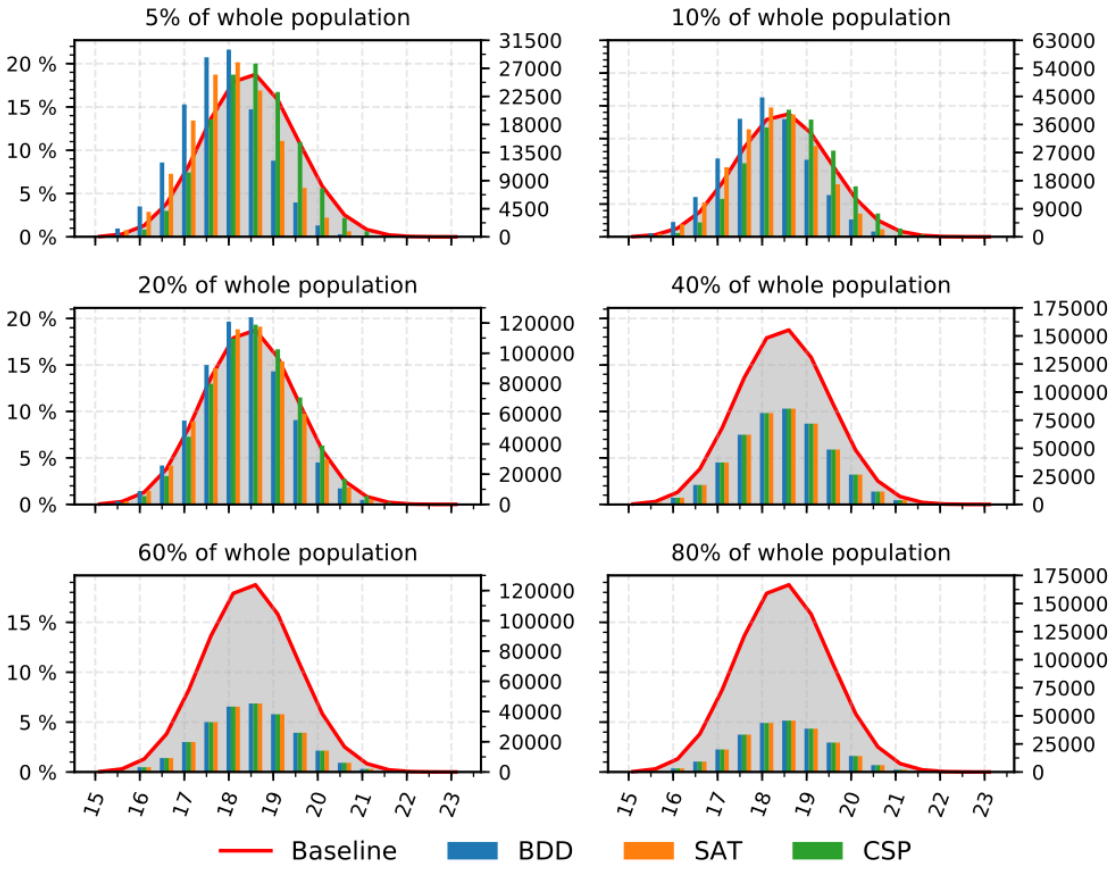


Figure A.29: Comparison of the cardinal distribution, for the sampling results of the mandatory and alternative feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

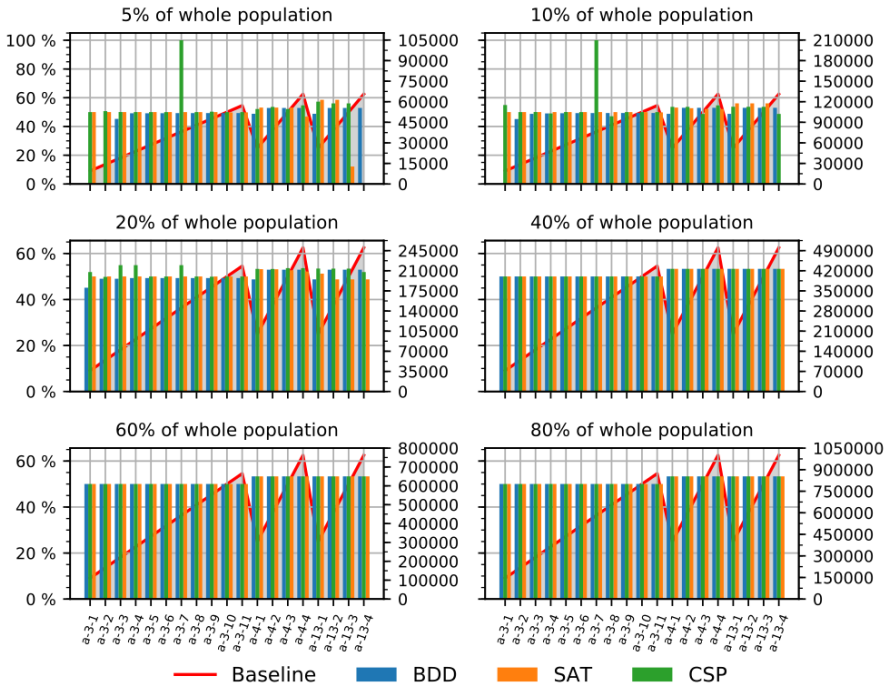


Figure A.30: Comparison of the feature frequency, for the sampling results of the mandatory and alternative feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

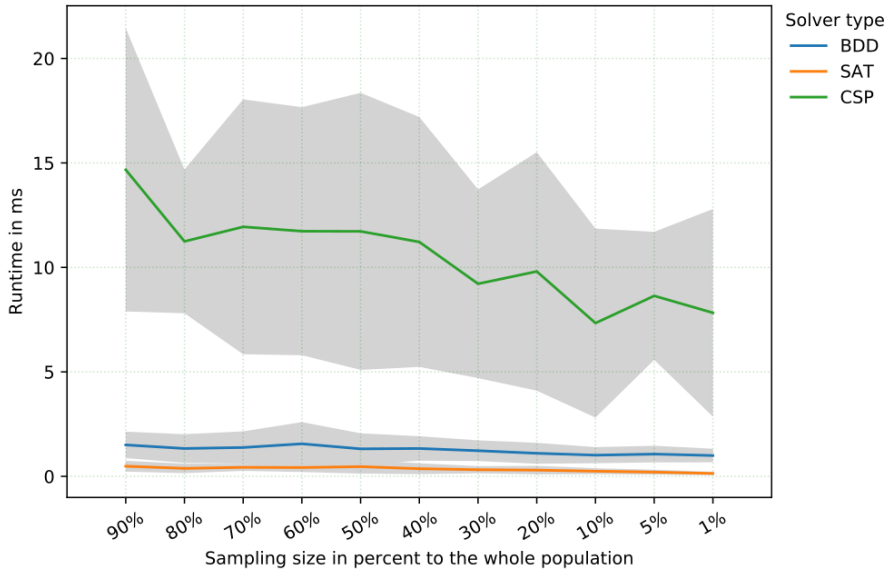


Figure A.31: Runtime performance comparison, for sampling of configurations from the mandatory and exclusive artificial feature model with 20 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

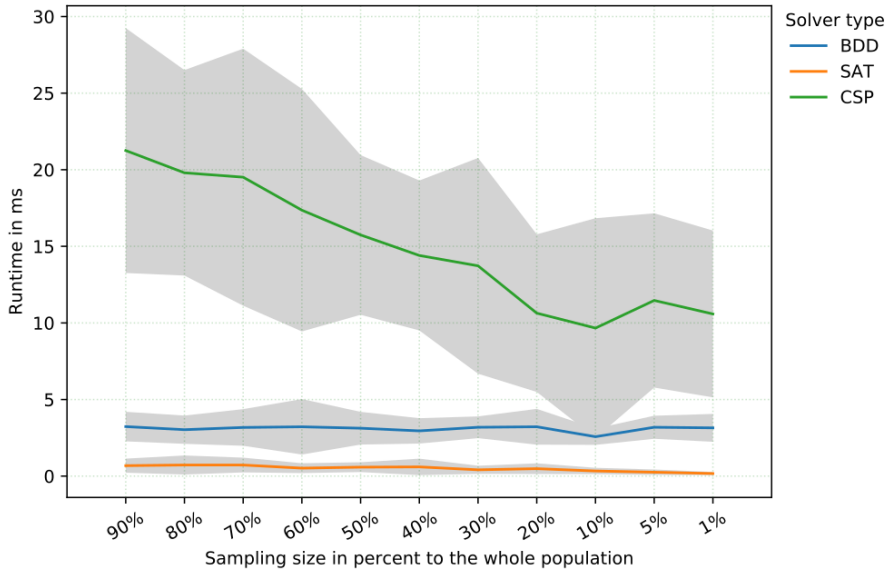


Figure A.32: Runtime performance comparison, for sampling of configurations from the mandatory and exclusive artificial feature model with 30 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

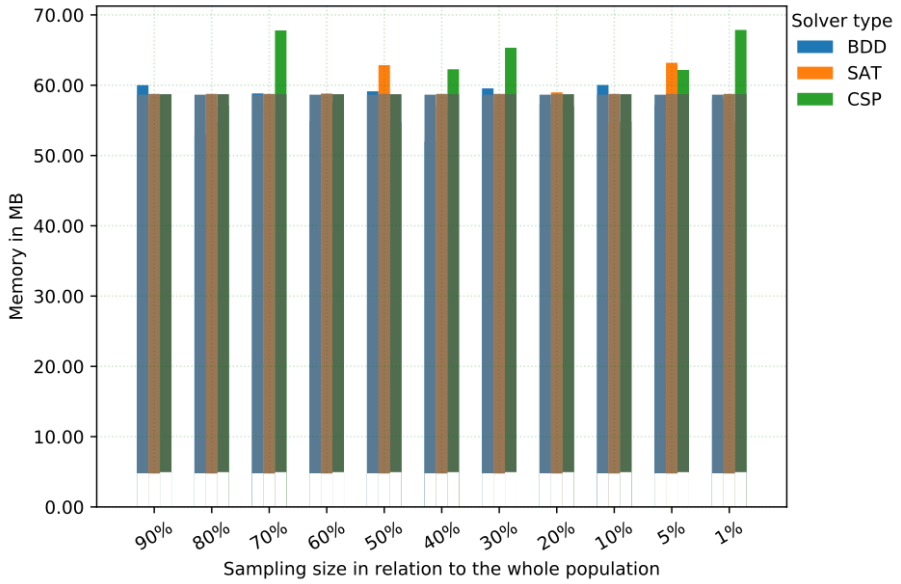


Figure A.33: Comparison of memory consumption, for sampling subsets of the mandatory and exclusive artificial feature model with 20 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

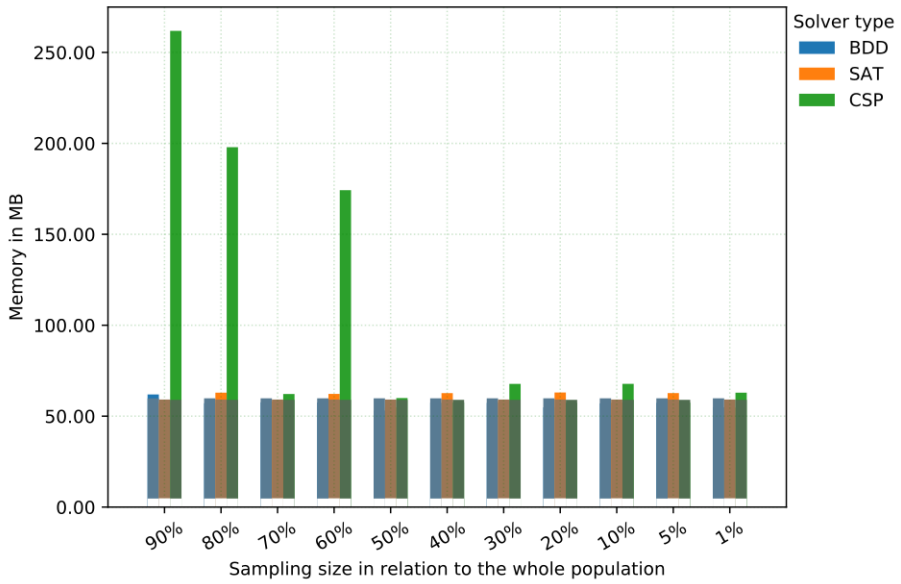


Figure A.34: Comparison of memory consumption, for sampling subsets of the mandatory and exclusive artificial feature model with 30 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

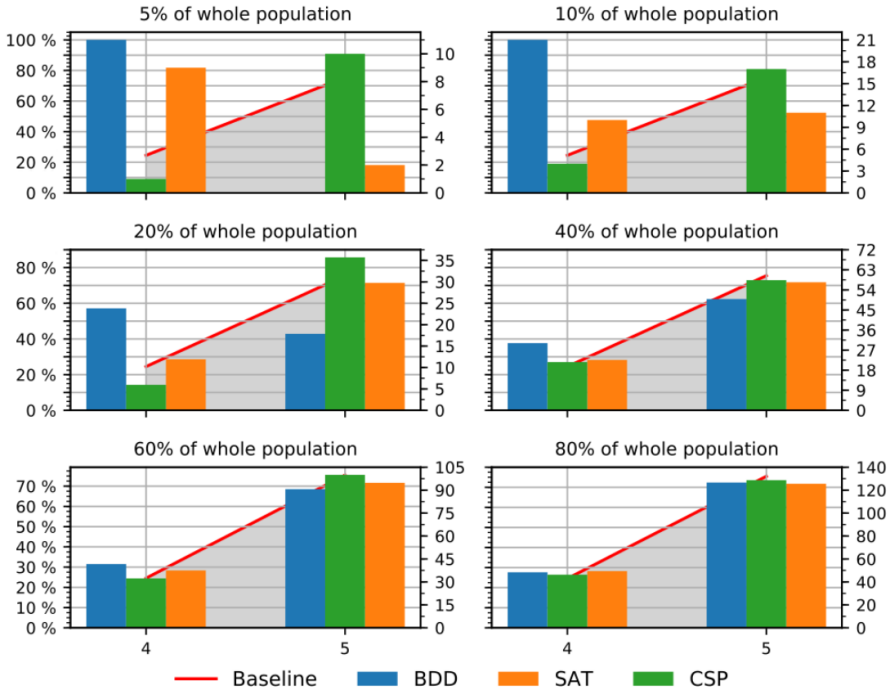


Figure A.35: Comparison of the cardinal distribution, for the sampling results of the mandatory and exclusive feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

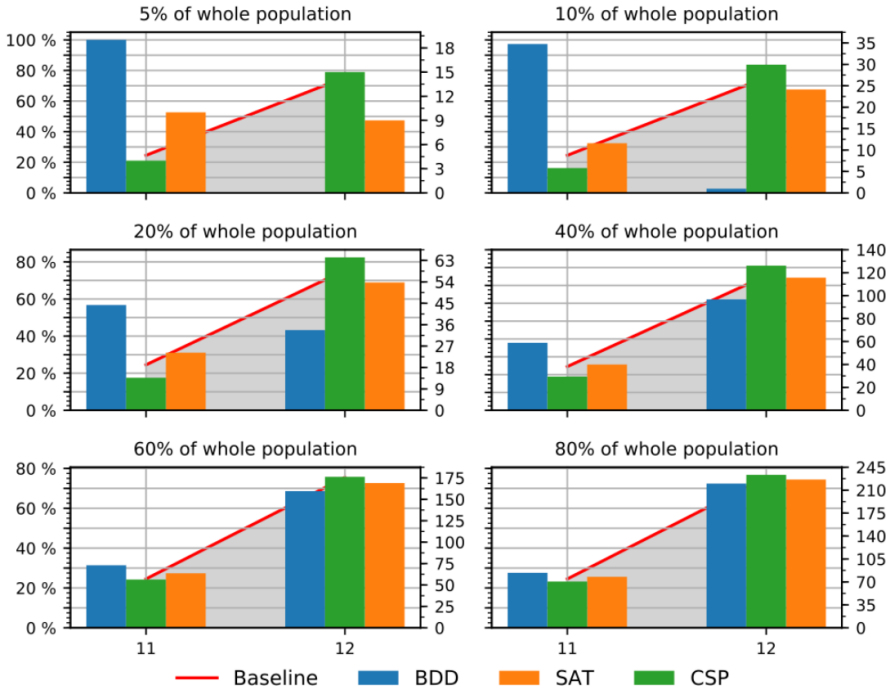


Figure A.36: Comparison of the cardinal distribution, for the sampling results of the mandatory and exclusive feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

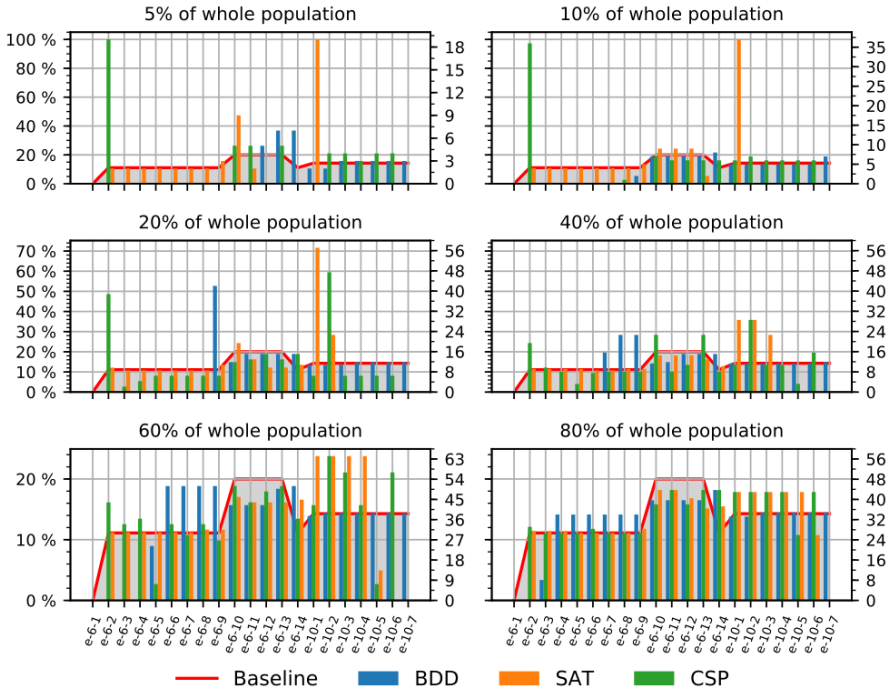


Figure A.37: Comparison of the feature frequency, for the sampling results of the mandatory and exclusive feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

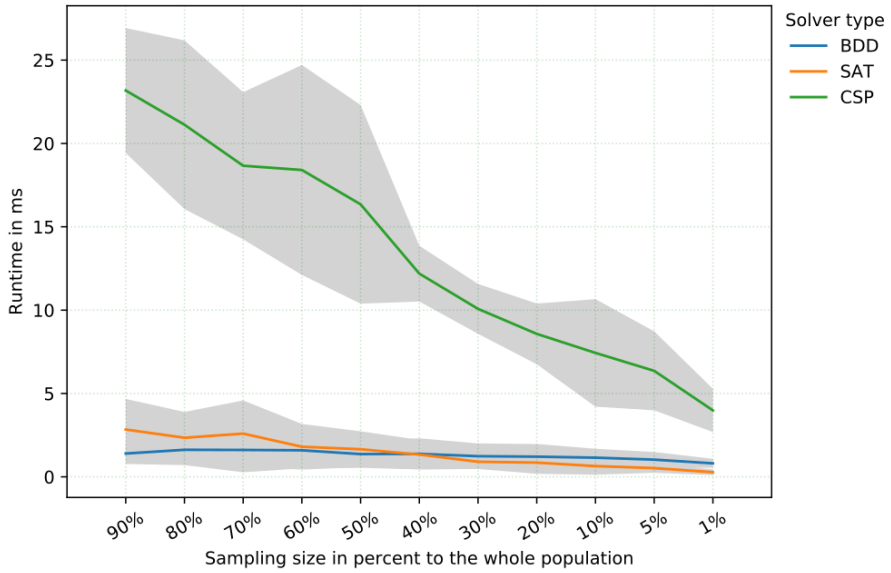


Figure A.38: Runtime performance comparison, for sampling of configurations from feature tree depth feature model with 20 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

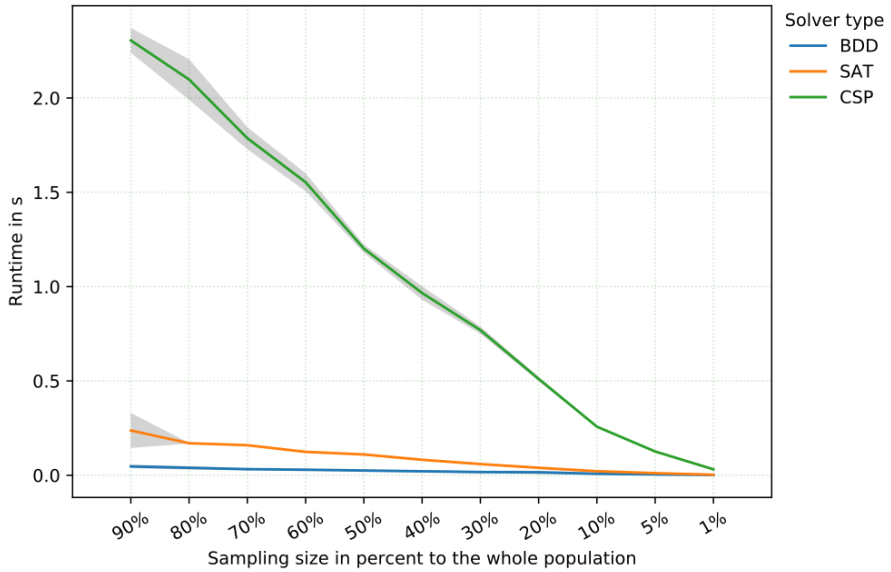


Figure A.39: Runtime performance comparison, for sampling of configurations from feature tree depth feature model with 30 configuration options. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

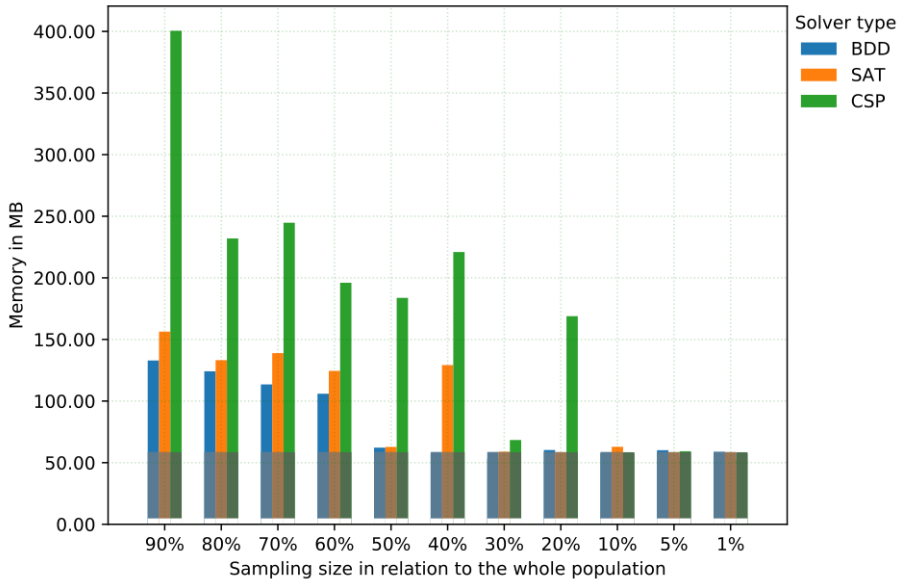


Figure A.40: Comparison of memory consumption, for sampling subsets of the tree depth artificial feature model with 20 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

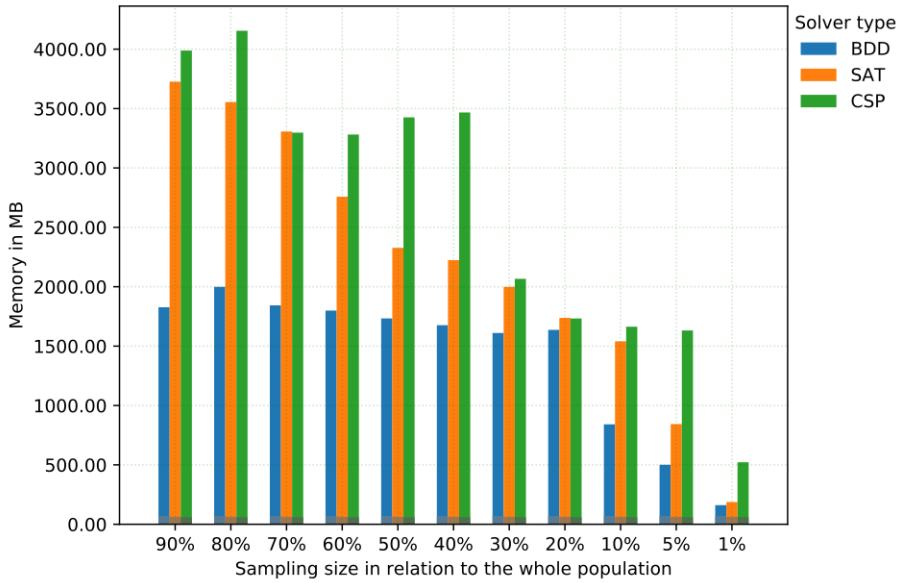


Figure A.41: Comparison of memory consumption, for sampling subsets of the tree depth feature model with 30 configuration options. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

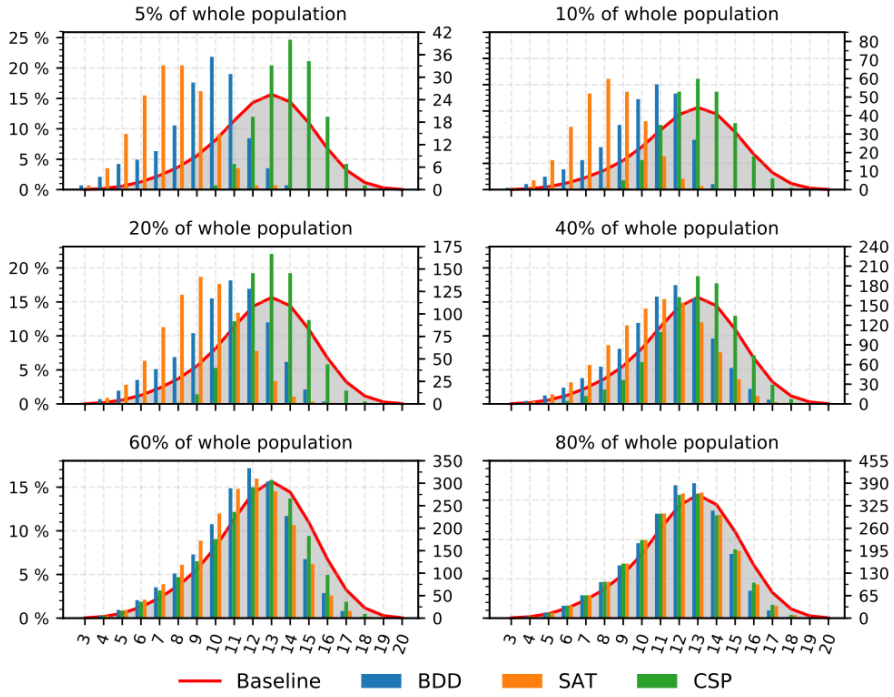


Figure A.42: Comparison of the cardinal distribution, for the sampling results of the tree depth feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

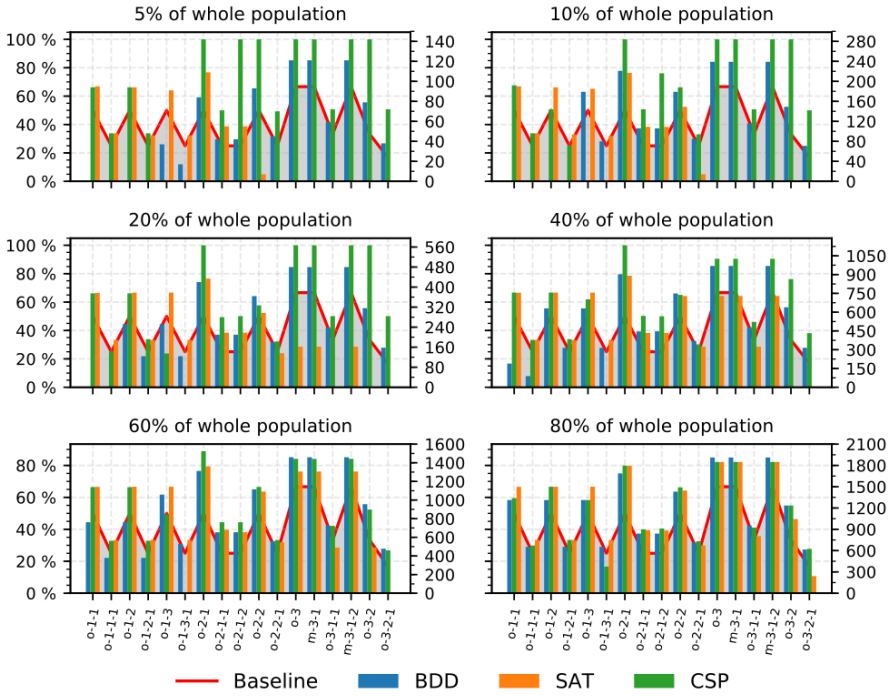


Figure A.43: Comparison of the feature frequency, for the sampling results of the tree depth feature model with 20 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

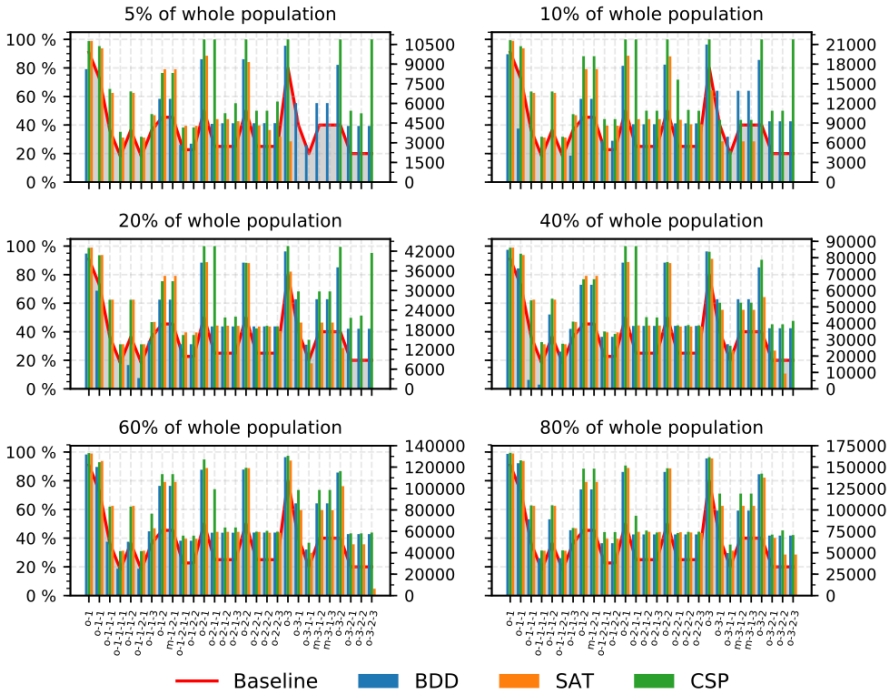


Figure A.44: Comparison of the feature frequency, for the sampling results of the tree depth feature model with 30 configuration options. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

A.3 Real World Feature Models

A.3.1 Solving the #SAT problem

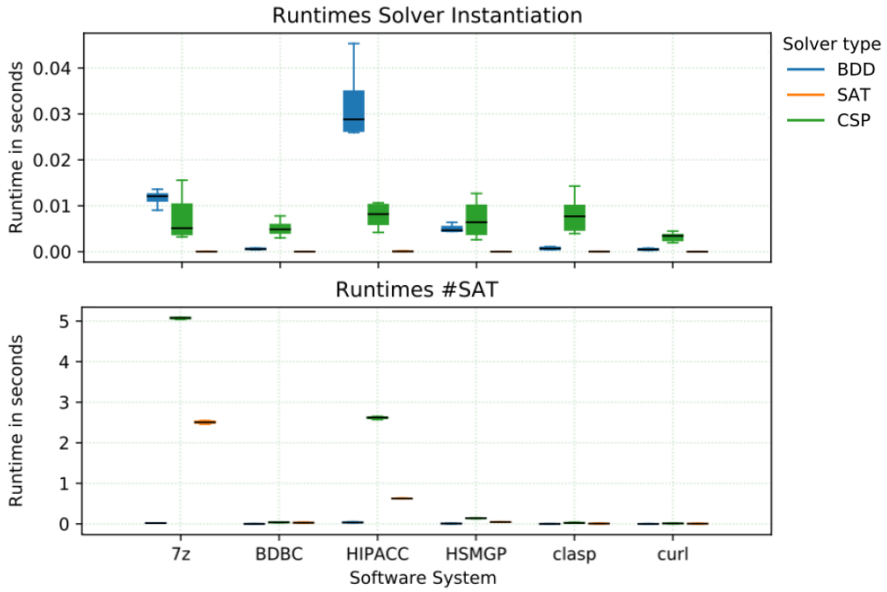


Figure A.45: Runtime performance comparison, for exact #SAT without TriMesh. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

A.3.2 Sampling

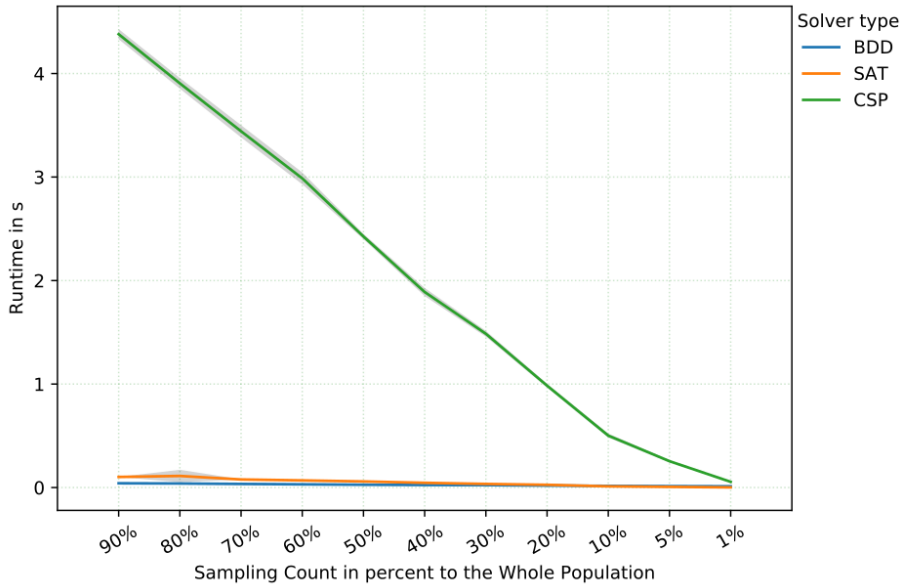


Figure A.46: Runtime comparison for sampling subsets of the 7z feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

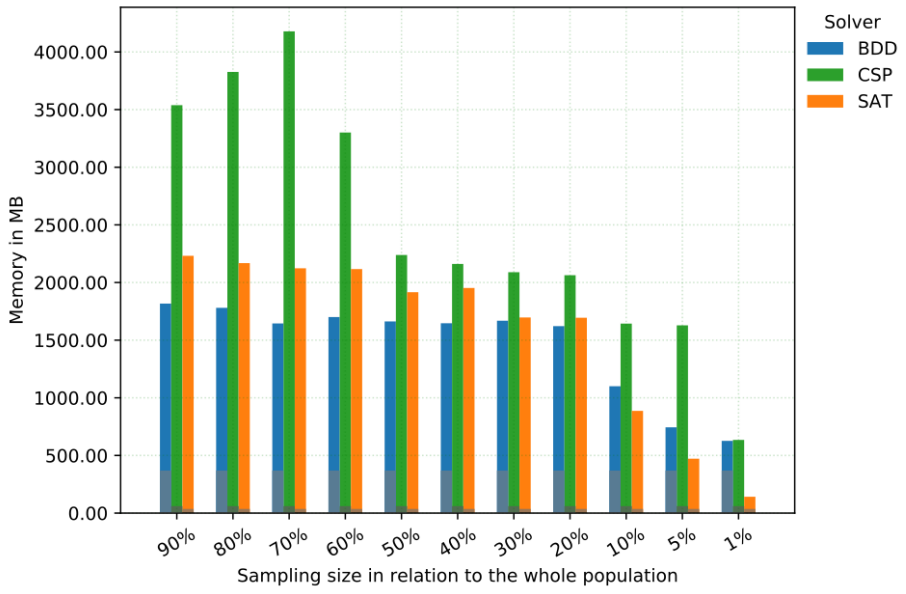


Figure A.47: Comparison of memory consumption, for sampling subsets of the 7z feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

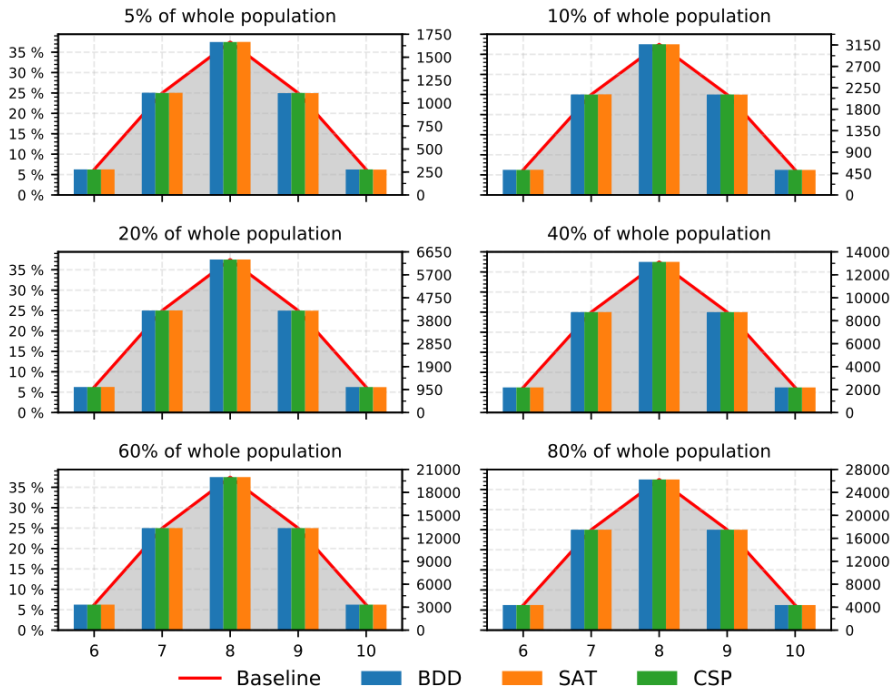


Figure A.48: Comparison of the cardinal distribution, for the sampling results of the 7z feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

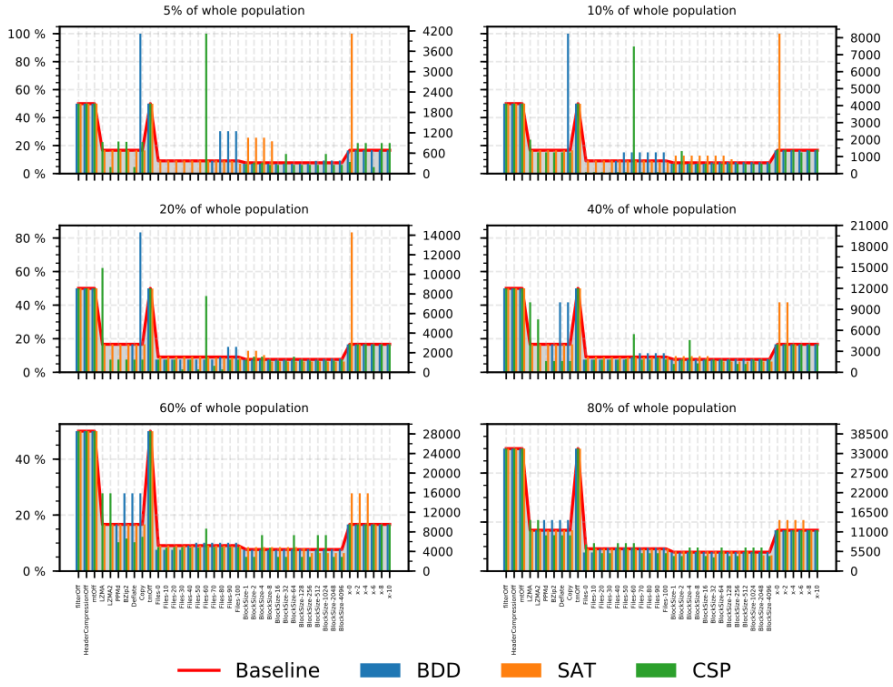


Figure A.49: Comparison of the feature frequency, for the sampling results of the 7z feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

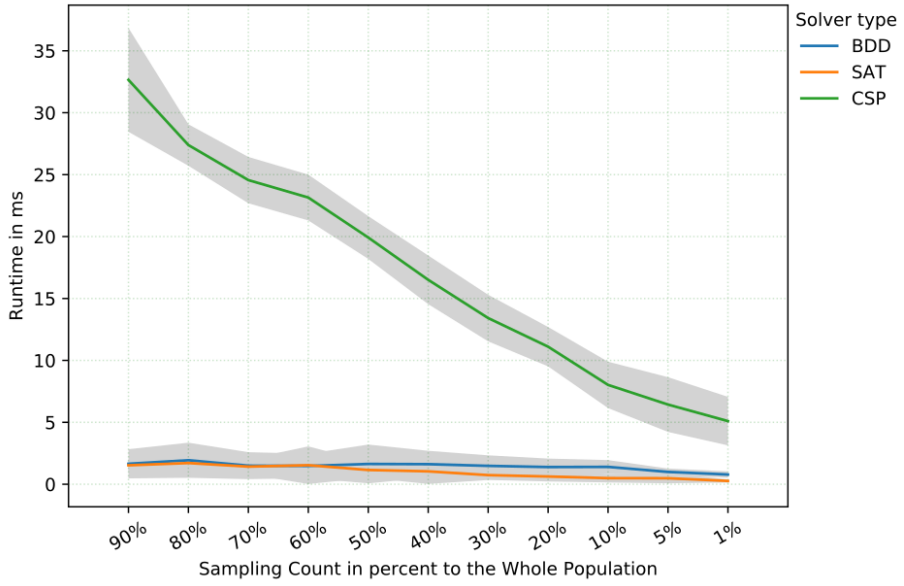


Figure A.50: Runtime performance comparison, for sampling subsets of the BerkeleyDBC feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

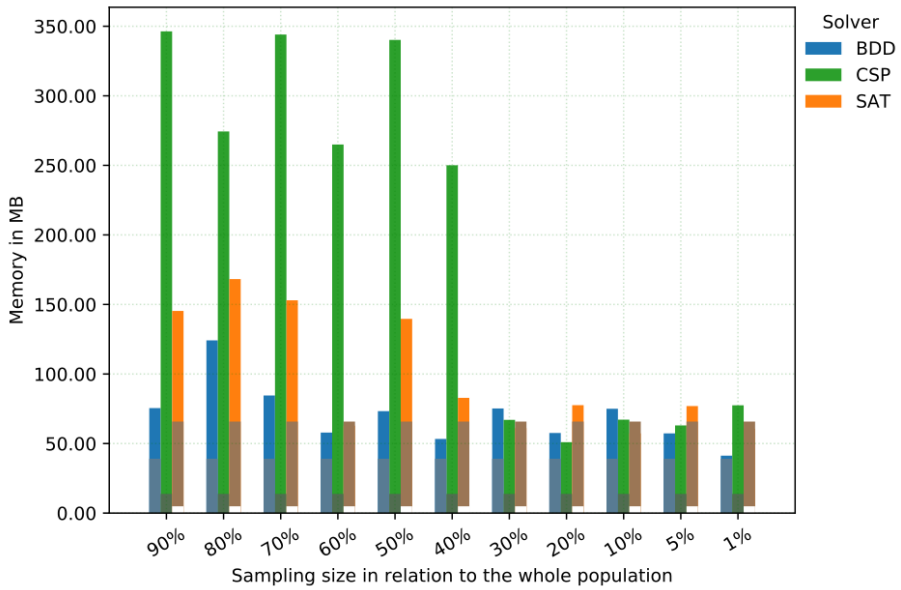


Figure A.51: Comparison of memory consumption, for sampling subsets of the BerkeleyDBC feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

Figure A.52: Comparison of the cardinal distribution, for the sampling results of the BerkeleyDBC feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

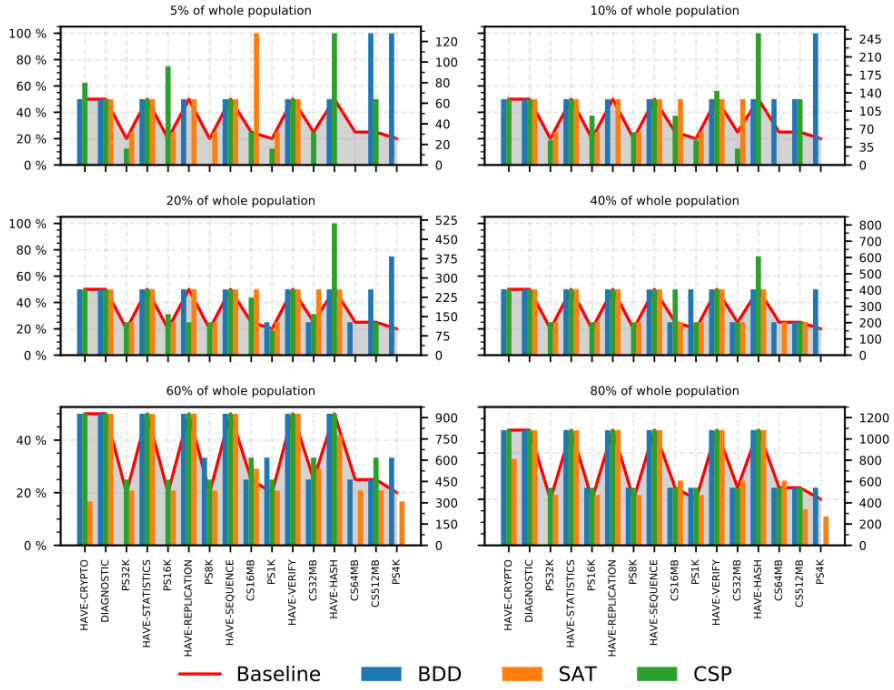


Figure A.53: Comparison of the feature frequency, for the sampling results of the BerkeleyDBC feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

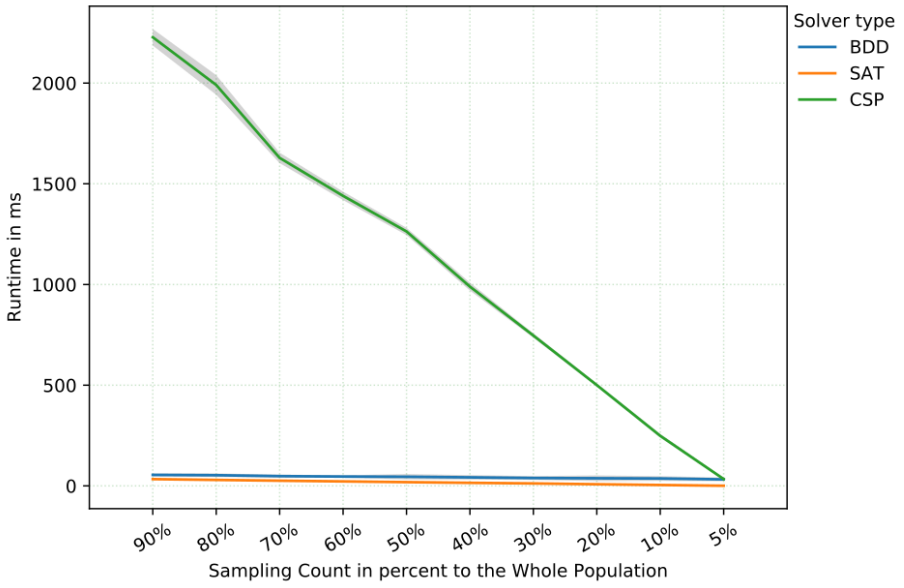


Figure A.54: Runtime performance comparison, for sampling subsets of the $HIPA^{CC}$ feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

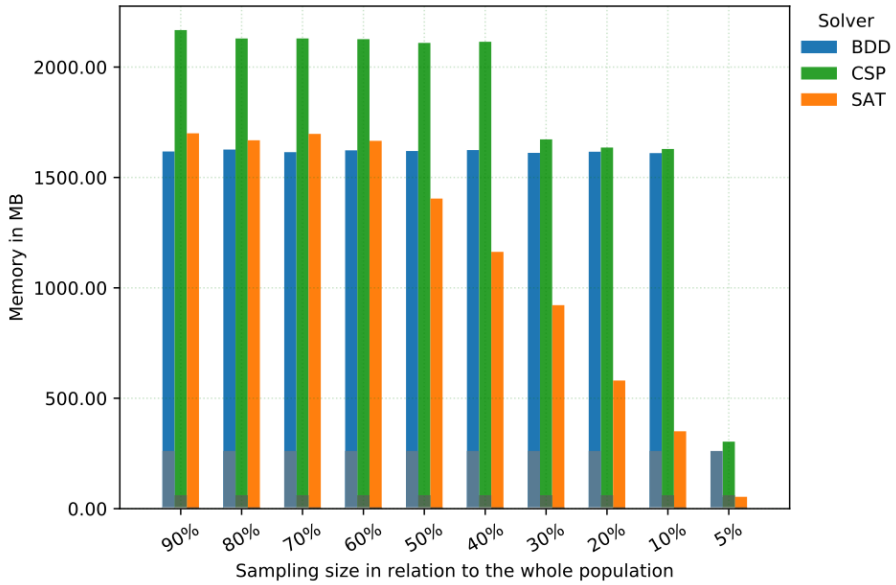


Figure A.55: Comparison of memory consumption, for sampling subsets of the $HIPA^{CC}$ feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

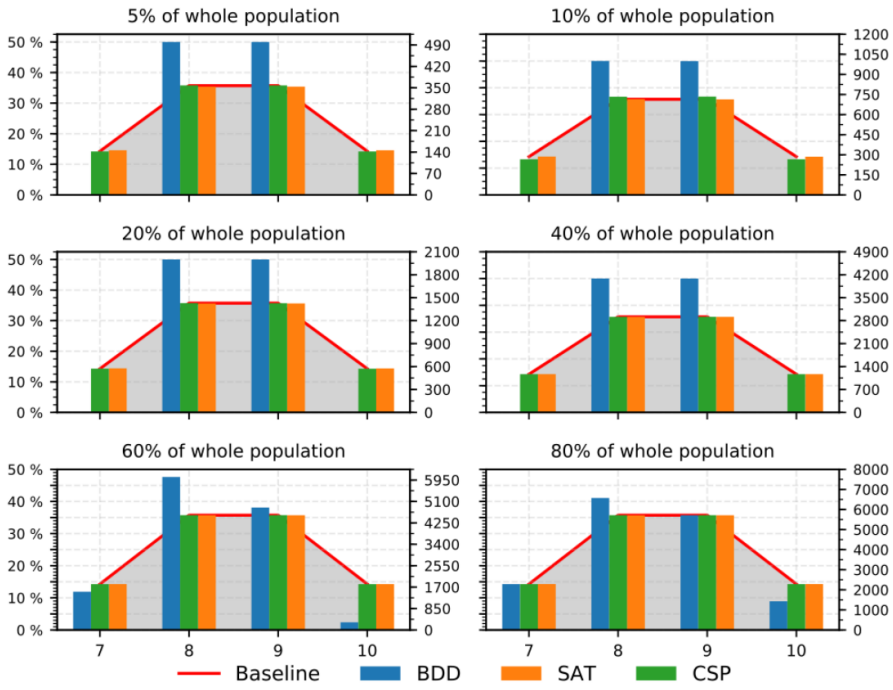


Figure A.56: Comparison of the cardinal distribution, for the sampling results of the $HIPA^{CC}$ feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

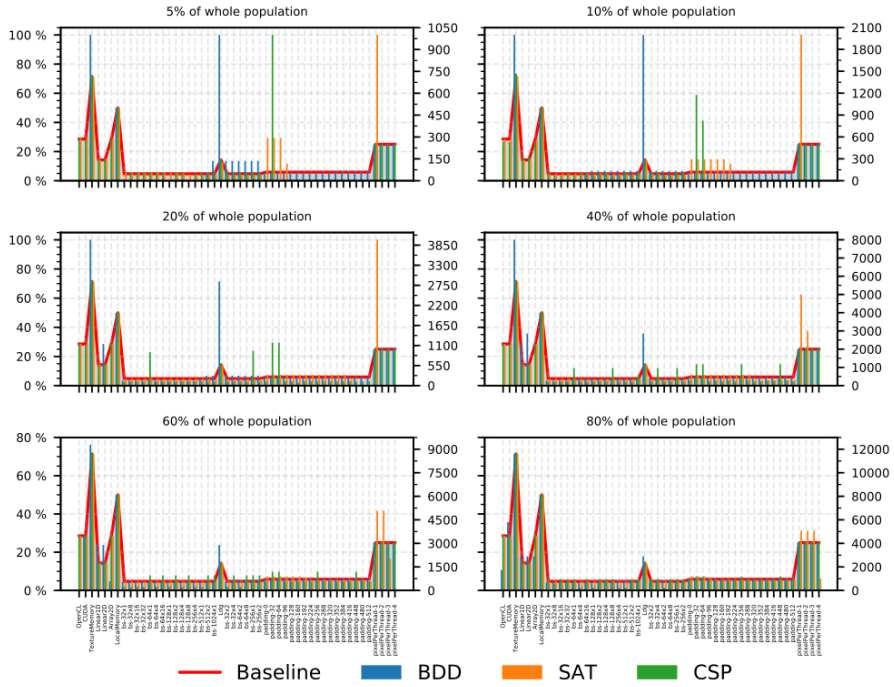


Figure A.57: Comparison of the feature frequency, for the sampling results of the $HIPA^{CC}$ feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

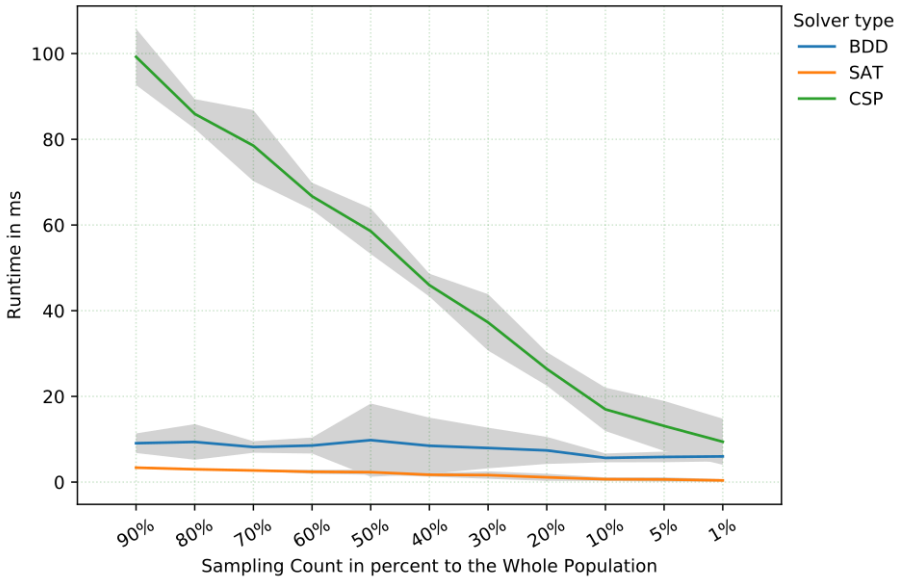


Figure A.58: Runtime performance comparison, for sampling subsets of the HSMGP feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

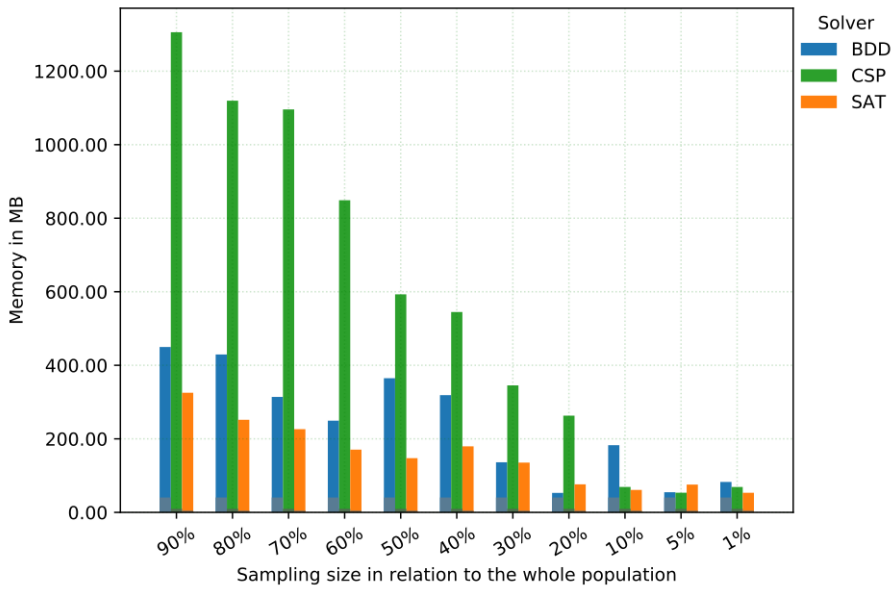


Figure A.59: Comparison of memory consumption, for sampling subsets of the HSMGP feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

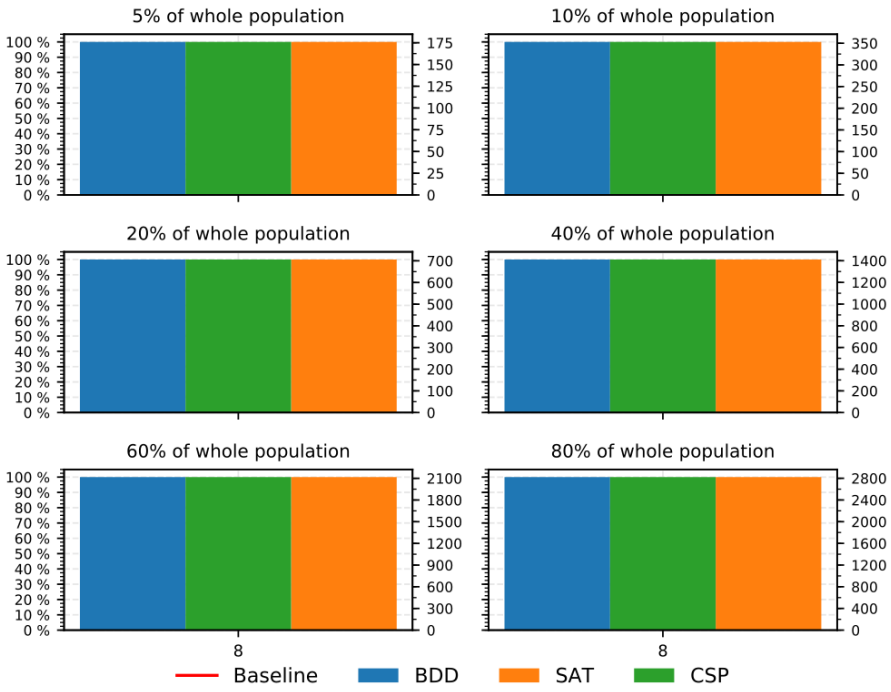


Figure A.60: Comparison of the cardinal distribution, for the sampling results of the HSMGP feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

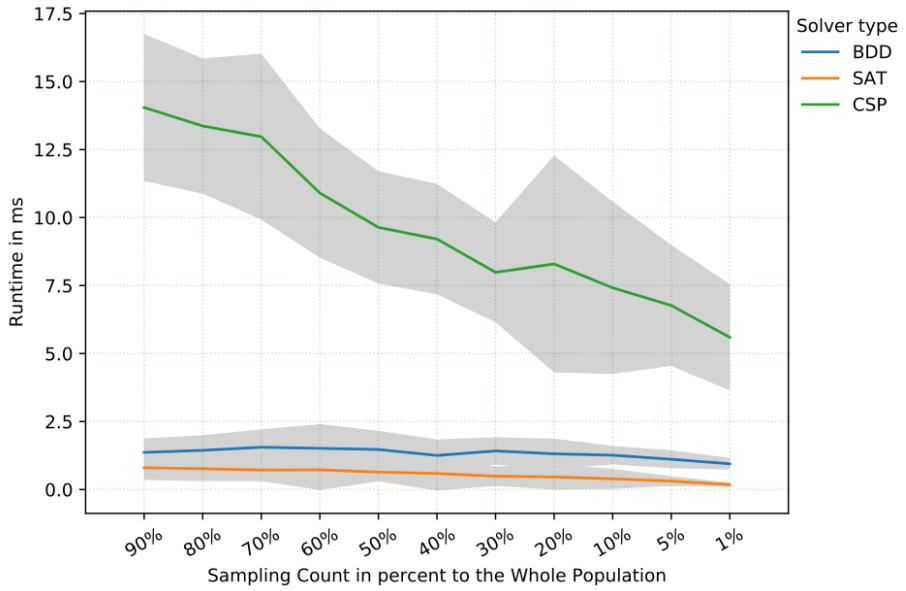


Figure A.61: Runtime performance comparison, for sampling subsets of the Clasp feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

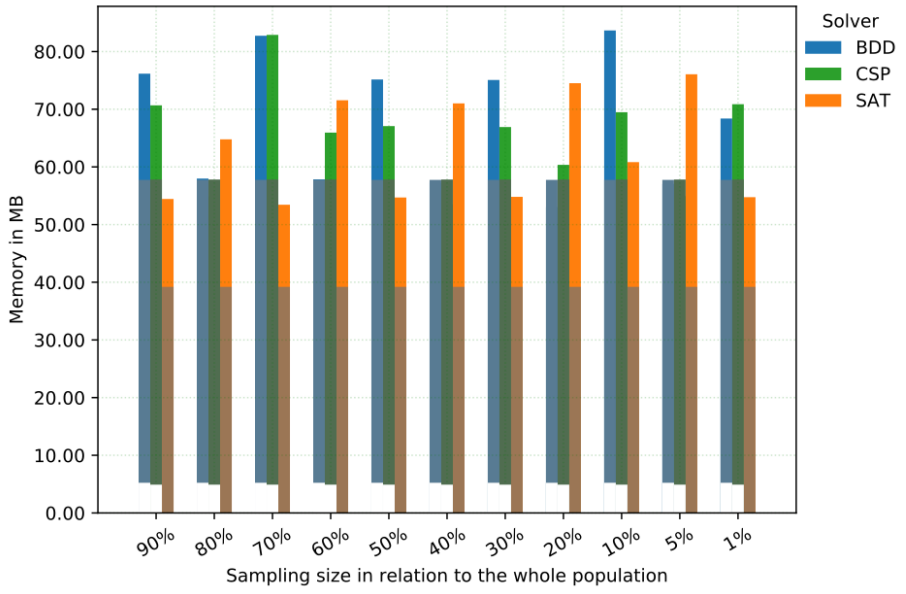


Figure A.62: Comparison of memory consumption, for sampling subsets of the Clasp feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

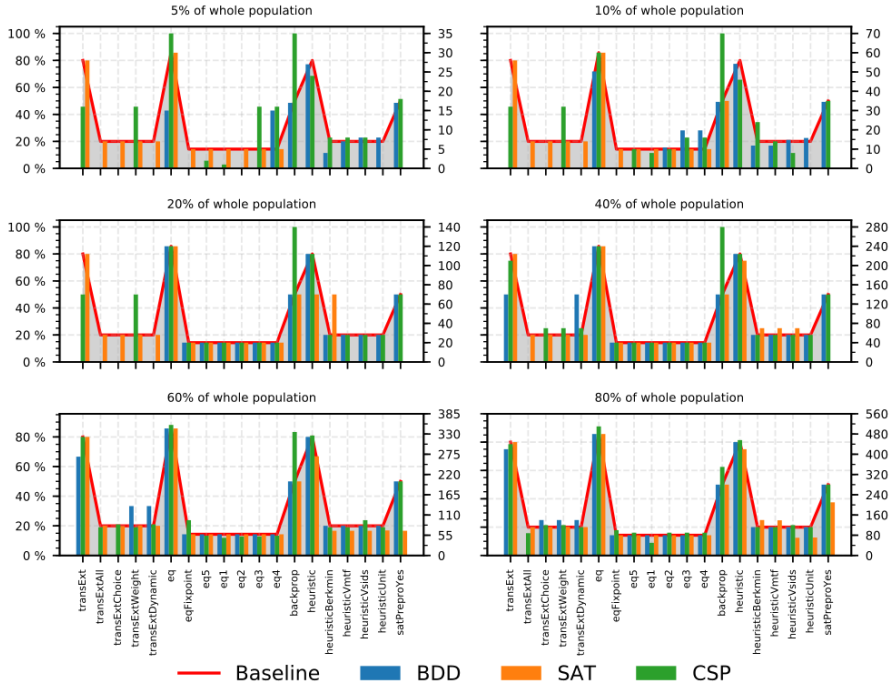


Figure A.63: Comparison of the feature frequency, for the sampling results of the Clasp feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

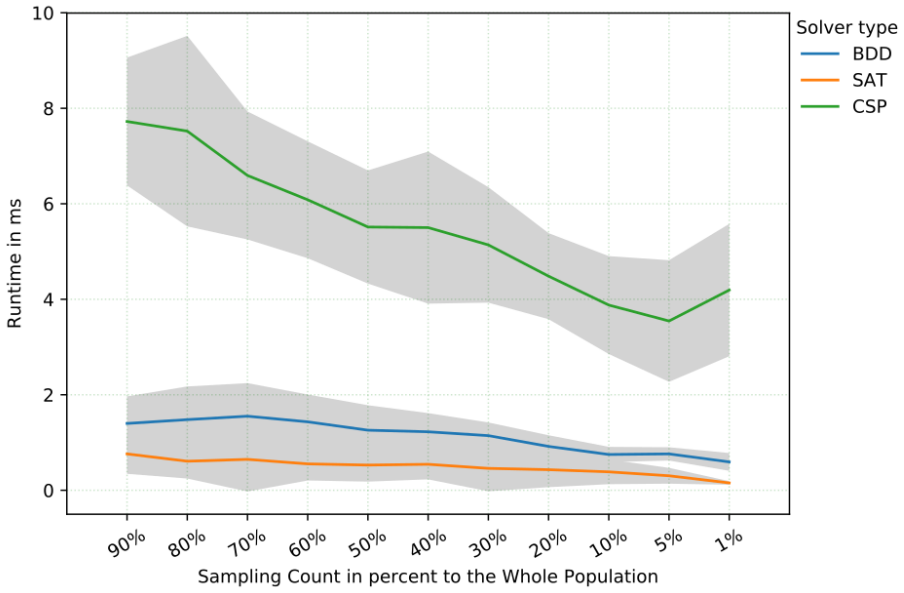


Figure A.64: Runtime performance comparison, for sampling subsets of the Curl feature model. The runtime is shown per solver, with a decreasing number of configurations to sample. Each line of the solver runtime are surrounded by the variance shown in gray. If no runtime variance is shown, the runtime variances are to small, compared to the mean runtime and therefore omitted.

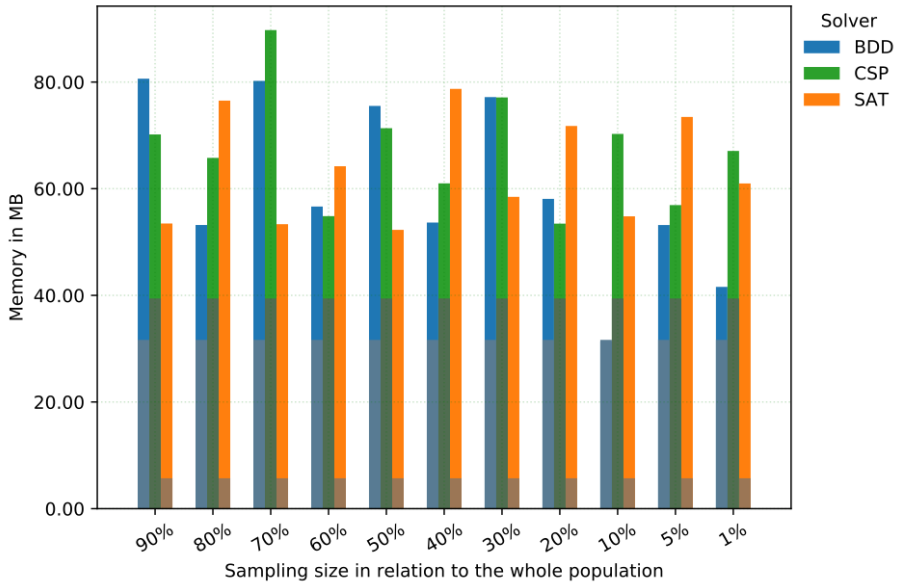


Figure A.65: Comparison of memory consumption, for sampling subsets of the Curl feature model. The consumption is shown per solver and sampling size in relation to the whole population. Each bar combination shows the memory consumption for the different steps. The darker bar parts show the memory consumption for the solver initialization, the lighter bar parts show the memory consumption for sampling.

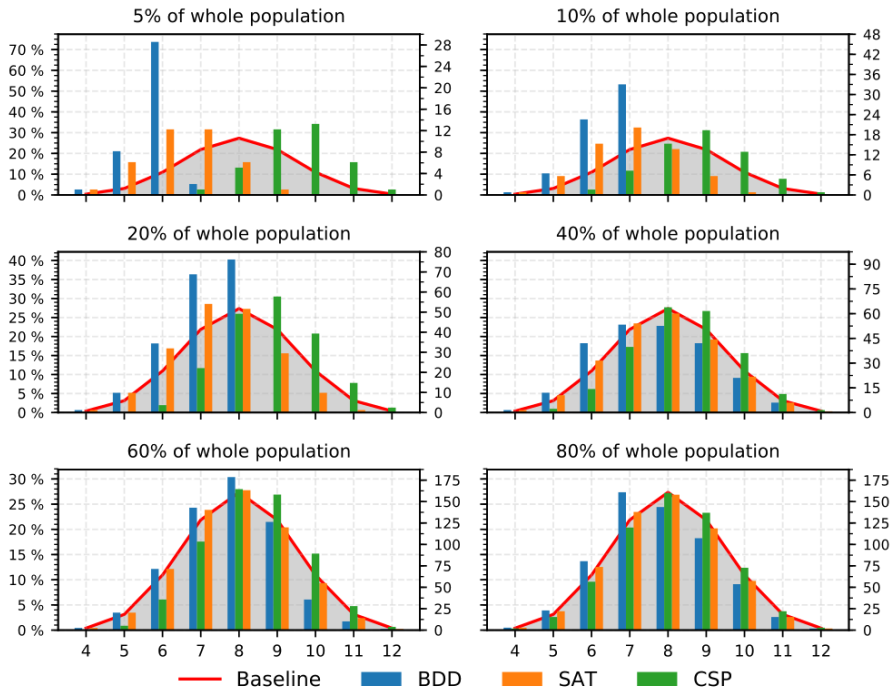


Figure A.66: Comparison of the cardinal distribution, for the sampling results of the Curl feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

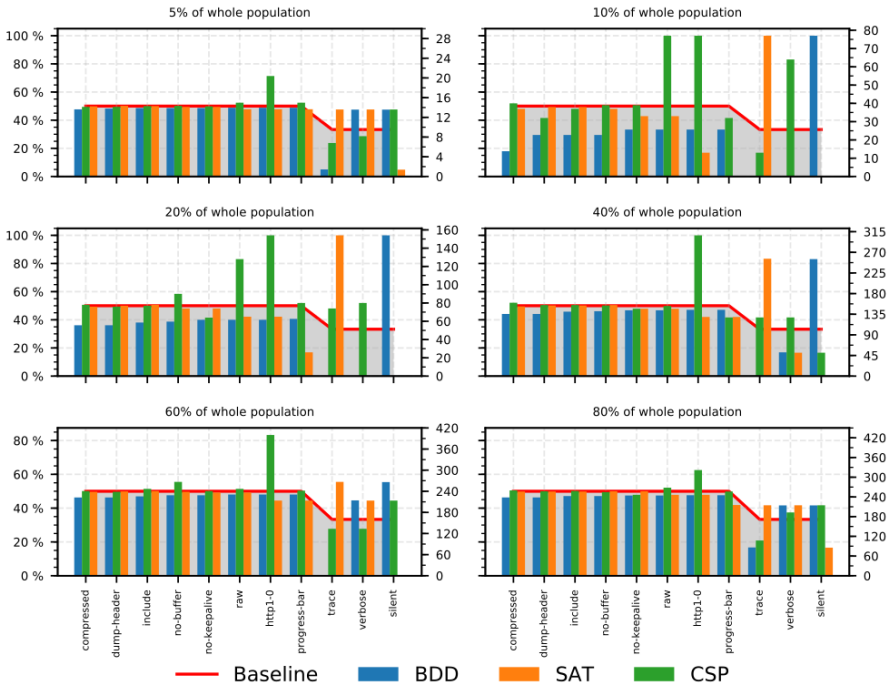


Figure A.67: Comparison of the feature frequency, for the sampling results of the Curl feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

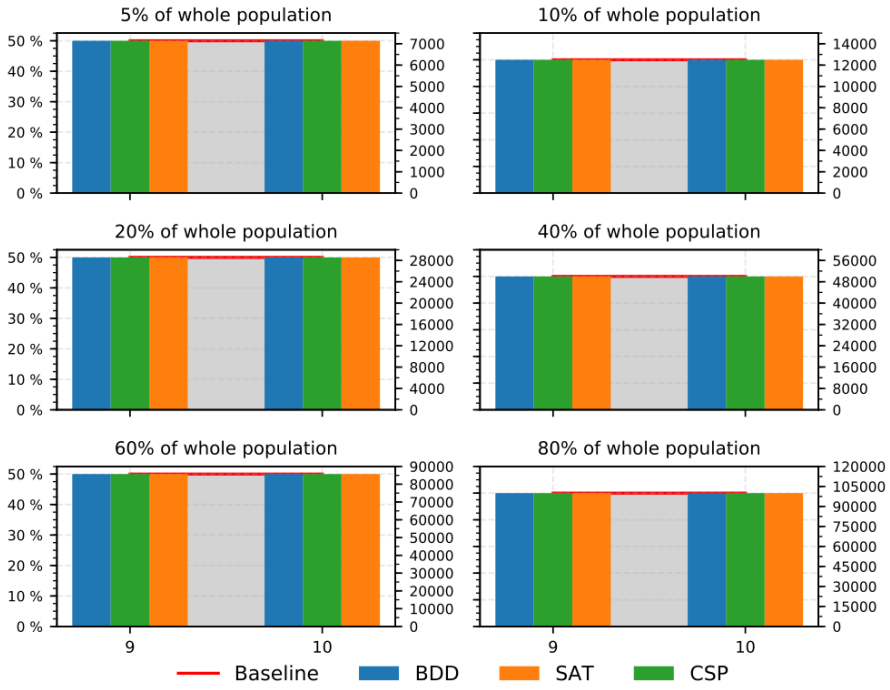


Figure A.68: Comparison of the cardinal distribution, for the sampling results of the TriMesh feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific number of selected configuration options are shown with a bar indicating the percentual number of configurations with this length, within the sampled subset. On the opposite side we show the absolute number of configurations, with this length within the sampled subset.

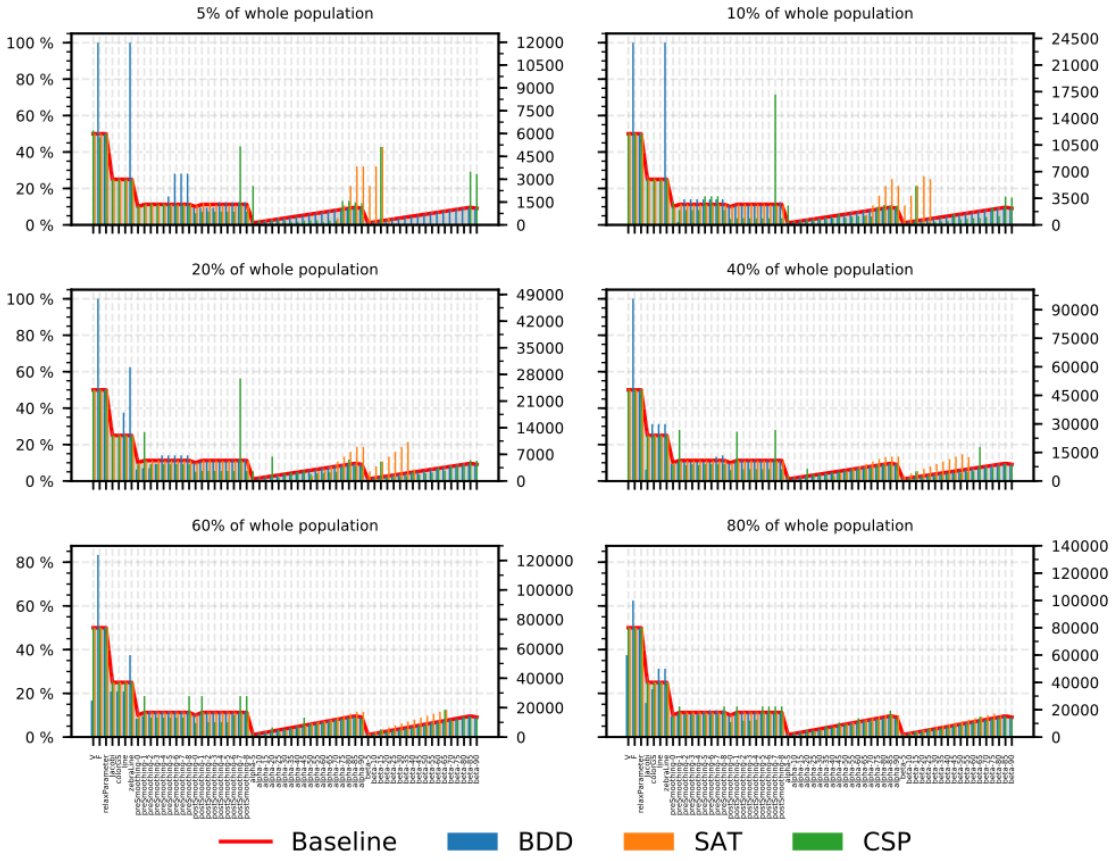


Figure A.69: Comparison of the feature frequency, for the sampling results of the TriMesh feature model. Shown are 6 subsets of sampled configurations in relation to the whole population. The appearance of a specific configuration option are shown with a bar indicating the percentual number of its usage within the sampled subset. On the opposite side we show the absolute number of configurations, using this configuration option within the sampled subset.

Bibliography

[ABKS16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2016. (cited on Page 1)

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. (cited on Page 9)

[Bry92] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992. (cited on Page 9 and 10)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. (cited on Page 1 and 49)

[BSTC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. *VaMoS*, 2007:01, 2007. (cited on Page 50)

[BSTRC06a] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47, 2006. (cited on Page 47, 49, and 50)

[BSTRC06b] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. In *Generative and Transformational Techniques in Software Engineering*, pages 399–408. Springer, 2006. (cited on Page 1, 47, 49, and 50)

[BTC05] David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. Using constraint programming to reason on feature models. In *SEKE*, pages 677–682, 2005. (cited on Page 47, 49, and 50)

[Cha93] DIMACS Challenge. Satisfiability: Suggested format. *DIMACS Challenge. DIMACS*, 1993. (cited on Page 5 and 16)

[CMV16] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. Technical report, Digital Scholarship Archive, 2016. (cited on Page 7)

[DH98] Raphaël Dorne and Jin-Kao Hao. A new genetic local search algorithm for graph coloring. In *International Conference on Parallel Problem Solving from Nature*, pages 745–754. Springer, 1998. (cited on Page 6)

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003. (cited on Page 8 and 9)

[FW74] Jay P. Fillmore and S. G. Williamson. On backtracking: A combinatorial description of the algorithm. *SIAM Journal on Computing*, 3(1):41–55, 1974. (cited on Page 8)

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007. (cited on Page 18)

[GHSS07] Carla P Gomes, Joerg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI*, pages 2293–2299, 2007. (cited on Page 6)

[GK99] Vladimir Gurvich and Leonid Khachiyan. On generating the irredundant conjunctive and disjunctive normal forms of monotone boolean functions. *Discrete Applied Mathematics*, 96:363–373, 1999. (cited on Page 5)

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007. (cited on Page 20)

[GSS06] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *AAAI*, pages 54–61, 2006. (cited on Page 6)

[GSS08] Carla P Gomes, Ashish Sabharwal, and Bart Selman. *Model counting*, chapter 20. Citeseer, 2008. (cited on Page 6, 7, and 49)

[HT08] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 12–21. IEEE, 2008. (cited on Page 4)

[Jan06] Gradus Janssen. Design of a pointerless bdd package, January 31 2006. US Patent 6,993,732. (cited on Page 9)

[KB07] Markus Kalisch and Peter Bühlmann. Estimating high-dimensional directed acyclic graphs with the pc-algorithm. *Journal of Machine Learning Research*, 8(Mar):613–636, 2007. (cited on Page 9)

[KGKR13] Sebastian Kuckuk, Björn Gmeiner, Harald Köstler, and Ulrich Rüde. A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids. In *PARCO*, pages 813–822, 2013. (cited on Page 20)

[KK02] Lee Kwanwoo and C Kyo. Concepts and guidelines of feature modeling for product line software egnineering. In *ICSR*, pages 62–77, 2002. (cited on Page 4)

[Knü16] Alexander Knüppel. The role of complex constraints in feature modeling. Master's thesis, Institute of Software Engineering and Automotive Informatics at Technische Universität Carolo-Wilhelmina zu Braunschweig, 07 2016. (cited on Page 1, 5, and 16)

[KU02] Boonserm Kijsirikul and Nitiwut Ussivakul. Multiclass support vector machines using adaptive directed acyclic graph. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 1, pages 980–985. IEEE, 2002. (cited on Page 9)

[LBP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. (cited on Page 9 and 16)

[Lie15] Jörg Liebig. *Analysis and Transformation of Configurable Systems.* PhD thesis, University of Passau, 2015. (cited on Page 4)

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009. (cited on Page 5, 16, and 51)

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. (cited on Page 8)

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *AAAI*, volume 92, pages 459–465, 1992. (cited on Page 9)

[OBMS16] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding product line configurations with high performance by random sampling. Technical report, Technical Report TR-16-22. University of Texas at Austin, Department of Computer Science, 2016. (cited on Page 10, 11, 15, 16, and 47)

[OBMS17] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 61–71. ACM, 2017. (cited on Page 15, 16, and 47)

[PFL17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. (cited on Page 8 and 16)

[PLP11] Richard Pohl, Kim Lauenroth, and Klaus Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 313–322. IEEE Computer Society, 2011. (cited on Page 51)

[PSW10] Bernhard Pfahringer, Claude Sammut, and Geoffrey I. Webb. *Conjunctive Normal Form*, pages 209–210. Springer US, Boston, MA, 2010. (cited on Page 5)

[PW94] Hilary A Priestley and Martin P Ward. A multipurpose backtracking algorithm. *Journal of Symbolic Computation*, 18(1):1–40, 1994. (cited on Page 8)

[RNI95] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995. (cited on Page xi, 1, 6, 7, and 8)

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. *SAT*, 4:7th, 2004. (cited on Page 6)

[Sch99] T Schoning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 410–414. IEEE, 1999. (cited on Page 1, 6, and 7)

[Seg08] Sergio Segura. Automated analysis of feature models using atomic sets. In *SPLC (2)*, pages 201–207, 2008. (cited on Page 50)

[SLM+92] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446, 1992. (cited on Page 9)

[Som99] Fabio Somenzi. Binary decision diagrams. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:303–368, 1999. (cited on Page 9)

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable

system software: facing the linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*, pages 47–60. ACM, 2011. (cited on Page 4)

[Vah15] Arash Vahidi. Jdd: a pure java bdd and z-bdd library. https://bitbucket.org/vahidi/jdd, 2015. (cited on Page 16)

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In *SAT*, pages 324–339. Springer, 2005. (cited on Page 6)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Adrian Marten

Passau, den 28.03 2018