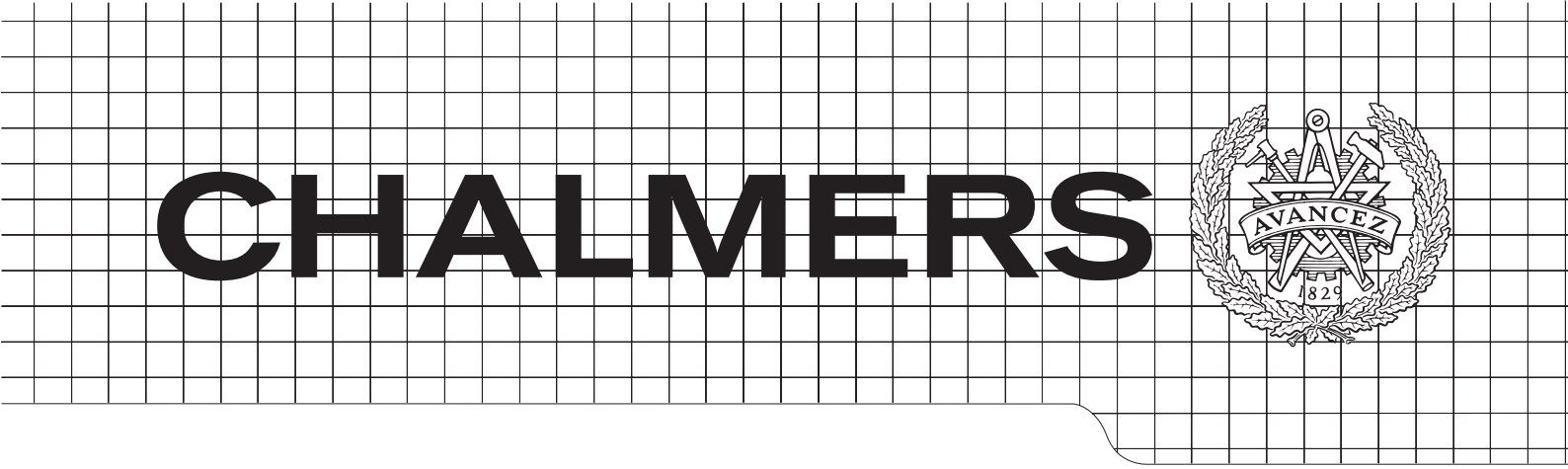


Using CSP solvers for Partial Configuration in Automotive Configuration Problems

Master's Thesis in Intelligent Systems Design(Applied Information Technology)

OXANA SACHENKOVA

SUVASH KESHARI THAPALIYA

Department of Applied Information Technology

Chalmers University of Technology

Göteborg, Sweden, 2011

Using CSP solvers for Partial Configuration in Automotive Configuration Problems

Master's Thesis in Intelligent Systems Design(Applied Information Technology)

Technical report no : 2011:069

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Geometry and Motion Planning
FRAUNHOFER CHALMERS CENTER
SE-412 88 Göteborg
Sweden
Telephone + 46 (0)31-772 4000

Abstract

A constraint satisfaction problem involves the assignment of values to variables subject to a set of constraints. A large variety of problems in artificial intelligence and other areas of computer science can be viewed as a special case of the constraint satisfaction problem. In many applications, one example being product design and configuration, user interaction is required to find a solution. This thesis project focuses on using CSP solver methods for configuration problems in the automotive industry. The configuration problem encompasses a number of finite variables, describing some variability of a vehicle such as the gearbox or the color of the chassis. A number of restrictions of physical, legal, or strategic nature limit the number of allowed combinations of these variables, lending CSP to be a natural setting. Furthermore, design and configuration in a large scale involves breaking down it into several small manageable design workflows, hence the requirement for being able to design and configure partially. This can be considered a special case of CSP, where partial configurations are solved which still satisfy all the constraints in the bigger set.

Keywords: Constraint Satisfaction Problem, Automotive industry, Configuration Problem, Partial Configuration

Acknowledgements

This dissertation would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, we offer our sincere gratitude to the Department of Geometry and Motion Planning at the Fraunhofer Chalmers Center, that provided us with the initiative and the resources all the way through the thesis project. We are heartily thankful to our supervisor at the center, Fredrik Ekstedt, whose guidance and support helped us reach our goals. One simply could not wish for a better supervisor.

We would then like to thank our academic supervisor, Claes Strannegård, who has helped us with discussions and clarifications on the subject now and then. While enabling us to develop a good understanding of the subject, we are also grateful of him for re-aligning us along the thesis project when deemed necessary.

Further along, we would like to thank Alexey Voronov, who provided us the necessary data, the explanation behind it and all the discussions regarding to the experiments and knowledge that was already built and being build over it.

We feel blessed having a wonderful bunch of friends and colleagues, who have provided us with necessary dose of laughter and encouragement throughout the thesis project. It would be impossible to think about getting through the project without all the friendly support, especially when the going went tougher.

Finally, we would like to thank our families, who have inspired us all the way through. We are forever indebted to the support and love that has helped us get this far ahead.

The Authors,
Department of Geometry and Motion planning
Fraunhofer Chalmers Center
Göteborg, Sweden
5th July 2011

Contents

List of Figures

List of Tables

1

Introduction

In the automobile industry, designers often require the flexibility of being able to quickly see the results of various individual changes in design. For example, it would be really helpful for a designer if s/he could possibly see what different types of engines could be used given the current chassis and transmission selection. While given that an automobile can be designed with different engines, transmissions and other various individual parts, all combinations of the individual parts are not possible. This is due to the fact that for example, not every engine works with every transmission. These rules in the form of constraints express the relation between various individual parts. The cause for these constraints may be due to, for example, engineering (e.g. geometrical or strength), legal, or marketing considerations. Representation of such configuration problems and problem solving strategies is surveyed in (Sabin and Weigel, 1998;).

A configurator is a software tool that is able to analyze the constraints in different ways. The use of software configurators started within the computer industry

with McDermott's development of the R1 configurer (McDermott, 1982) and is now widely used when the products are complex and can be configured in multiple ways. Configurators are commonly used in the sales process (Haag, 1998), but there is another type of configurators, engineering configurators (Tiihonen et al., 1998). In sales, a customer or a salesman interactively composes a product, while the configurator guarantees that an undeliverable product will never be configured. Deliverable products are typically described using logic constraints over a set of variables, where each variable has a given domain. The task of an engineering configurator, on the other hand, is to create, debug and maintain configuration constraints as a part of the product data.

This work concentrates on the use of configurators during product development. For large scale organizations, like the automotive companies, multiple product development teams are working on developing the product. Very few, if any, have a complete overview of the full product and the constraints that must be satisfied. Each product development team might only be interested in a small subset of all variables used to describe the full product. One use of a configurator is to compute the set of allowed combinations for such a subset of the variables. In doing this the configurator must, in general, take all constraints into account. This is a complicated task given the size of the problem. An example from the car industry has 200 variables. Thus naive approaches to solve the configuration problem will fail for all but the smallest problems. Another fact that contributes to the complexity is that in large scale product development applications, constraints are changing constantly; rules are added, deleted, and modified. Thus, a configurer must be able to handle new situations without any manual tuning of its algorithms.

1.1 Problem formulation

This Master Thesis project focuses on using CSP solver methods for configuration problems in the automotive industry. The configuration problem in consideration encompasses a number of finite variables, describing some variability of a vehicle. The variability could be explained by examples as in the type of the engine is use or the color of the chassis. Furthermore, a number of restrictions are imposed on the variables. These restrictions can be of of physical, legal, or strategic in nature thus limiting the number of combinations possible within the variables. Hence the presence of variables with domains and restrictions between the various possible values automatically calls out for CSP solvers to be used as one of the natural choices.

1.2 The Configuration Problem.

1.2.1 Configuration Problems in general

Configuration involves selecting combination of predefined components subject to a number of problem constraints. Configuration problems may involve sales (sales configuration), design, manufacturing, installation, or maintenance. The components involved need not be physical but can also be paragraphs of a legal document, financial services, actions in a plan, etc. The possibility to automate the product configuration task using a configuration system was recognized in the 1980's, and is now a rapidly growing industry. A product configuration problem in design of a car involves many customizable parts or features which can be modeled as a set of variables. Different values can be assigned to each part or feature for a specific

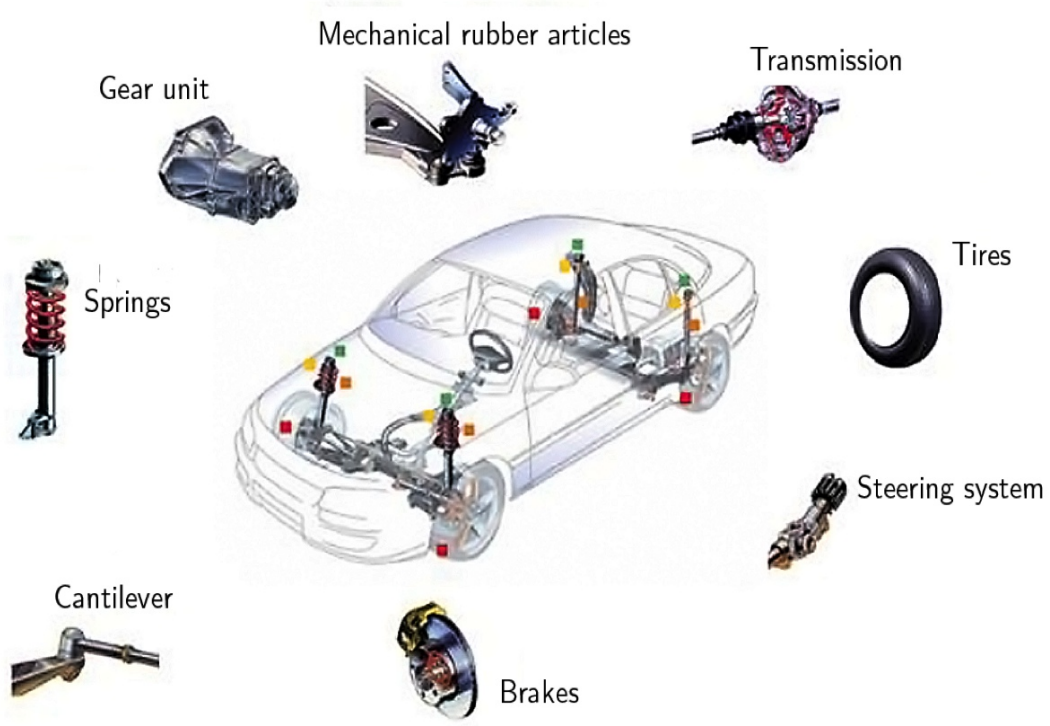


Figure 1.1: Various parts are assembled together for a valid configuration

product. The set of all possible values of a variable forms its domain.

For example, cars can have engines, generators or type of seats. Then the generator can have a power of 120, 140 or 150 AMP. Seats can have values ”sport” or ”custom” and the engine could have a domain of values Engine Diesel, Engine Gas 4 CYL, Engine Gas 6 CYL.

The rules that state which combinations of values are allowed and which cannot go together in our case are represented by propositional formulas over assignments of values to variables. These assignments, like, assigning a value Engine Diesel to the variable engine is in fact a boolean variable or arom, which can be true or false. It will be true if the engine was selected to Engine Diesel and false otherwise.

Configuration rules are formed by combining such atoms in propositional formulas. A propositional formula is constructed from simple propositions, such as "engine is not selected to Engine Diesel" using connectives such as NOT, AND, OR, and IMPLIES; For example, to express that Engine Diesel does not fit together with sport seats, the rule can be constructed as follows:

```
(engine == Engine Diesel) IMPLIES NOT(seats == sport) AND (other options required)
```

This formulation of a Configuration Problem allows us to represent it as a Constraint Satisfaction problem.

1.2.2 Elements of a Configuration System

At a high level, a constraint based configuration system is made up of two parts, as shown in figure below.

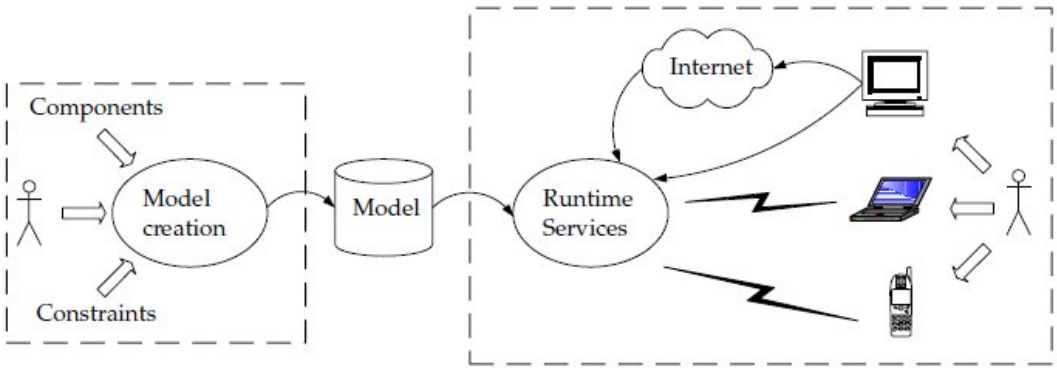


Figure 1.2: A general view of configuration system

1. A modeling part where a configuration model is created. The model can be created manually by a user, automatically by extracting data from various enterprise systems or by a combination of the two approaches.

2. A runtime part where the end users of the configuration system interact with the configuration model to determine the configuration that suit their needs (whether it is a product, a service, or something completely different). The task of selecting the components is called the configuration task.

Creating a model involves specifying the components that are available and the constraints between these components.

Changes to an already defined model involve changes in components (because some components are no longer available or new features be come available to existing components) and/or changes to the constraints (because of new or changed components or because market requirements cause some combinations to be invalid). These changes occur infrequently relative to the number of configuration tasks carried out.

It is important from an algorithmic point of view because it allows us to justify spending some time to preprocess a model to speed up the algorithms used during the configuration task.

1.2.3 Currently existing popular configuration systems

There are various configuration systems that are currently being used in the industry. Based on different types of algorithms used they can mostly be categorized under BDD, CSP and SAT techniques.

BDDs (Bryant, 1986; Jensen, 2004) is an abbreviation of *Binary Decision Diagrams* is one of the techniques in which the whole configuration space is compactly represented in the memory. Because of this, the space can be quickly searched for the properties required. However, the compaction and representation of the

space takes a significant amount of pre-processing. Furthermore, depending on the dataset, huge amount of memory might be required as well.

SAT, also more formally known as *Boolean Satisfiability* is another technique in which all variables involved have a Boolean domain. SAT solvers solve problems that are written in Boolean expression using only AND, OR, NOT variables and parentheses. Such and expression is then evaluated for possible states of different variables to see if the results holds to be TRUE or FALSE. Contrary to BDDs, this technique relies on searching the space for solution, which requires considerable time.

CSPs that stands for Constraint Satisfaction Programming, on the other hand might require less preprocessing compared to SAT or the BDD techniques. Preparing the correct model for the solution is the only preprocessing that is required (excluding parsing the data). However, CSPs are often known for being rather slow compared to their SAT counterparts. For a fair balance between preprocessing the time required for a solution, CSPs might be one the best approaches if modelled correctly.

1.3 The Constraint Satisfaction Problem.

The classic definition of a Constraint Satisfaction Problem (CSP) is as follows. A CSP is a triple $P = (X,D,C)$ where X is an n-tuple of variables $X = (x_1, x_2, ..., x_n)$, D is a corresponding n-tuple of domains $D = (D_1,D_2, ...,D_n)$ such that $x_i \in D_i$, C is a t-tuple of constraints $C = (C_1,C_2, ...,C_t)$. A constraint C_j is a pair (R_{S_j}, S_j) where R_{S_j} is a relation on the variables in $S_i = \text{scope}(C_i)$. In other words, R_i is a subset of the Cartesian product of the domains of the variables in S_i.

A solution to the CSP is an n-tuple $A = (a_1, a_2, ..., a_N)$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of A onto the scope S_j.

In a given task one may be required to find the set of all solutions, sol(P), to determine if that set is non-empty or just to find any solution, if one exists. If the set of solutions is empty the CSP is unsatisfiable.

The space of all solutions normally is large, but in a real life problems computing all of them is usually not needed. More specific questions are rather interesting; for instance, "If I set these variables to these values, is there still any solution?" And in this project, more specifically, we are interested at only a few selected variables at once, so we compute a subset of all solutions, which we call a partial configuration.

1.4 Definition of Partial Configuration

In large scale organizations, multiple product development teams are working individually on multiple features. Hence, it is very likely that each team will be interested in only small amount of variables important to them, instead of considering all the variables in place. One can then use a configurator to compute the set of allowed combinations but only for a subset of the whole variables. However, while doing this the configurator has to take in consideration all possible restrictions as well. This technique of using a subset of variables for computing the combinations while considering all the restrictions is what we have defined as partial configuration during this thesis project.

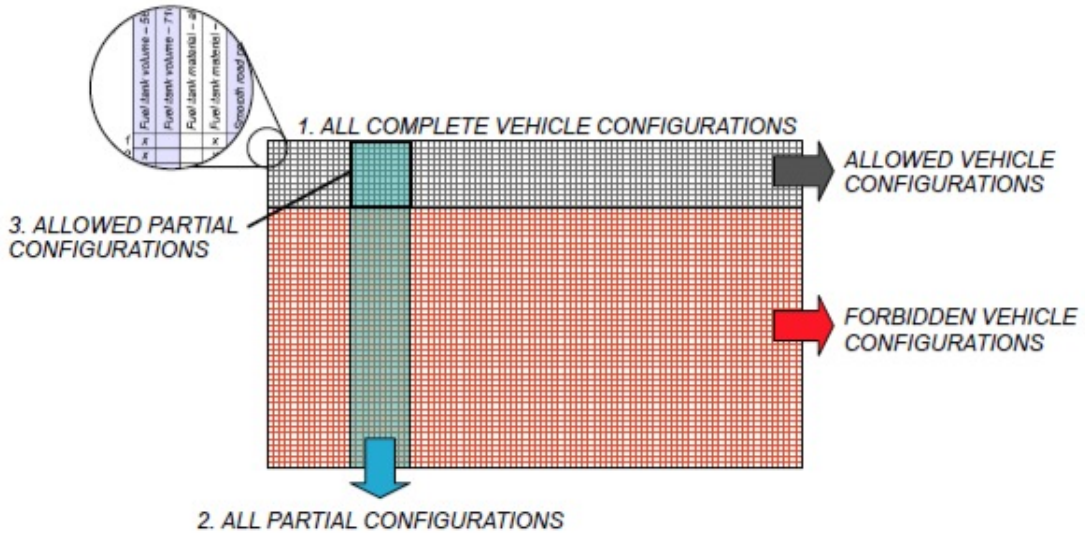


Figure 1.3: A visual representation of partial configuration.

2

Method

The configuration data considered is from an automobile company which is presented in the VDS (Vehicle Description Summary) format. The data is prepared as a standard spreadsheet (Excel) file. It contains all the information required about *options*, *families*, *models* and *restrictions* on option usage.

Options are parts or features of the car that can be selected. *Families* group options and add a restriction that no more than one option can be selected from a family. *Models* can be seen as a very influential high-level family, with different body styles and series as options. *Restrictions* limit how options can be combined with each other.

To manage some more complexities, families can further be of two types: *regular* and *modular*. Regular families have either zero or one option selected, while modular families must have exactly one option selected.

After the basic explanation of data, the problem can be presented more con-

cisely. The end user of the *configurator* in this case is a designer rather than a vehicle customer. A designer is responsible for creating various designs and verifying them. However, in a normal scenario of vehicle industry this means painstakingly building a configuration which holds true to all the different restrictions of all the parts involved. The configurator tends to solve this problem by correctly generating the possible configurations for the parameter(family), that the designer is interested in.

2.1 Modelling Configuration problem as CSP.

As defined in Section 1.3 on page 7, the main parts to a constraint satisfaction problem are a set of variables X, a set of domains of values D and a set of constraints C. For every element in the set X, there is a corresponding domain of values defined in the set D, such that the element belongs to the values within domain D. Further, the set C defines constraints which constructs the relation between various variables in the set X. Solving the problem leads to a result which satisfies all the constraint requirement and variable domain belongings.

In our case, the data is not a straightforward representation of a general CSP problem. However, adding layers to manage the complexity results in a much concise representation of the data.

Families bring together all the *options* that belong to it. Some self-contained yes-no *options* are converted into modular *families* with *selected* and *not-selected* as options. Every *option* has a set of *models* that it has been assigned to and a set of *restrictions* that it is imposed. This forms the very basis of the task of modeling the problem into a CSP problem.

Families are modelled as the variables X, which belongs to the set of *options* modelled as the set D. Constraints between the *options* are modelled as the variable C as mentioned above. *Models* are further imposed as another layer of restriction between the *options*.

A valid solution to the problem would be a set of *options*, each belonging to one particular *family*, which satisfies all the restrictions within the *options* as well as in the *models*.

Furthermore, as the designer would most of the times be interested only in the families that belong to his design necessities, s/he must be able to select the *families* that interests him/her. In this case, the valid solutions will the ones that contains the results that s/he is interested while still satisfying all the restrictions within the *options* and the *models*.

2.2 General algorithm for CSP solver

2.2.1 Constraint Propagation and Search

Constraint satisfaction problems on finite domains are usually solved using a form of search, and the most popular techniques are variants of *backtracking*, *constraint propagation*, and *local search*.

In our case, we have chosen to use *constraint propagation* as the search technique. Our primary reason to select the technique was also based on the tool that we chose to use. *Gecode* was the constraint programming framework that we chose to use, as it has been proven robust in the industry for modeling and processing constraint satisfaction problems. It searches through the solution space using

propagation and *branching*, and hence our technique selection.

Constraint propagation excludes those values from being assigned to a variable which are incompatible with a solution. Values not yet excluded are stored in the domain of a variable, also sometimes called as *constraint variable*. A *propagator*, which represents an individual constraint over a number of variables, excludes such incompatible values from the domains encapsulating an algorithm to filter out incompatible values, using a *filter algorithm*.

Propagation is typically *not* able to exclude all but one value per variable and thus, to produce an assignment for a given problem. Hence, constraint propagation is complemented by search. As soon as constraint propagation stopped because no further values can be excluded, search *branches* to different alternatives in a speculative way excluding values and thus, triggering new constraint propagation in the alternatives. A *brancher* determines the shape of the search tree, defining what speculative ways the values should be further excluded while searching down the tree.

Constraint propagation (with branching and search) is performed until a desired solution is eventually found or all alternatives are explored.

2.2.2 Search strategies

A search engine performs search by computing *spaces*. A space implements a constraint model on which the search operation is performed. The search operation normally goes through the following process for each space it encounters.

It begins by trying to solve the space and then queries for the result. If the space is fully solved, the solution is found and the search can terminate for that

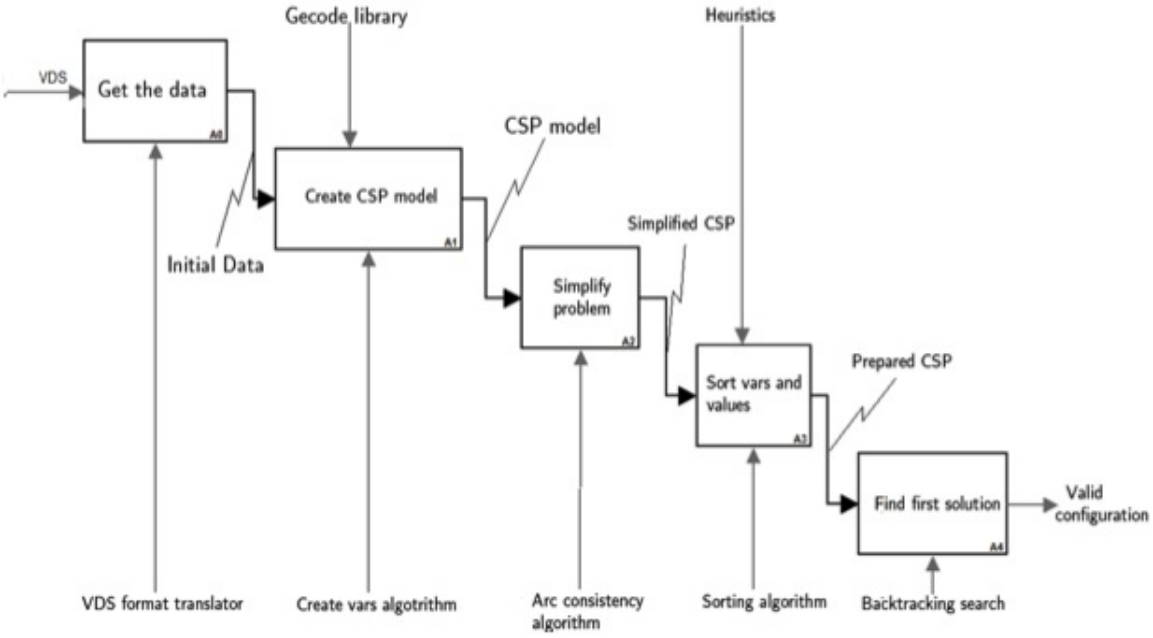


Figure 2.1: Visual respresentation of workflow in the implementation of the configuration

particular space.

If the solution is not found, then the engine finds the unsolved variables and branches on it. But, before the engine can branch it has to *clone* its current space, which is required so that the it can perform *backtracking*. By cloning its current space, the engine can later come back to the same space in case the branch it traveled down did not yield any results.

Furthermore, search engines can be equipped with special heuristics. Doing so allows them to be further fine-tuned, for instance, they can determine the best variables to branch on, keep track of the best solution found so far etc, and all other customizable logic required in the relevant project more than a generic search engine.

DFS, better known as *Depth First Search* is an algorithm for traversing a search

tree, a tree structure, or a graph. The search tree explained above performs search in a tree structure, and DFS is one of the best algorithms that ensures proper search results in a tree structure.

DFS in its simplest form can be defined as an *uninformed search* that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it has not finished exploring. DFS can be optimised further to give better results faster with the addition of different search heuristics.

Because of the nature of the problem at hand, its representation as a tree structure and the search engine's requirement, DFS seems to be a natural choice to start with. With further added heuristics, DFS can be improved to provide quality results within acceptable time.

2.2.3 Search Heuristics

As we pointed out above, a CSP heuristic includes a variable/value selection procedure. Classical value ordering strategies can be summarized as follows: min-value selects the minimum value, max-value selects the maximum value and mid-value selects the median value in the remaining domain. Usually variable selection heuristics are more important and comprehend more sophisticated algorithms.

Various heuristics are considered for the preliminary tests, and then these are further benchmarked against each other to make the final selection. The heuristics involved are explained further in the results with benchmark data.

2.3 Partial configuration

The idea of partial configuration, as defined in Section 1.4 on page 8 is further implemented as another layer on the top of the general CSP solver to obtain concise results.

In this case, the *families* that are of specific interest to the designer is selected and the solver is run to generate results only enumerating the ones in concern to those *families*. Two approaches have been considered in our case, both of which are explained below.

2.3.1 Enumerate and Query Search

The first approach is to enumerate all complete partial configuration and then ask the solver if each of them are allowed. This process can also be listed a a brute-force approach to finding all the solutions. In this approach, the *configurator* considers all the variables in interest. After the necessary constraint propagation for all these variables, the valid domain for each of these variables are enlisted. An enumeration of all the possible combinations between the elements in the list of valid domains is done. The *configurator* then queries for each enumeration to see whether it is a valid result or not. The *configurator* has to traverse through the whole list before providing *complete* solutions.

2.3.2 Dynamic Constraint Propagation and Search

Dynamic CSPs (DCSPs) are useful when the original formulation of a problem is altered in some way, typically because the set of constraints to consider evolves because of the environment. DCSPs are viewed as a sequence of static CSPs, each

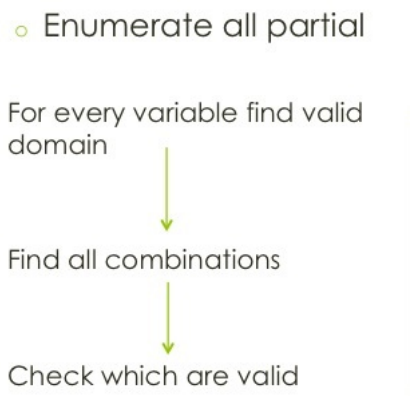


Figure 2.2: Search flow for 'Enumerate and Query' Search

one a transformation of the previous one in which variables and constraints can be added (restriction) or removed (relaxation). Information found in the initial formulations of the problem can be used to refine the next ones.

In this approach, initially the *configurator* considers all the variables in interest. It then looks for the first solution possible. After finding the first solution, further constraints are added to the *configurator* that specify the solution just found not to be a solution. This process is repeated for every solution found, which forces the *configurator* to look for new solutions.

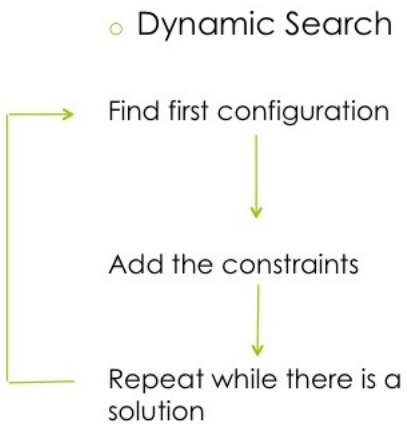


Figure 2.3: Search flow for 'Dynamic Constraint Propagation' and Search

3

Implementation

A lot of libraries exist for modelling and solving CSP problems. The two most popular frameworks among the ones we studied are Choco and Gecode and they show almost the same results at CSP Solvers competition, but we choose Gecode as it provides more complete documentation.

Gecode is an open, free, portable, accessible, and efficient environment for developing constraint-based systems and applications. It is implemented in C++ and offers competitive performances w.r.t. both run-time and memory usage. It implements a lot of data structures, constraints definitions, and search strategies, allowing also the user to define his own ones.

3.1 Modelling in Gecode

Gecode models are implemented using spaces, space is the home(storage) for variables, propagators (implemented constraints), branchers (or labellings, describing

the shape of the tree), and possibly an order - determining the best solution during search. Therefore we need to define what is variables and constraints in our problem and then define branchers for them.

Configuration data is exported in a specific industrial format. It contains information about options, families, models, and restrictions on option usage.

Options are parts or features of the car that can be selected. Families group options and add a restriction that no more than one option can be selected from a family. Models can be seen as a very influential high-level family, with different body styles and series as options. Restrictions limit how options can be combined with each other.

3.1.1 Creating variables.

Families of options we see as integer variables. Each variable has its own domain which consists of options that can be selected in this family. Another variable is a model with the domain of available models.

Options and families appear in the format in different flavors. Families can be of two different family types: regular and modular. Regular families must have either zero or one option selected from it. Modular families must have exactly one option selected from it. Therefore we convert regular families to modular ones by adding an extra option ”not selected” to the family.

Options can belong to a family or be self-contained yes-no options. Self-contained yes-no options (they have no family associated with them) considered as a modular family with options ”Yes” and ”No”.

All options and families have unique codes, respectively called option codes and

family codes. Knowing this we can create a unique mapping of options and families to the natural numbers, which allows us to deal with them as integer variables.

3.1.2 Posting constraints.

Each option has one or more restrictions or conditions that specify when the option can be used.

```
OptionA -> (ConditionA1 and ConditionA2 and...
                                          and ConditionAN) (1)
```

where a -> b = ((not a) or b).

Each condition consists of two parts. The first part, model part, describes the models for which the condition applies. The second part, options part, describes the options that have to be selected or not selected to satisfy the condition. The model part can describe several models, which means that any of them can be selected for this condition:

```
ConditionA1 = ((Model1 or Model2 or ... or ModelN) and
                                        OptionsExpression) (2)
```

3.1.3 Model part

Model part describes brand, carline, model series and body style.

3.1.4 Options part

Options part of the condition describes the options that have to be selected or not selected. This part can be written on one or more lines and can be seen as a boolean expression with regular operators - and, or, not.

3.1.5 Translating restrictions

Translating restrictions part of the string to the Gecode format is performed in 2 steps:

1. Parse restriction string to the abstract syntax tree, according to the grammar.
2. Traverse the tree recursively and create boolean expressions for every node.
   Create boolean expression of the form :
   Family Variable == OptionCode
   for every option code leaf in the restriction.

After this process we have got one BoolVar, which is constructed as boolean expression in the string.

3.1.6 Translating models

All models are gathering from the string separately to get needed BoolVar with expression of the following form:

```
Model1 or Model2 or ... or ModelN
```

Between all these restrictions for one option there is an ”or” relation, either of them can be true.

Hence, the final relation posted to the model, for every option derives from the logic expressed below.

```
rel(*this, expr (*this,(families[cur_fam] == cur_opt)
                                        >> restriction), IRT_EQ, 1)
```

3.2 Search

Search involves two techniques: branching and exploration. Branching defines the shape of the search tree. Exploration defines a strategy how to explore parts of the search tree and how to possibly modify the tree's shape during exploration (for example, in the partial configuration search by adding new constraints).

3.2.1 Branching

Gecode offers predefined variable-value branching: when calling Branch to post a branching, the third argument defines which variable is selected for branching, whereas the forth argument defines which values are selected for branching.

The posted brancher assigns all variables and then ceases to exist. If more branchers exist, search continues with the next brancher.

We have two branchers - for family variables and for the model.

```
branch(*this, families, INT_VAR_SIZE_MIN, INT_VAL_MAX);
branch(*this, model, INT_VAL_MAX);
```

Defining the optimal arguments to pass to the brancher, i.e. what heuristics to use is the matter of experimentation.

3.2.2 Tiebraking

Usually tie-breaking for the variable selection chooses the first variable (the variable with the lowest index in the array) that satisfyes the selection criteria. For many applications that is not very sufficient behaviour, including our case. For integer variables it is important to select the most constrained variable first (the variable most propagators depend on, that is, with the largest degree). Then, among the most constrained variables select the variable with the smallest size of the domain. This can be achieved by using a tiebreak function:

```
branch(\*this, families, tiebreak(INT_VAR_DEGREE_MAX,
                                  INT_VAR_SIZE_MIN),INT_VAL_MIN);
```

This technique could be useful in our case, so this kind of heuristic was also included to the experiment.

3.2.3 Recomputation

For the backtracking search the most expensive operation is returning to the previous states: as spaces constitute nodes of the search tree, a previous state is nothing but a space. Returning to a previous space of solutions is performed when an alternative suggested by a branching did not lead to a solution.

Two spaces are equivalent if propagation and branching and hence search behaves exactly the same on both spaces. There are two techniques for restoring spaces: recomputation and cloning.

Cloning creates a clone of the space. A clone and the original space are equivalent. Restoration with cloning is straightforward: before following a particular alternative during search, a clone of the space is created and used later if necessary.

Recomputation remembers what has happened during branching: rather than storing an entire clone of a space it saves just enough information to redo the effect of a brancher. In Gecode this information stored is called a choice. Redoing the effect is called commiting a space: given a space and a choice committing reexecutes the brancher as described by the choice and the alternative to be explored (for example, left or right branch).

Hybrid recomputation. The hybrid of recomputation and cloning works as follows. For each new choice node, a choice is stored. Then, every now and then search also stores a clone of the space (after every constant number of steps). Now, restoring a space at a certain position in the search tree traverses the path in the tree upwards until a clone is found on the path. Then recomputation creates a clone of this clone. Then all choices on the path are committed on the found clone yielding an equivalent space.

If only cloning but no recomputation is used, there is one clone operation, one commit operation, and one status operation to perform constraint propagation. With hybrid recomputation, one clone operation, three commit operations, and one status operation to perform constraint propagation are needed (as shown above). Commit operations are very cheap (most often, just modifying a single variable or posting a constraint). What is essential is that in both cases only a single status operation is executed. Hence, the cost for constraint propagation during hybrid recomputation turns out to be not much higher than the cost without

recomputation.

Adaptive recomputation. If a node must be recomputed, adaptive recomputation creates an additional clone in the middle of the recomputation path. A clone created during adaptive recomputation is likely to be a good investment. Most likely, an entire failed subtree will be explored. Hence, the clone will be reused several times for reducing the amount of constraint propagation during recomputation.

Hybrid and adaptive recomputation can be easily controlled by two integers C_d (commit distance) and A_d (adaptive distance). The value for C_d controls how many clones are created during exploration: a search engine creates clones during exploration to ensure that recomputation executes at most C_d commit operations. The value for A_d controls adaptive recomputation: only if the clone for recomputation is more than ad commit operations away from the node to be recomputed, adaptive recomputation is used. Values for C_d and A_d are used to configure the behavior of search engines using hybrid and adaptive recomputation. The number of commit operations as distance measure is approximately the same as the length of a path in the search tree. It is only an approximation as search engines use additional techniques to avoid some unused clone and commit operations.

3.2.4 Parallel search

Gecode uses a standard work-stealing architecture for the parallel search: initially, all work (the entire search tree to be explored) is given to a single worker for the exploration, making the worker busy. All other workers are initially idle, and try to steal work from a busy worker. Stealing work means that part of the search

tree is given from a busy worker to an idle worker such that the idle worker can become busy itself. If a busy worker becomes idle, it tries to steal new work from a busy worker. As work-stealing is indeterministic (depending on how threads are scheduled, machine load, and other factors), the work that is stolen varies over different runs for the very same problem: an idle worker could potentially steal different subtrees from different busy work ers. As different subtrees contain different solutions, it is indeterministic which solution is found first.

Naturally, the amount of search needed to find a first solution might differ both from sequential search and among different parallel searches. Note that this might actually lead to super-linear speedup (for n workers, the time to find a first solution is less than $1/n$ the time of sequential search) or also to real slowdown.

Using parallel search allows significantly improve search time for our problem and was used in all experiments.

3.2.5 Partial configuration search

Both methods of partial search described in the Chapter 2 were implemented and used in our experiments.

3.2.6 Previous solution reuse

Solution reuse for the partial configurations search was implemented using Restart search engine available in Gecode. New constraints are posted for every next solution search using the function constraint. First of all, those that make previous found configuration not allowed and then those that help to narrow down the solution space.

4

Results

As described in the previous section, the *configurator* was successfully developed to process the data that was obtained. The sucess of the program and the algorithms in use were verified using the test data provided along. As the *configurator* was able to present exactly the same solutions as required by the test data, it was considered reliable to use for experimental inputs. After verifying the *configurator*, time was spent on selecting the proper heuristics required for the search and speeding up the search process.

Considerable amount of time was spent on speeding up the *configurator* after which the results were then collected. This was done so as to be able to benchmark the various algorithms considered as well as external tools built with the same purpose. While one of the main goal of the project was to use CSP as the solver for the problem in hand, the other goal was also to investigate whether it was worthy enough of industrial usage in case the *configurator* could generate the

solutions using CSP.

4.1 Selection of the Search Heuristic

After verifying the test cases, the next step involved the selection of proper heuristics to speed up the search process algorithmically. Initially, a lot of different combination of heuristics were tested for generation of the first result of the soltion being tested. The time taken was considered and then 6 different heuristics were selected finally for the best speed.

The heuristics named as *Heuristics 0* to *Heuristics 5* are described below.

1. Heuristic 0 : Branching over a tie-break of variables of largest degree and smallest domains size with the largest values
2. Heuristic 1 : Branching over a tie-break of variables of largest degree and smallest domains size with the smallest values
3. Heuristic 2 : Branching on variable of largest degree with smallest values
4. Heuristic 3 : Branching on variable of largest degree with largest values
5. Heuristic 4 : Branching on variable of smallest degree with smallest values
6. Heuristic 5 : Branching on variable of smallest degree with largest values

To investigate the performance of the *configurator* various datasets of different complexity size was required. The official dataset had around 35000 constraints to parse; the same dataset was further processed manually and constraints were removed making sure that it would only simplify the data and not eventually break

it. Datasets having around 12000 to the original 35000 constraints were created in intervals of around 2000. Each heuristic was then run over the various datasets to obtain the first valid solution, the runtime of which were measured. The results thus obtained are presented on the following table with an accompanying plot.

The following table lists the time taken by each heuristic averaged over the datasets with various number of constraints as represented in the plots.

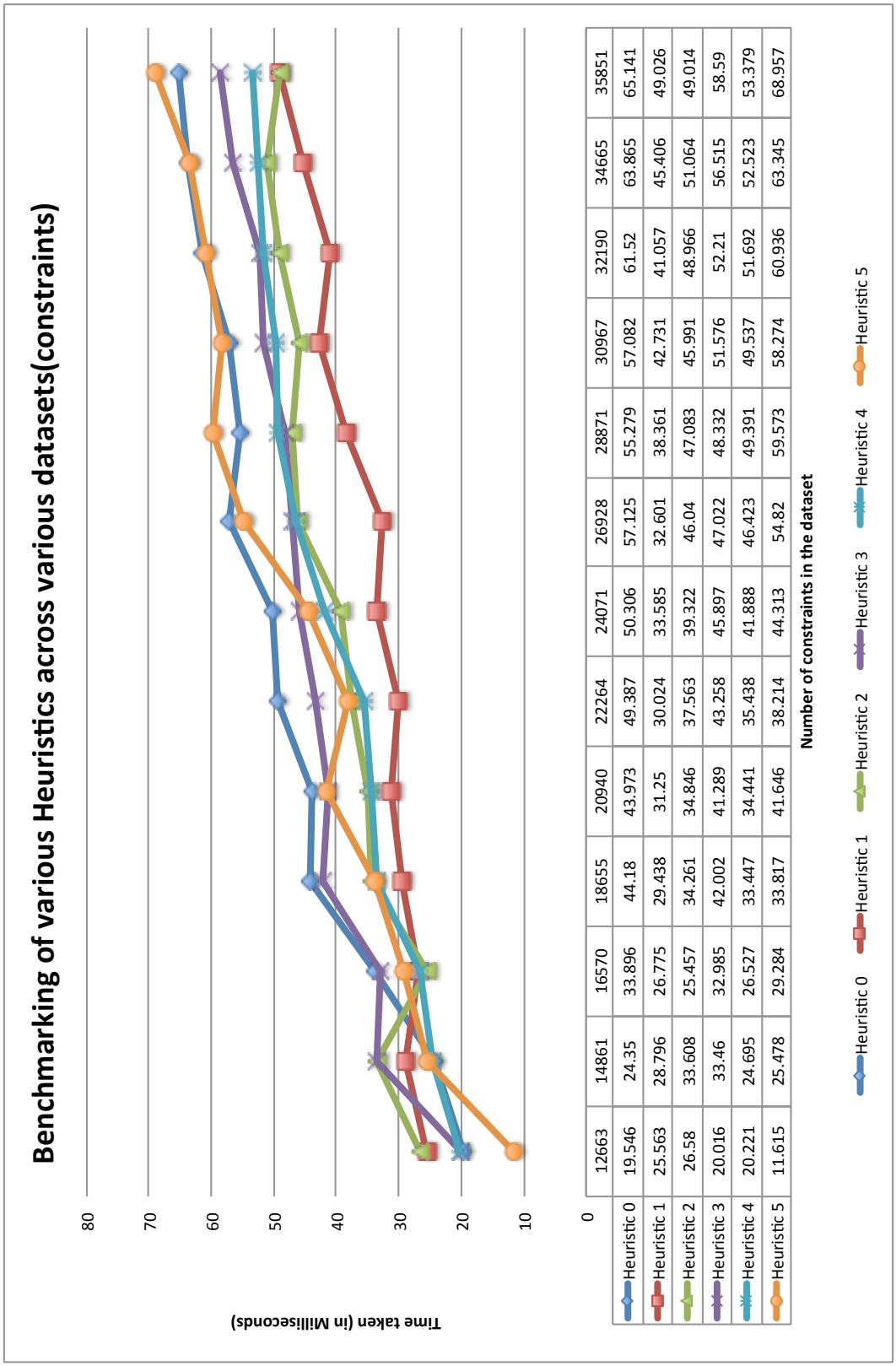
Heuristic Used (to find the first possible solution)	Time taken (Average) (in milliseconds)
Heuristic 0	48.1269 ms
Heuristic 1	34.9702 ms
Heuristic 2	39.9842 ms
Heuristic 3	44.0886 ms
Heuristic 4	39.9693 ms
Heuristic 5	45.4055 ms

Table 4.1: Heuristics used and Time taken

The above table lists the time taken to obtain the first valid solution using various heuristics, averaged over the datasets created.

Based on the results, it is clearly visible that *Heuristic 1* takes the least amount of time to solve the problem. The same conclusion can be reached by inspecting the accompanying previous plots as well.

This clearly led to the usage of *Heuristic 1* for further configurations in the case of generating partial configurations.



	12663	14861	16570	18655	20940	22264	24071	26928	28871	30967	32190	34665	35851
Heuristic 0	19.546	24.35	33.896	44.18	43.973	49.387	50.306	57.125	55.279	57.082	61.52	63.865	65.141
Heuristic 1	25.563	28.796	26.775	29.438	31.25	30.024	33.585	32.601	38.361	42.731	41.057	45.406	49.026
Heuristic 2	26.58	33.608	25.457	34.261	34.846	37.563	39.322	46.04	47.083	45.991	48.966	51.064	49.014
Heuristic 3	20.016	33.46	32.985	42.002	41.289	43.258	45.897	47.022	48.332	51.576	52.21	56.515	58.59
Heuristic 4	20.221	24.695	26.527	33.447	34.441	35.438	41.888	46.423	49.391	49.537	51.692	52.523	53.379
Heuristic 5	11.615	25.478	29.284	33.817	41.646	38.214	44.313	54.82	59.573	58.274	60.936	63.345	68.957

Figure 4.1: Benchmarking results for the selection of best heuristic.

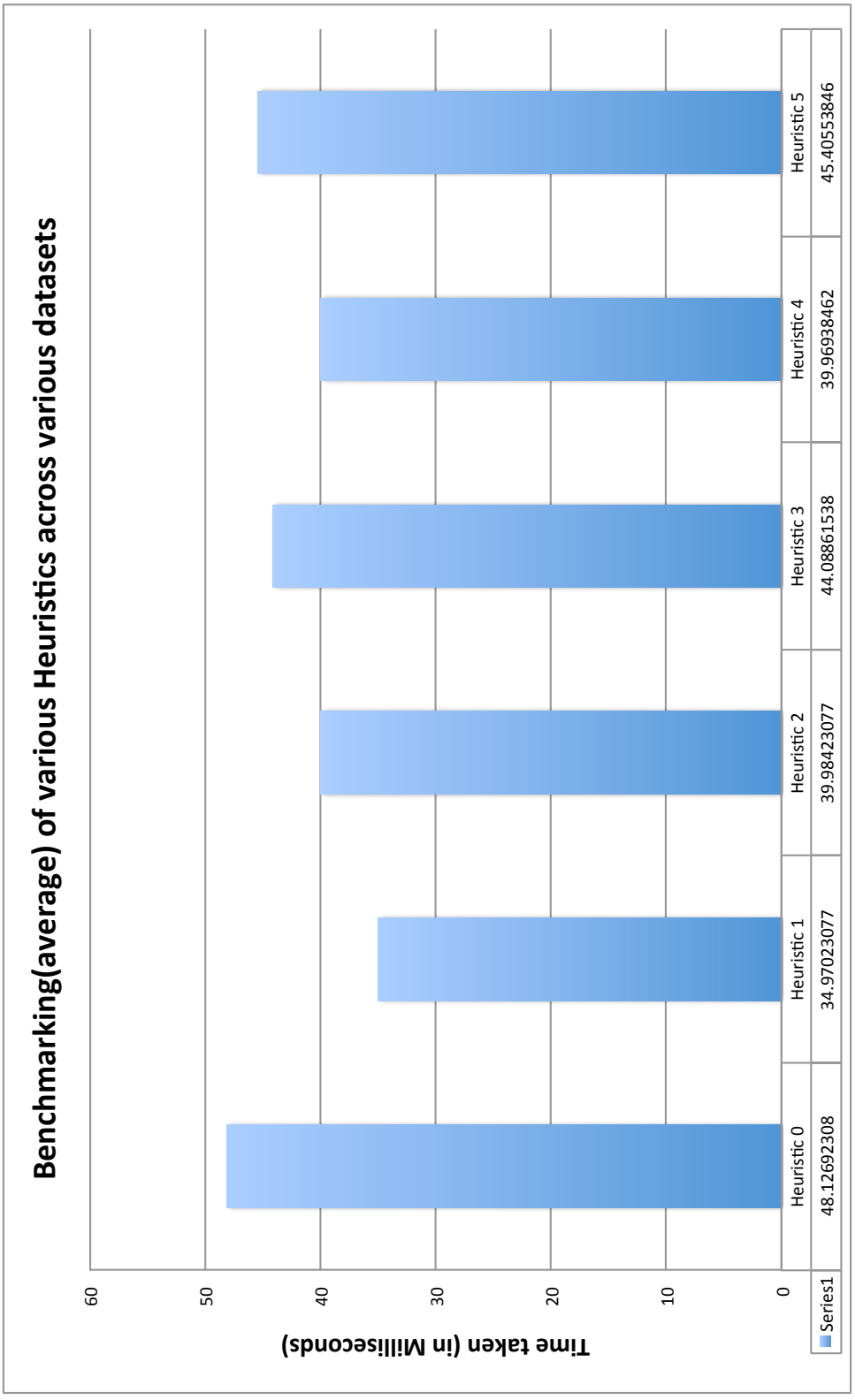


Figure 4.2: Averaged benchmarking results for the selection of best heuristic.

4.2 Benchmarking of Partial Configuration

Mentioned in the previous sections, two approaches were considered for the generation of partial configurations.

The first technique (Enumerate and Query Approach) enumerated all valid combination and then the solver was asked to solve for each of the combination. The second technique (Dynamic Search) involved the dynamic addition of constraints to the solver; every time a valid solution was found, the solution was added as a further constraint hence asking the solver to search for another solution further.

From the results generated before, *Heuristic 1* was chosed to be used in all of the tests to benchmark the techniques. For each of the technique, the process was the same. The test was run with inputs as the interesting variables and the output was checked for as expected. To benchmark the solutions, the runtimes were compared once again across the techniques and with the result obtained from the external tool (SAT-based) as well. Based on the technique in use, the search time was calculated differently, in the Enumerate and Query Approach the search time was measured excluding the re/modeling time, while in the case of Dynamic Search the re/modeling time was included as it was one of the main parts of the search loop in itself. The results obtained from the benchmarking tests are presented below.

Variables	All Combinations	Valid Combinations	Search Time
8	256	8	5391 ms
8	5760	498	92427 ms
11	103680	106	2181223 ms

Table 4.2: Results for CSP Solver (Enumerate and Query Approach)

Variables	All Combinations	Valid Combinations	Search Time
8	256	8	87654 ms
8	5760	498	289417 ms
11	103680	106	204578 ms

Table 4.3: Results for CSP Solver (Dynamic Search Approach)

4.3 Explanation of the Partial Configuration Results

After obtaining the final benchmarking results as listed above, the comparision between various systems could eventually be made.

For the first approach, that is the ”Enumerate and Query” approach, we can see that the total search time increases propotionally to the number of total combinations to be enumerated and queried. The explanation for it is quite simple

Variables	All Combinations	Valid Combinations	Search Time
8	256	8	150 ms
8	5760	498	3218 ms
11	103680	106	881 ms

Table 4.4: Results for external SAT solver (SAT constructive Search)

as the design of the algorithm. In this approach, the *configurator* considers all the variables of interest. After the necessary constraint propagation for all these variables, the valid domain for each of these variables is enlisted. An enumeration of all the possible combinations between the elements in the list of valid domains is done. The *configurator* then queries for each enumeration to see whether it is a valid result or not. The *configurator* has to traverse through the whole list before providing *complete* solutions, hence the explanation for the time taken to find all the results. Since the *configurator* queries all the possible combinations, the total time taken directly is a summation of time taken for individual queries on all these combinations.

For the second approach, that is the "Dynamic Search" approach, the explanation is not quite straightforward as above. In this approach, initially the *configurator* considers all the variables of interest. It then looks for the first solution possible. After finding the first solution, further constraints are added to the *configurator* that specify the solution just found not to be a solution. This process is repeated for every solution found, which forces the *configurator* to look for new solutions. As we can see that while the search time does increase with the increased amount of total combinations, they are rather proportional to the number of valid combinations. We can see that, unlike in the previous approach the search time actually is a function of valid solutions. However, during the test processes we also came across the search behaviour of the *configurator* in which the first 50-70% of the solutions took about 1/10,000 to 1/1,000 of the total search time, while the rest of the solutions consumed the remaining time. Despite various debugging sessions, we could not come to the conclusion for the displayed behaviour.

The search time for the SAT constructive search was taken from a separately

developed *configurator*; not a part of this project. The results were used to benchmark the developed CSP solution against the pre-existing SAT solution.

4.4 Conclusions

The above results points towards one conclusion univocally. While CSPs might be easy to represent the data verbally and mentally, SAT solvers however beat CSPs in terms of computation. One of the main goal of the project was to perform a study on suitability of CSPs over SAT solvers. With the results presented above, it clearly marks the poor performance of CSPs in comparision to SAT, in this case. Despite the fact that there lies some room for improvement in the CSP performance, the SAT solution looks much more capable than the one that was developed for the project. Hence, it has been concluded that despite the richness of CSP vocabulary in explaining the problem at hand, SAT seems to be significantly better in terms of performance.

Bibliography

[1] K. R. Apt, Some Remarks on Boolean Constraint Propagation, System.

[2] S. Brailsford, Constraint satisfaction problems: Algorithms and applications, European Journal of Operational Research 119 (3) (1999) 557–581.
URL http://linkinghub.elsevier.com/retrieve/pii/S0377221798003646

[3] B. O. Callaghan, B. O. Sullivan, E. C. Freuder, Corrective Explanation for Interactive Constraint Satisfaction, Computer.

[4] R. Dechter, T-PROCESSING Cognitive System Labcbratory , Computer Science Department Net address : dechter@csmla.edu Net address : judea@cs.ucla.edu, Office.

[5] A. Dechtes, Rim Dechter Cognitive Systems Laboratory Computer Science Department University of California, Los Angeles, CA 90024, aaai.org (1988) 37–42.
URL http://www.aaai.org/Papers/AAAI/1988/AAAI88-007.pdf

[6] P. Flener, A Programming Paradigm on the Rise (December).

[7] E. Gelle, Dynamic Constraint Timo Soininen Satisfaction in Configuration The DCSP Formalism Definition (1999) 95–100.

[8] S. M. Gonz, Open , Interactive and Dynamic CSP.

[9] M. Janota, Do SAT solvers make good configurators, in: First Workshop on Analyses of Software Product Lines (ASPL), Citeseer, 2008, pp. 1–5.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.155.4099&rep=rep1&type=pdf`

[10] U. Junker, Q UICK X PLAIN : Preferred Explanations and Relaxations for Over-Constrained Problems, Computing 3 167–172.

[11] V. Kumar, Algorithms for Constraint- Satisfaction Problems :, AI Magazine (1992) 32–44.

[12] M. Z. Lagerkvist, C. Schulte, Advisors for Incremental Propagation.

[13] J. Madsen, Methods for interactive constraint satisfaction, Master's thesis, University of Copenhagen.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.7905&rep=rep1&type=pdf`

[14] S. Mittal, Towards a generic model of configuration tasks, Proceedings of the Eleventh International Joint (1989) 1395–1401.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.3469&rep=rep1&type=pdf`

[15] X. Palo, H. Road, Dynamic Constraint, Sciences-New York (1990) 25–32.

[16] A. Productique, R. Ampere, C. D. G. Industriel, J. R. D. Teiuet, From CSP to configuration problems (1999) 101–106.

[17] I. Razgon, B. O'Sullivan, Efficient recognition of acyclic clustered constraint satisfaction problems, in: Proceedings of the constraint solving and contraint logic programming 11th annual ERCIM international conference on Recent advances in constraints, no. 05, Springer-Verlag, 2006, pp. 154–168.
URL `http://portal.acm.org/citation.cfm?id=1776508`

[18] N. Roos, Y. Ran, J. V. D. Herik, Combining Local Search and Constraint Propagation to Find a Minimal Change Solution for a Dynamic CSP, Search (2000) 272–282.

[19] Y. Schreiber, Cost-Driven Interactive CSP with Constraint (2009) 707–722.

[20] G. Tack, Constraint Propagation.

[21] A. Tidstam, Information Modelling for Automotive Configuration, Leather.

[22] G. Verfaillie, T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: Proceedings of the National Conference on Artificial Intelligence, JOHN WILEY & SONS LTD, 1994, pp. 307–307.
URL `http://www.aaai.org/Papers/AAAI/1994/AAAI94-302.pdf`

[23] J. White, B. Dougherty, D. C. Schmidt, Automated Reasoning for Multi-step Feature Model Configuration Problems, Challenge.

[24] P.-A. Yvars, Using constraint satisfaction for designing mechanical systems, International Journal on Interactive Design and Manufacturing (IJIDeM) 2 (3)

(2008) 161–167.

URL http://www.springerlink.com/index/10.1007/s12008-008-0047-3

A

List of Heuristics investigated

A.1 Available setting for branchers in Gecode

INT_VAR_NONE	first unassigned
INT_VAR_RND	randomly
INT_VAR_DEGREE_MIN	smallest degree
INT_VAR_DEGREE_MAX	largest degree
INT_VAR_AFC_MIN	smallest accumulated failure count (AFC)
INT_VAR_AFC_MAX	largest accumulated failure count (AFC)
INT_VAR_MIN_MIN	smallest minimum value
INT_VAR_MIN_MAX	largest minimum value
INT_VAR_MAX_MIN	smallest maximum value
INT_VAR_MAX_MAX	largest maximum value
INT_VAR_SIZE_MIN	smallest domain size
INT_VAR_SIZE_MAX	largest domain size

INT_VAR_SIZE_DEGREE_MIN	smallest domain size divided by degree
INT_VAR_SIZE_DEGREE_MAX	largest domain size divided by degree
INT_VAR_SIZE_AFC_MIN	smallest domain size divided by AFC
INT_VAR_SIZE_AFC_MAX	largest domain size divided by AFC
INT_VAR_REGRET_MIN_MIN	smallest minimum-regret
INT_VAR_REGRET_MIN_MAX	largest minimum-regret
INT_VAR_REGRET_MAX_MIN	smallest maximum-regret
INT_VAR_REGRET_MAX_MAX	largest maximum-regret
INT_VAL_MIN	smallest value
INT_VAL_MED	greatest value not greater than the median
INT_VAL_MAX	largest value
INT_VAL_RND	random value
INT_VAL_SPLIT_MIN	values not greater than mean of smallest and largest value
INT_VAL_SPLIT_MAX	values greater than mean of smallest and largest value
INT_VAL_RANGE_MIN	values from smallest range, if domain has several ranges; otherwise, values not greater than mean of smallest and largest value
INT_VAL_RANGE_MAX	values from largest range, if domain has several ranges; otherwise, values greater than mean of smallest and largest value
INT_VALUES_MIN	all values starting from smallest
INT_VALUES_MAX	all values starting from largest