DASH:
Dynamic Approach for Switching Heuristics

Giovanni Di Liberto

Supervisors:

Doctor Yuri Malitsky

Professor Barry O'Sullivan

Professor Matteo Fischetti

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Laurea Magistrale in Ingegneria Informatica

A mio fratello

Table of Contents

Acknowledgements

This master's thesis has been carried out at Cork Constraint Computing Group (4C), University College Cork, since January 2013, thanks to the Erasmus program which gave me the opportunity to spend an amazing year in Ireland. A number of people deserve thanks for their support and help.

First of all, I would like to convey my gratitude to my supervisor Professor Barry O'Sullivan (University College Cork), who gave me the opportunity to conduct my study in his research group and to my supervisor Professor Matteo Fischetti (Università degli Studi di Padova), for your advice and valuable suggestions. I would like to thank my supervisor Doctor Yuri Malitsky (University College Cork), who first proposed the research topic behind this thesis and offered his research guidance. Thank you for all of the work you put into supervising this project, for having taught me how to live in a research environment, and that the hard work can also be fun.

At 4C, I had the opportunity to meet great people from many different places around the world and grow as a person and master student. Therefore, I would like to thank who, even without collaborating directly on this thesis, helped in making the environment more relaxed and stimulating with reading groups, biscuits, and tea-breaks.

I would also like to thank my coauthors: Serdar Kadioglu and Kevin Leo, your suggestions and indications have given an important contribution in keeping me focused on the main goals of this work.

Besides the coauthors and the research group, I really need to thank Claire Donohue, who has been so helpful in proof reading this thesis and, together with Vahid Yazdan, has made the tea breaks even mightier. I also need to thank my family and my friends in Italy, for having always being ready to help me, despite the many kilometers of distance.

Finally, I must thank Marco Collautti, for an awesome year together in Cork, for pushing me in looking for a master's thesis at UCC, and for the long discussions about our futures that helped me to understand what I really want to do.

Abstract

Complete tree search is a highly effective method for tackling MIP problems, and over the years, a plethora of branching heuristics have been introduced to further refine the technique for varying problems. Recently, portfolio algorithms have taken the process a step further, trying to predict the best heuristic for each instance at hand. However, the motivation behind algorithm selection can be taken further still, and used to dynamically choose the most appropriate algorithm for each encountered subproblem. This thesis identifies a feature space that captures both the evolution of the problem in the branching tree and the similarity among subproblems of instances from the same MIP models. A method for exploiting these features is presented here, which decides the best time to switch the branching heuristic and it is shown how such a system can be trained efficiently. Experiments on a highly heterogeneous collection of MIP instances results in significant gains over the pure algorithm selection approach that for a given instance uses only a single heuristic throughout the search.

Chapter 1

Introduction

Mixed Integer Programming (MIP) is a powerful problem representation that is ubiquitous in the modern world. The problem is represented as the maximisation or minimisation of an objective function while maintaining the specified linear inequalities and restricting some variables to only take integer values while others are allowed to take on any real value.

Through this abstraction, it is possible to define a wide variety of problems, ranging from scheduling [1] to production planning [2] to network design [3] to auctions [4] and many others.

The scheduling problem is an example of MIP problem which involves, among other formulations, service and vehicle scheduling in transportation networks. An application may be assigning buses, trains, or subways to specific routes in order to obtain a timetable which satisfies particular conditions (for example the train scheduling problem [5, 6]). Another important problem that can be expressed in the MIP formulation, is related to production planning. This could be related, for example, to industrial or agricultural production, where the goal is to maximise the total production, without exceeding the available resources. An example of minimisation MIP problem is related to the telecomminication networks, where the total cost has to be minimised while meeting a predefined set of communication requirements.

In all the above-mentioned cases, and many others, the task of the program is to obtain a feasible solution, which satisfies the constraints imposed by the formulation, and optimises a specific objective function.

Consider, for example, the problem of determining the lessons timetable for a

secondary school, given:

- M subjects $S = \{s_1, s_2, ..., s_M\}$;
- K course blocks $C = \{c_1, c_2, ..., c_K\}$;
- N instructors $T = \{t_1, t_2, ..., t_N\}$;
- Q rooms $R = \{r_1, r_2, ..., r_Q\}$.

Furthermore, each instructor t_i can teach a set of J_i subjects $L_i = \{s_{l_1}, s_{l_2}, ..., s_{l_{J_i}}\}$, $l_i \in [1, M]$. In order to obtain a feasible timetable, a set of constraints must be defined: for each time-slot of a course block there can be just a single lesson; an instructor can't be in more than one room at the same time; each course block has a specific number of hours for each subject; each instructor should have a minumum and a maximum amount of hours per week. Finally, what makes this problem a suitable example to our case is the objective function, which targets to produce a compact timetable. These are just some of the many variables and constraints that are possible to identify, but they are enough to make the problem very difficult to solve without a methodological approach, even having small values of M, N, K, and Q.

A quite natural way to solve this problem could be to iteratively assign an instructor to a course, a room, and a subject, aiming to reduce the problem domain, and therefore obtaining a simpler problem. Good choices reduce the problem domain but still permit an optimal solution. On the contrary, a bad choice could limit the solutions domain to an area where just sub-optimal solutions can be found, or worse, no feasible solution at all. When this happens, it is possible to go back and reconsider some of the previous choices. The latter is called backtracking.

The approach presented above is generally referred to as branch-and-bound (B&B), and represented with the help of a tree. The latter gives the opportunity to remember past choices and to perform a backtrack and correct past mistakes. In practice, the main idea is to perform deterministic and inductive reasoning to lower the domains of the variables at each node. When this is no longer possible, a variable is selected and assigned a value based on some guiding heuristic, thereby obtaining a new subinstance, which is a child of the previous node. Once such a decision is made the search

proceeds to function deterministically. If or when it is later found that a decision led to an infeasible or sub-optimal solution, the search backtracks, returning to the parent node to try an alternate assignment.

The key behind the success or failure of this complete search approach is the order in which the variables are selected and the values assigned to them. Choosing certain variables can significantly reduce the domains of all others, allowing the deterministic analysis to quickly find a contradiction or determine that no improving solution can exist in the subproblem. Alternatively, choosing the wrong variables can lead to exponentially longer run times.

Due to the critical importance of the selection of the branching variable and value, there have been a number of heuristics presented [7, 8, 9]. Several of these are based on simple rules, for example, Most/Least Infeasible Branching base their decisions on the variable's fractionality, i.e., a value which indicates how far the current variable is from its nearest integer value for a linear relaxation. Other heuristics, like Pseudo cost Branching, can adapt over time while others, like Strong Branching, test which of the fractional candidates gives the best progress before actually committing to any of them. Finally, there are also hybrid techniques, like Reliability Branching, that put together the positive aspects of Strong Branching and Pseudo cost Branching. A good overview of these and other heuristics can be found in [9].

Typically, the available MIP solvers offer many parameters that define which general solution approach to be applied. In particular, some of these values define which heuristics are active. Often, selecting a set of heuristics enables the selection of several other parameters, that allow users to adapt the algorithm to their particular scenario. As an example, consider CPLEX [10], the most widely used commercial optimisation tool for solving MIP problems. Its version 12.5, used in this thesis, has more than 80 parameters that affect the solver's search mechanism and can be configured by the user.

Work with portfolios, where solvers with many configurations of these parameters can be employed, has shown that there is often no single solver or approach that works optimally on every instance [11, 12, 13]. Therefore, there are techniques that try to simulate an oracle which returns an optimal assignment of values for the

solver's parameters. This field of study is called *algorithm configuration* or *algorithm selection* [14, 15], depending on whether the focus is, respectively, on choosing the configuration of a single parameterised solver or choosing a solver among a portfolio of available ones. In both cases, the choice is critical and could determine huge differences in the solving time and in the solution quality.

Generally, the majority of the solvers tend to only use one set of heuristics throughout the search. However, throughout the branching process, as certain variables get assigned and the domains of others are changed, the underlying structure of the subproblems changes. A possible consequence of having a different problem structure is that the heuristic with the best performance, for the current subproblem, may be not the same one that was used in the beginning of the solving process. Therefore, efficiency of the search can be much improved using the correct heuristic at the correct time in the search. This thesis shows how to identify changes in the problem structure and therefore how to make a decision of when it is the best time to switch the employed guiding heuristic.

While a similar approach was recently introduced in [16], this work expands the research from the set partitioning problem with problem dependent heuristics, to the much more general problem of MIP. A detailed analysis of how the problem structure changes over time and a demonstration of the effectiveness of the proposed approach is also provided.

This thesis is organised as follows. The second chapter gives basic definitions and a brief introduction into MIP, branch-and-bound (B&B), and several heuristics. The third chapter introduces the related works, with a particular focus on algorithm selection approaches. The fourth and the fifth chapters present the motivations behind this work and the procedure that realises it, divided in offline procedure and online algorithm. Next, the implementation details and the computational results are presented and discussed.

Chapter 2

Background

The following chapters will present a dynamic algorithm selection approach. This will be applied using solvers that differ in the selection mechanism of the MIP branching rule. In order to motivate this work, we will introduce the concept of MIP, of branch-and-bound, and of branching heuristic, giving a description of the most used ones. Next, this dissertation presents related works about algorithm selection, which helps to define the state-of-the-art context that we aim to improve. Furthermore, we will describe and analyse our working space assisted by pictures obtained using dimensionality reduction techniques. Finally, we will present the set of MIP instances on which we performed tests and observations on the obtained results.

2.1 MIP

This section provides definitions of the most important terms used in this thesis. For a detailed description into linear and integer programming see [17, 18, 19].

Definition 2.1. Let $m, n \in \mathbb{R}$, $A \in \mathbb{R}^{m*n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, $l, u \in \mathbb{R}^n \cup \{\pm\infty\}$, and $I = \{1, ..., n\}$, $I \subseteq \mathbb{N}$.

$$\begin{aligned} &maximise : c^T x \\ &\text{subject to} : Ax \leq b \\ &\qquad\qquad l \leq x \leq u \\ &\qquad\qquad x_j \in \mathbb{Z}, \quad \forall j \in I \end{aligned}$$

is called a mixed integer program (MIP).

The function that has to be maximised, $c^T x$, is called the *objective function*. l and u are called the *lower* and *upper bounds* of the *variables* x. The constraints $l \leq x \leq u$ are called the variables *bounds*, and the last line of Definition 2.1 represents the *integrality constraints* of the MIP problem. A row A_i of the matrix A is often identified with the linear constraint $A_i x \leq b_i$. Let $B := \{j \in I \mid l_j = 0, u_j = 1\}$. We call $\{x_j \mid j \in I\}$ the set of *integer variables*, $\{x_j \mid j \in B\}$ the set of *binary variables*, $\{x_j \mid j \in I - B\}$ the set of *general integer variables*, and $\{x_j \mid j \in \mathbb{N} - I\}$ the set of *continuous variables*.

Definition 2.2. A MIP given in the form of Definition 2.1 is called:

$$\begin{aligned} \text{a linear program (LP)} \quad & if \quad I = \emptyset, \\ \text{an integer program (IP)} \quad & if \quad I = \mathbb{N}, \\ \text{a binary program (BP)} \quad & if \quad B = I = \mathbb{N}, \\ \text{a mixed binary program (MBP)} \quad & if \quad B = I. \end{aligned}$$

Definition 2.3. Let $\hat{x} \in \mathbb{R}^n$. Referring to Definition 2.1, we call $\hat{x}$:

$$\begin{aligned} \text{LP-feasible} \quad & if \quad A\hat{x} \leq b \text{ and } l \leq \hat{x} \leq u, \\ \text{integer feasible} \quad & if \quad \hat{x}_j \in \mathbb{Z} \ \ \forall j \in I, \\ \text{a feasible solution} \quad & if \quad \hat{x} \text{ is LP-feasible and integer feasible,} \\ \text{an optimal solution} \quad & if \quad \hat{x} \text{ is a feasible solution and } c^T\hat{x} \leq c^T x \\ & \qquad \text{for all other feasible solutions } x. \end{aligned}$$

The terms *LP-infeasible*, *integer infeasible*, and *infeasible solution vector* are defined analogously. If a MIP is given, the LP which arises by omitting the integrality constraints is called the *LP-relaxation* of the MIP.

Definition 2.4. Let $x \in \mathbb{R}^n$ and I the index set of integer values of a given MIP. We call $f(x_j) := |x_j - \lfloor x_j + 0.5 \rfloor|$ the fractionality of the variable x_j, $j \in I$ and $f(x) := \sum_{j \in I} f(x_j)$ the fractionality of the vector x.

Obviously, a vector x is integer feasible if and only if $f(x) = 0$. A variable $x_j, j \in I$, with $f(x_j) \neq 0$ is called *fractional*.

2.1.1 Branch-and-Bound

Linear programming based branch-and-bound (B&B) algorithms are currently among the most successful methods to solve MIPs. The B&B is a general algorithm for finding optimal solutions of several kinds of optimisation problems. The idea is to partition the original problem into many simple subsets. This process is commonly represented using a tree structure, where each non-root node represents one of the obtained subproblems. This approach is strongly related to the divide-and-conquer principle. In particular, B&B algorithms have two main choices: how to split a problem (branching) and which subproblem to select next.

This dissertation uses the following notation: X_{MIP} denotes the set of feasible solutions of a MIP problem P, as it is introduced in Definition 2.1. The *linear programming relaxation* of P is obtained by removing the integrality constraints: $\bar{c}_{P_{LP}} = max\{c^T x \mid x \in P_{LP}\}$, where $P_{LP} = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. Furthermore, if $P_{LP} = \emptyset$ then $\bar{c}_{P_{LP}} = \infty$. Trivially, $\bar{c}_{P_{LP}} \geq c^*$, where $c^* = max\{c^T x\}$, since $P_{LP} \supseteq X_{MIP}$.

The ideas behind divide-and-conquer and branch-and-bound can be summarised as follows:

- Divide-and-conquer:
  - *Divide* a large problem into several smaller ones;
  - *Conquer* by working on the smaller problems and combine their solution.
- Branch-and-bound:
  - Solve the continuous relaxation of the original problem;
  - *Divide* (Branch): given the problem P_0, choose an integer infeasible variable x_p. Then, create two subproblems, P_1 and P_2, with added constraints $x_p \leq \lfloor x_p \rfloor$ and $x_p \geq \lceil x_p \rceil$, respectively;
  - *Conquer* (Bound/Fathom): if the optimal solution of the continuous relaxation of P_i is worse than any known feasible solution for the original

problem P_0, then P_i is discarded since the MIP subproblem P_i can't have a better solution than its relaxation [18].

2.1.2 Branching Rules

MIP problems are commonly solved with linear programming based branch-and-bound (B&B) algorithms. These algorithms leave two choices: how to split a problem (branching) and which subproblem to select next. The success of the solver strongly depends on the strategy used to select the variable to branch on, that is called branching strategy or variable selection heuristic. For this reason, we focus on the branching strategy.

In order to split a problem P within a LP based B&B algorithm, the technique that split the feasible interval of a singleton variable. To be more precise, if i is a variable with fractional value $\bar{x}_i$ in the current optimal LP solution, two subproblems can be obtained, one by adding the trivial inequality $x_i \leq \lfloor \bar{x}_i \rfloor$ and one by adding $x_i \geq \lceil \bar{x}_i \rceil$, called respectively left and right *subproblems* or *children*. The two partial fractionality values are defined as ${f_i}^+ = \lceil \bar{x}_i \rceil - \bar{x}_i$ and ${f_i}^- = \bar{x}_i - \lfloor \bar{x}_i \rfloor$. The fractionality of the variable $\bar{x}_i$, as introduced in Definition 2.4, can also be computed as $f_i = min\{{f_i}^+, {f_i}^-\}$. This branching rule is also called *branching on variables*, because it only requires to change the bounds of variable i. This is the approach chosen by most of the available MIP solvers.

As discussed in detail in [7], Algorithm 1 represents the general algorithm for variable selection. Let Q be the current subproblem with an optimal LP solution

Algorithm 1 Generic variable selection

```
1: function variableSelection(Q, x̄)
2:     C ← getCandidatesIdx(Q, x̄)
3:     for all i ∈ C do
4:         s_i ← scoreEval(Q, x_i)
5:     end for
6:     return argmax_{i∈C}{s_i}
7: end function
```

$\bar{x} \notin X_{MIP}$ (recall that X_{MIP} is the set of feasible solutions). Given a set of branching candidates $C = \{i \in I \mid \bar{x}_i \notin \mathbb{Z}\}$, the idea is to compute on each one of its elements a score value (function *scoreEval*), for which a higher value indicates a better choice. The function returns the index i of the variable with the maximum score s_i.

In the following, the focus is on the most common variable selection rules, which are variants of Algorithm 1. The difference relies mainly in how the function *scoreEval* is realised. The common goal of these strategies is to solve an instance minimising, on average, the evaluation time. Therefore, as described below, it is inevitable to deal with the trade-off between the computational complexity of the function *scoreEval* and the number of nodes visited. In fact, usually a score function easy to be computed brings to a wider and more complete exploration. Instead, a more complex and slow to compute score function should bring to a more precise choice that permits a quick diving, with a reduced "horizontal exploration" of the tree.

The following are the most common branching techniques. The general ideas introduced here are described together with the implementation details in the following chapters.

Least/Most fractional rounding

This is very simple set of branching techniques consists of choosing the variable with fractional part closest to 0.5. The heuristic reason behind this choice is that this selects a variable where the least tendency can be recognized to which "side" (up or down) the variable should be rounded. Another possibility is to choose the variable with fractional part closest to 0, and in this case the idea is to select a variable that is "almost integer".

Pseudo costs based branching

This is a sophisticated rule that keeps the history of the variables on which branching has been performed. For each variable i, this information is stored in the two values ψ_i^+ and ψ_i^-, called pseudo costs, and derives mainly from the objective gain variation at each step of the B&B process [7]. The pseudo costs are then combined using a score

function that returns a numeric value for each branching candidate variable. The idea is to continue the B&B choosing the variable that maximises this score function. The behaviour of this branching heuristic adapts to the specific solving process, in fact the information on the past branching is used in the score evaluation. Therefore, if the root node's depth is zero, the nodes at higher depths of the solving tree have more collected history, therefore the heuristic choice is more reliable.

Strong branching

The idea of strong branching is to test which of the branching candidates gives the most progress before actually executing any branching operation. If the chosen candidate set C is the full set $C = \{i \in I \mid \bar{x}_i \notin \mathbb{Z}\}$ and if the resulting LPs has to be solved to optimality, this strategy is named *full strong branching*. Unfortunately, this look ahead operation requires high computation times per node. One possibility to speed-up this technique, is to restrict the candidate set in some way. Furthermore, often only a few simplex iterations are performed, because the change of the objective function in the simplex algorithm usually decreases with the iterations [7].

Hybrid Strong/Pseudo cost Branching

The computation times per node of *full strong branching* is high. The speed-up provided by the techniques indicated in Section 2.1.2 can be relevant, but the trade-off between speed-up and decisions precision greatly limits this approach. On the other hand, *pseudo costs branching* is weak at the very beginning of the solving process, since the decisions are taken with respect to pseudo cost values that, at the start, contain no relevant information. To circumvent these drawbacks, the positive aspects of *pseudo cost branching* and *strong branching* are put together in the combination *hybrid strong/pseudo cost branching*, where *strong branching* is the upper part of the solving tree and, from a given depth d, *pseudo costs branching* is used. Alternatively, *strong branching* can be used just for variables with uninitialised pseudo costs. In this case, the resulting *strong branching* estimates are used to initialise the pseudo costs.

Reliability branching

The idea described in Section 2.1.2 can be generalised by not only applying *strong branching* on variables with uninitialised *pseudo cost* values, but also on variables with *unreliable pseudo cost* values. A variable i has *unreliable pseudo cost* values if the condition $min\{\eta_i^+, \eta_i^-\} < \eta_{rel}$ is true, where η_{rel} is a threshold parameter and the two values η_i^+ and η_i^- count how many times, in the overall solving process, respectively the upward and downward branching on the variable i has already been solved and was feasible. This technique relies on the assumptions that *strong branching* tends to make the variables' pseudo costs reliable and *pseudo costs branching* performs effective choices, when they are based on reliable pseudo costs.

Inference History Branching

The inference history of a variable is a record of how many inferences have been discovered as a result of branching on this variable in the past. These inferences take the form of variables counters, whose domains have been effected by LP bounds propagation or domain propagation that might have happened during pre-solving / insolving. These values can be used instead of the pseudo costs in order to evaluate a score function.

Random Branching

When developing a new branching technique, the results are often compared with the execution times while using a *random branching* rule. The trivial idea is to branch, at each step, on a variable selected randomly among the infeasible ones, using a uniform probability distribution.

2.2 Clustering

Cluster analysis or clustering is a general concept that can be defined as the task of grouping a set of elements according to a *similarity measure*. Each resulting group is

called a cluster and its elements are more similar to each other than to those in other groups. Furthermore, the clusterisation is defined in a n-dimensional working space which uses a specific a distance metric. In this space, each cluster is associated to a specific point, called center

Clustering is a common technique for statistical data analysis and data mining. It does not refer to a specific algorithm, but to the general task to be solved. In this dissertation, clustering refers to the *unsupervised learning* approach that works on MIP problems, where every instance is represented by its *feature vector*. The latter refers to an n-dimensional *feature space* that describes structural information of an instance. In the case of MIP problems, the same feature space can be used for any MIP problem. Some examples of features specific to the MIP problem set could be the number of variables in the objective function, the percentage of integer infeasible variables (i.e., variables that don't satisfy the integrality constraint, as introduced in Section 2.1), or the number of constraints.

The features have to capture the differences between distinct instances. Therefore, a distance metric that represents a *similarity measure* is defined. For this purpose, the Euclidean distance between instances is commonly used [20].

From the many clustering techniques available, this dissertation presents two algorithms: *k-means*, an algorithm that has shown to offer good results for algorithm selection problems [11], and *g-means*, a clustering approach based on *k-means* with a higher level of automation.

2.2.1 k-means

One of the most straightforward clustering algorithms is Lloyds *k-means* [21]. The algorithm first selects k random points in the feature space, where k is a given parameter. It then alternates between two steps until some termination criterion is reached. The first step assigns each instance to a cluster according to the shortest distance to one of the k points that were chosen. The next step then updates the k points to the centers of the current clusters. While this clustering approach is very intuitive and easy to implement, the problem with *k-means* clustering is that it requires the user to specify the number of clusters k explicitly. If k is too low, this means that

some of the potential is lost to tune parameters more precisely for different parts of the instance feature space. On the other hand, if there are too many clusters, the robustness and generality of the parameter sets that are optimized for these clusters is sacrificed. Furthermore, for most training sets, it is unreasonable to assume that the value of k is known.

2.2.2 g-means

g-means [22] is a clustering technique proposed by Hamerly and Elkan in 2003. This approach has the purpose of solving the problem of *k-means* related on the choice of the parameter *k*. In fact, the algorithm automatically returns the clustered space, without taking as a parameter the interested number of clusters.

This work proposes that a good cluster exhibits a Gaussian distribution around the cluster's center. *g-means* starts considering all the instances as forming one large cluster. In each iteration, one of the current clusters is picked and is assessed whether it is already sufficiently Gaussian. To this end, g-means splits the cluster in two by running *2-means* clustering. All points in the cluster can then be projected onto the line that runs through the centers of the two sub-clusters, obtaining a one-dimensional distribution of points. *g-means* now checks whether this distribution is normal using the widely accepted statistical Anderson-Darling test [23]. If the current cluster does not pass the test, it is split into the two previously computed clusters, and the process is continued with the next cluster.

Among the many clustering techniques, *g-means* offers consistent results and permits the automation of the process, that is particularly important for our purposes.

2.3 Feature Filtering

It is well established that the success of a machine learning algorithm depends on the quality of its features. In fact, it is essential to have enough features to capture the differences between distinct instances, but too many features could introduce several problems. In particular, even when resources are not an issue, it is preferable

to remove unneeded features because they might degrade the quality of discovered patterns, for the following reasons:

- Some features are noisy or redundant. This noise makes the discovery of meaningful patterns from the data more difficult;
- To discover quality patterns, most data mining algorithms require much larger training data on high-dimensional data set. But the training data is very small in some applications. Therefore, having less dimensions enables to obtain quality results, even with a small amount of training data.

For example, imagine a large feature set of 1,000 values where only 10 of them are needed in order to completely describe a problem. In such a scenario, it is likely that the remaining 990 features are just random noise. Statistically, a noisy feature could accidently correlate to the output. Furthermore, reducing the feature set, therefore, requires less data to be stored and of fewer computations on it.

If only a subset of the feature set has useful information for building a model, it is possible to leave the remaining features out of the model. Feature selection techniques help in finding a quality solution which uses a small amount of data.

The idea of feature selection is to find a way to filter out features that have little chance of being useful in analysis of data. Generally these kind of filters are based on some kind of performance evaluation metric calculated directly from the data. In this case, the filter is based on a function that returns a relevance index $J(S \mid D)$ that estimates, given the data D, how relevant a given feature subset S is for the task Y. Through the computation of the relevance index for each individual feature $X_i, i = 1, ..., N$, it is possible to obtain a ranking order $J(X_{i_1}) \leq J(X_{i_2}) \leq ... \leq J(X_{i_N})$. The latter permits to filter out the features with the lowest ranks. In order to obtain a "good" ranking order, it is essential to define what *relevant* means.

Definition 2.5. A feature X is relevant in the process of distinguishing class $Y = y$ from others if and only if $\exists X = x \mid P(X = x) > 0 \wedge P(Y = y \mid X = x) \neq P(Y = y)$.

There are many state-of-the-art feature filtering techniques available. This thesis, which works with the R package FSelector [24], employes the *information gain* technique, that is based on information theory.

Chapter 3

Related Work

Given an optimisation problem and a portfolio of available solvers, algorithm selection is the problem of choosing a solver with optimal performance on the given instances. The outcome could be a solver of the initial portfolio (for example, the best single solver, that is the solver performing best in a training set of MIP instances), or it could be an algorithm which combines the available solvers in such a way to improve the performance of the best available solver in the portfolio. This chapter introduces different techniques at the state-of-the-art that aim to obtain a solving algorithm which improves the performance of the best solver in the portfolio.

3.1 The Algorithm Selection Problem

Many optimisation problems can be solved using several algorithms usually with different performance. Considering the set of all possible instances of a problem type, it has long been recognised that there is no single algorithm or system that will achieve the best performance in all cases [25]. This phenomenon is of great relevance among algorithms for solving NP-Hard problems, because the high variability between instances of a particular problem type [26, 27]. This is especially the case for MIP problems, which are the core of this thesis.

In this context, the ideal solution would be to consult an oracle that knows the amount of time that each algorithm would take to solve a given problem instance, and select the one with the best performance. In the last decades this issue has been referred to as the Algorithm Selection Problem [14, 15].

This problem, as first described by John R.Rice in 1976 [14] and presented by [15], has three main aspects that must be tackled:

- The selection of the *set of features of the problem* that might be indicative of the performance of the algorithm;
- The selection of the *set of algorithms* (often referred to as *solvers*) that together allow solving of the largest number of instances of the problem with the highest performance;
- The selection of an *efficient mapping mechanism* that permits to select the best algorithm to maximise the performance measure.

The features definition must be unique for all the instances of the same problem set. Furthermore, it is of extreme importance that they highlight the differences between distinct instances.

The set of algorithms (often referred to as a portfolio) can be exploited using several techniques that can be grouped into instance-oblivious and instance-specific algorithm selection.

3.2 Instance-Oblivious Algorithm Selection

Given a representative set of problem instances (training set), instance-oblivious algorithm selection attempts to identify the solver or the combination of solvers resulting in the best average performance on all the training data. After the training phase, for each approach in this group the execution follows the same rules independently of the particular instance being solved.

A trivial solution is to measure the solving time on the training set, and then to use the algorithm that offered the best performance (e.g. arithmetic mean, geometric mean, or median). As was shown in SNNAP [20], using this approach, simply called *winner-takes-all*, the single best solver might not be best on any instance.

A more elaborate solution consists of trying to solve each new instance with a sequence of algorithms, each one with a particular time-limit. The training phase

aims to identify this sequence of solvers and to assign an execution time-limit to each one of them. This approach is called *sequential portfolio* [28]. At least since the invention of CP-Hydra [12] and SatPlan [29], *sequential portfolios* also schedule solvers. That is, they may select more than just one constituent solver and assign each one a portion of the time available for solving the given instance.

3.3 Instance-Specific Algorithm Selection

One of the main drawbacks of instance-oblivious algorithm selection is to ignore the specific instances, solving each new one in the same way. As already claimed, there is no single algorithm or system that will achieve the best performance in all the instances of a certain problem [25]. Therefore, selecting the solver that performs better on the specific instance could result in a technique that performs significantly better than any of the algorithms in the portfolio. The latter, that is called Virtual Best Solver (VBS), is an oracle-based portfolio approach, in fact it assumes the existence of an oracle which chooses the fastest solver for each instance.

Several different techniques of instance-specific algorithm selection have been developed to simulate this oracle, all based on the common assumption that instances prefer different solvers due to the variation in their structure. Therefore, it is possible to construct a vector of features that aims to represent these structural differences and permits the realization of a mapping mechanism between instances and best solving algorithm.

3.3.1 SATzilla

SATzilla [30] is an example of an algorithm portfolio approach applied to the propositional satisfiability problem (SAT). SAT is the problem of determining if there exists an assignment of values that satisfies a given Boolean formula. Obviously, it is equally important to determine whether no such assignment exists, which would imply that the result of the formula is FALSE for all possible variable assignments.

SAT is one of the most fundamental problems in computer science. This *NP*-complete problem is interesting both for its own sake and because other *NP*-complete problems can be encoded into SAT in polynomial time and solved by the same solvers. Since it is conceptually simple, significant research efforts have been put in developing sophisticated algorithms with highly-optimised implementations. Furthermore, the SAT competition benchmark [31] incentivises further work on this problem offering visibility to the best solvers. Overall, since its initial introduction in 2007, SATzilla has won medals at the 2007 and 2009 SAT Competitions.

The approach is based on a simple idea: given a new instance, the runtime of each solver in the portfolio A is forecasted using ridge regression. It is then possible to run the algorithm with the best predicted performance. Therefore, SATzilla uses a well-defined set of features specific for the SAT problem. The approach can be divided in two phases: training process and testing process, respectively called *SATzilla-Learn* and *SATzilla-Run* in Algorithm 2.

In the first part, the features F are filtered using forward selection (that tries to select the most important features), then they are expanded using all the quadratic combinations of the reduced feature set, and finally the forward selection is performed again. In Algorithm 2, this group of operations is executed by the function *FeatureChanges(F)*. The training phase also finds the two algorithms (*pre1* and *pre2*) that solve the most number of instances if each is given a 30 seconds time-out (*preTimeout*). The identified algorithms will be used as pre-solvers, with the goal of solving quickly the easy instances, reducing the risk of doing a bad choice for them while introducing a limited overhead (this approach has shown to improve the average result). Finally a ridge regression model is trained on the training instances T and the best subset of solvers to use in the final portfolio is determined, using a validation dataset V.

The testing process consists of executing the SATzilla strategy on each instance x in the testing set. In particular, the pre-solvers are executed sequentially. When an instance is unsolved, its features are computed and then SATzilla predicts the expected runtime of each solver. Finally, it runs the solver with the lowest predicted runtime.

Algorithm 2 SATzilla

function SATzilla-Learn(T, V, F, A)
  $(\bar{F}) \leftarrow FeatureChanges(F)$
  $(pre1, pre2) \leftarrow FindBestSolvers(T, 30s)$
  for all $i = 1, \ldots, length(A)$ **do**
    $models_i \leftarrow RidgeRegression(T, \bar{F}, A_i)$
  end for
  $(\bar{A}) \leftarrow PortfolioFiltering(A, V)$
  return $((pre1, pre2), \bar{A}, models)$
end function

function SATzilla-Run($x, \bar{A}, pre1, pre2, models, preTimeout$)
  $execTime \leftarrow ExecuteSolver(x, pre1, preTimeout)$
  if $execTime \geq preTimeout$ **then**
    $execTime \leftarrow ExecuteSolver(x, pre2, preTimeout)$
    if $execTime \geq preTimeout$ **then**
      $F \leftarrow FeaturesComputation(x)$
      $times[\,] \leftarrow PredictRuntime(x, \bar{A}, models)$
      return $ExecuteSolver(x, \bar{A}_{argmin(times)})$
    end if
  end if
end function

An issue of this approach is that it requires prior knowledge about the relationship between features and performance. It can be therefore effective when the studied instances are of a specific problem type, for which a wide dataset has already been provided and modelled. Since the prediction is on the solving time, the task of defining a feature set of high quality for a very general problem type could be very difficult. Furthermore, the time predictions are not accurate, but in the case of the SAT competition they were accurate enough to distinguish between good and bad solvers.

3.3.2 CP-Hydra

CP-Hydra [12] is an algorithm portfolio approach for Constraint Satisfaction Problems (CSP). It is well-known that in constraint programming, as for SAT problems, different solvers are better at solving different problem instances, even within the same problem class [15]. The idea is to manage the solving process with a scheduler, that defines the active solvers and the portion of time to assign to each one of them. In order to build the scheduler, CP-Hydra uses a *Case-Based Reasoning* (CBR) approach instead of building an explicit model of the problem domain. The idea is to store a set of past examples called *cases*, each one made up of a description of the past experience and its respective solution. The full set of past examples is called the case base.

In CBR problems are solved by using or adapting solutions of old problems [32]. This approach has a number of advantages. In particular, there is no need to detect and model general patterns over the entire problem space. Moreover, CBR has proven to be successful in solving *weak-theory* problems [12], in which the problem domain could be complex and not provide much information about its structure. For these reasons, CBR may be a good candidate for algorithm selection.

In the original dissertation [12], different kinds of schedule are presented:

- *Split schedule:* schedule giving each solver an equal portion of the total time (note that this is an instance-oblivious approach);
- *Static schedule:* schedule generated using the entire case base;

- *Dynamic schedule:* schedule generated using the k=10 nearest neighbours.

The core of CP-Hydra is the computation of the solver schedule, that is a function $f : S \mapsto \Re$ mapping an amount of CPU-time to each element of a set of solvers S. In the *static schedule* approach, CP-Hydra works with the whole case base. While using the *dynamic schedule*, instead, given a new instance a set C of k similar cases is extracted from the case base. This operation is executed by the case base reasoner. The idea is to obtain the schedule which maximises the number of cases in C that would be solved. Formally, given a set C of similar cases, a solver $s \in S$, and a time value $t \in [0..1800]$, The subset $C(s,T)$ is defined as $C(s,t) \subseteq C$, where $\forall c \in C(s,t)$, c is solved by s if given at least time t. The schedule f can be computed using the following constraint program:

$$max \mid \bigcup_{s \in S} C(s, f(s)) \mid; \tag{3.1}$$

$$\sum_{s \in S} f(s) \leq 1800. \tag{3.2}$$

Expression 3.1 can be refined by weighting the cases according to their similarity to the new instance. Let $d(c)$ be the distance of case $c \in C$ to the analysed instance, the objective function could become:

$$max \sum_{c \in \bigcup_{s \in S} C(s,f(s))} \frac{1}{d(c)+1}. \tag{3.3}$$

This problem is NP-hard as a generalisation of the knapsack problem. Therefore, the main drawback is that, even if it works well restricting the approach to 5 solvers and up to 50 neighbours, solving this problem to optimality could become very inefficient while using a larger solvers portfolio.

3.3.3 ISAC

ISAC, Instance-Specific Algorithm Configuration [11], combines a configuration method and unsupervised learning obtaining a high performance algorithm selection method.

Algorithm 3 ISAC

function ISAC-LEARN(T, F, A)
    $(\bar{F}) \leftarrow Normalise(F)$
    $(k, C, S) \leftarrow Cluster(T, \bar{F})$
    for all $i = 1, \ldots, k$ **do**
        $BS_i \leftarrow FindBestSolver(T, S_i, A)$
    end for
    return (k, C, BS)
end function

function ISAC-RUN(x, k, C, BS)
    $f \leftarrow FeaturesComputation(x)$
    $\bar{f} \leftarrow Normalise(f)$
    return $BS_j(x)$
    $j \leftarrow FindClosestCluster(k, \bar{f}, C)$
end function

Most algorithms have several parameters that affect the performance. In order to have a fast execution, these parameters need to be tuned. Therefore, the goal is to find an assignment of values that guarantees the best performance possible. ISAC does that by exploiting the genetic algorithm GGA (Genetic Gender-based Algorithm) [33]. The motivation behind ISAC is that by having a portfolio algorithm it is also possible to tune it, so that it can classify a new instance by itself and choose the most promising parameters for that specific input automatically.

The portfolio algorithm that ISAC propose is based on clustering the inputs. As described in Algorithm 3, the overall process is divided into 2 phases: the learning (*ISAC-Learn*) and the runtime phase (*ISAC-Run*). In the learning phase, the input consists of a set of training instances T, their corresponding feature vectors F, and the parameterised algorithm A (that can be seen as a collection of solvers, if only a limited combination of values is admitted). First, the features are linearly normalised in order to have values that span the interval [-1,1], memorising the scaling and translation

values (s,t) for each feature. Then, a clusterisation on the set of normalised feature vectors is performed. The algorithm used is *g-means* [22], which returns a set of k clusters S_i represented by their centres C_i in the normalised feature space. The final step consists in computing favorable parameters for the parameterised solver A. For this purpose, the instance-oblivious tuning algorithm GGA [33] has been selected. In this scenario, unlike alternate approaches like k-nearest neighbour, clustering allows us to tune solvers offline since, given a training set of instances, it works on a specific grouping of instances which does not depend on the instance to be solved.

In the second phase, given a new input instance, its features are computed and then normalised, using the values (s,t) previously stored. Then, for this normalised feature vector x the cluster with the nearest centre is determined. This step is realised using the Euclidean distance as distance metric. Finally, the algorithm A is executed using the parameters for the identified cluster.

Instance-Specific Algorithm Configuration (ISAC), is a general approach that can tackle several kind of problems. In our context, in order to do algorithm selection, the parameterised algorithm A is the portfolio algorithm, which takes as parameters a value for each cluster that identify the solver to use for the input instances that belong to it. Each solver could be identified by a single parameter (e.g. solver 1, solver 2), or by a set of parameters (that could express, for example, the solver identifier followed by its parameters).

One of the major drawbacks of ISAC, however, is its dependence on the feature vector it uses to differentiate the problem instances. The success of the feature vector hinges on its ability of correctly grouping instances that are likely to behave similarly under the same solver. Another issue is that once the clusters are defined the approach is committed to them. Therefore, if the features were erroneous or, given new input data, if it is shown that there could be a better clusterisation, ISAC will have sub-optimal performance. A new approach that tackle these issues has been recently proposed in [34].

3.3.4 3S

3S, SAT Solver Selector, has been the best-performing *sequential* dynamic portfolio at the SAT Competition 2011 [28, 31]. 3S extends ideas behind ISAC by combineing solver scheduling and dynamic clustering. The latter is realised using a nearest neighbour methodology. In particular, it is defined a working space that, together with a distance metric (in this case the Euclidean distance), aims to offer a measure of similarity between instances. For this purpose, 3S uses the same 48 core features as SATzilla in 2009 [30].

3S works in two phases: an offline learning phase, and an online execution phase.

- *At Runtime:* In the execution phase, 3S first computes the feature vector of the given problem instance. Given $k \in \mathbb{N}$, computed offline, 3S selects k instances that are most "similar" to the given one in a training set of SAT instances. It then selects the solver that can solve most of these k instances within the given time limit. Finally, 3S runs a fixed schedule of solvers for 10% of the time limit and then it runs the selected solver for the remaining 90% of the available time.

- *Offline:* Given a training set of SAT instances, for each one of them 3S computes the correspondent feature vector and it executes each solver, storing their execution times. Using cross validation by random subsampling, 3S repeatedly splits the training set into a base and a validation set and it determines which size of k results in the best average performance on the validation set, when using only the base set to determine the optimal solver. Finally, 3S computes the fixed schedule of solvers that run for 10% of the time-out. The goal is to maximise the number of instances that can be solved within a reduced time limit, in this case given by the SAT Competition. This maximisation problem can be modelled and solved as an Integer Problem (IP). In particular it can be solved as a Set Covering Problem (SCP).

A main issue of this approach is that the solvers in the scheduler pre-solver (10% of the solving time-out) work independently, without passing information. Another problem regards the choice of the long running solver, that can't be corrected if it turns out to be sub-optimal during the solving process.

In 2012 a work that generalises the 3S technology for development of parallel SAT solver portfolios was published [28].

3.3.5 SATzilla 2012

SATzilla2012 [13] is an improved version of SATzilla that performs algorithm selection based on cost-sensitive classification models [35]. The main improvement is the new algorithm selection procedure. The previous version uses *empirical hardness models* [36, 37] in order to predict the time required for an algorithm to solve a given instance. This was an intuitive way to compare solvers in an instance-specific context. The idea behind SATzilla2012 is to compare every pair of solvers predicting which one will be better for the current instance, and finally select the solver with the majority vote.

SATzilla2012 constructs a classification model (decision forest, DF) offline for predicting whether the cost of computing the feature vector is too expensive, given the number of variables and clauses in an instance. Moreover, it constructs a cost-sensitive classification model (DF) for every pair of solvers in the portfolio, predicting which solver performs better on a given instance based on the feature vectors. Then, in order to solve a given instance, SATzilla2012 predicts online the feature vector computation time. If the latter is too costly then the backup solver is executed, which is the algorithm that achieves the best performance on the training set. Otherwise, SATzilla2012 performs a pre-solving phase and, if the instance is not solved yet, computes the features' values and selects the algorithm to run. In particular, for every pair of solvers, it predicts which one performs better using the DF trained offline, and it casts a vote for it. Finally, the selected solver is the one that receives the highest number of votes.

The main drawback of this methodology is that it is not sustainable once the number of solvers continues to grow, because it trains a model for every pair of solvers in the portfolio.

3.4 Non-Model-Based Search Guidance for SPP

The idea behind of Instance-Specific Algorithm Configuration (ISAC) is combined in [16] with a dynamic branching scheme that bases the branching decision on the features of the current subinstance to be solved. This approach uses a set of solvers, that differs just in the parameter that identifies the applied branching heuristic. Having a set of features that are representative of the specific MIP problem SPP (Set Partitioning Problem), the idea is essentially an extension of the ISAC approach: the feature space is clustered based on a set of training instances. Given a new instance and its feature vector, it is possible to identify which cluster it belongs to. Having an assignment of solvers for the clusterisation, the one assigned to the identified cluster is applied. This approach applies this selection before the solver execution and also during the solving process.

The idea to adapt the search dynamically during the solving process takes inspiration from [38, 39]. In particular, in [38] a value selection heuristic for Knapsack was studied and it was found that accuracy of search guidance may depend heavily on decisions higher in the search tree, since they can have effects on the distribution of subinstances that are encountered deeper in the tree. The latter clearly creates a serious chicken-and-egg problem for statistical learning approaches: *the distribution of instances that requires search guidance affects the choice of heuristic but the latter then affects the distribution of subinstances that are encountered deeper in the tree*. In [39] a method for adaptive search guidance for QBF solvers (the satisfiability problem of Quantified Boolean Formula) was based on logistic regression. The issue of subproblem distributions was addressed by adding subinstances that were encountered during previous runs to the training set.

Non-Model-Based Search Guidance boosts the CPLEX MIP solver to solve set partitioning problems faster. In particular, the following approach is proposed:

- First, the normalised feature space is clustered using the training instances;
- The CPLEX solver is parameterised, by leaving open the assignment of branching heuristic to cluster; It is then possible to consider each different configuration of the parameterised solver as a distinct solver;

- At runtime, whenever the solver reaches a new search node (or at selected nodes), the features of the subinstance are computed;
- The current cluster is identified and the next branching heuristic is selected.

The problem is then reduced to finding a good assignment of heuristics to the clusterisation. For this goal, a standard instance-oblivious algorithm configuration system is used, the algorithm GGA (Gender-based Genetic Algorithm) [33].

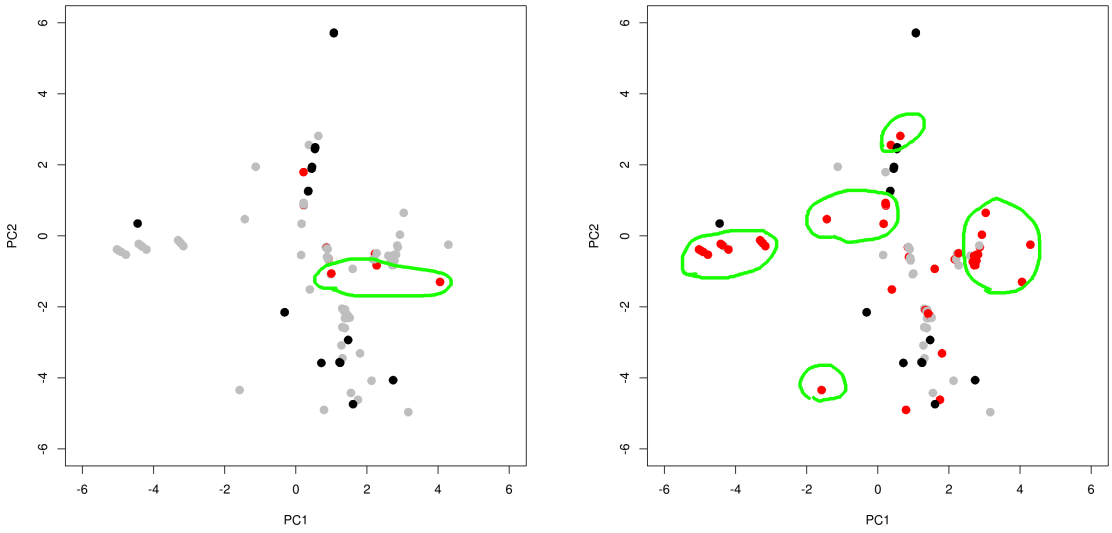
The results have proved the effectiveness of this approach. The main issue is that it is applied just to a single subset of MIP problems. In order to generalise this idea and build a solver that works with a wider set of problems, several issues need to be tackled, for example the identification of a new set of instances and a new set of features are essential. Moreover, other issues that need a solution are: which behaviour has to be studied in order to find an optimal approach, and which allows a clusterisation to be obtained that describes the data well. This thesis analyses and tackles these problems, introducing a dynamical methodology that works on the much wider MIP problem set.

Chapter 4

DASH

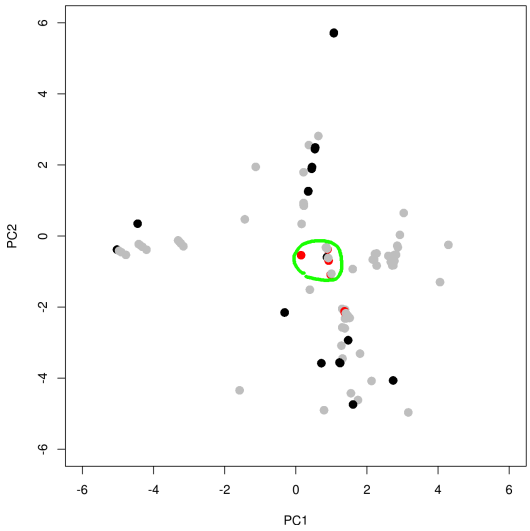
The objective motivating this work is to create a solver that dynamically adjusts its search strategy, selecting the most appropriate heuristic for the subinstance at hand. In a high-level overview, the solver performs a standard branch-and-bound procedure that, before choosing the next branching variable and value, analyses the structure of the current subinstance using a set of representative features. Working with this structural information, the solver would be able to predict that a specific heuristic has better performance than the alternatives, and employ it to make the next decision. We refer to such a strategy as Dynamic Approach for Switching Heuristics (DASH).

A number of MIP problems has been collected by selecting instances of different datasets, obtaining a large and heterogeneous set (see Section 5.4). Using structural information of the studied problems, a set of values, which captures as many aspects of a MIP problem as possible, has been selected. This set is called *feature space*. Employing Principal Component Analysis [40] (PCA) as dimensionality reduction technique, it is possible to visualise the dataset in either 2D or 3D representations. Figure 4.1 shows these representations with additional information: each plot employs a colouring scheme where red indicates an instance solved quickly (in less that 10 times of the best available solver execution time on the same instance), grey a slower solving performance, and black indicates that the algorithm timed-out. In each of the four pictures, the colours refer to the performance obtained using a specific branching rule, respectively least fractional and highest objective rounding (LFHO), most fractional and highest objective rounding (MFHO), pseudo cost branching weighted score (PW), and pseudo cost branching product score (P). The other heuristics available in the CPLEX implementation (see Section 5.3) do not help in this analysis because of their

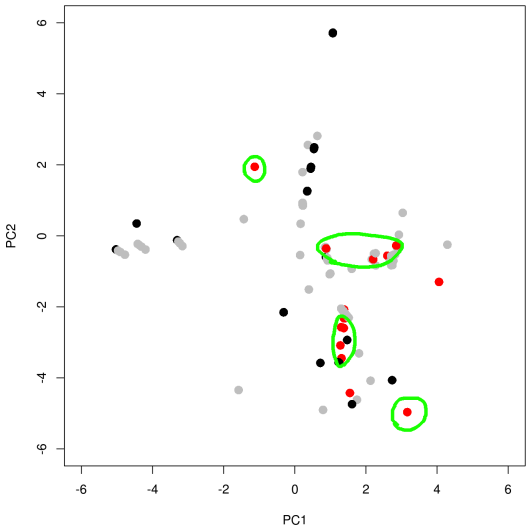


(a) Branching rule: LFHO

(b) Branching rule: MFHO



(c) Branching rule: PW



(d) Branching rule: P

Figure 4.1: PCA of the instances (at the root node) in the dataset. The colour scheme refers to the solving time using CPLEX with a single branching rule. Each instance is represented by a point, which can be black, if the solver timed-out, red if it was solved quickly (in less than 10 times of the best available solver), or grey otherwise.

low performance, therefore they are not presented here.

Figure 4.1 shows, using green lines, that instances solved quickly with the same branching rule can be easily grouped in the 2D space. The bidimensional representation, that is a simplification of the data since it is produced with a dimensionality reduction technique, allows us to immediately see a possible space partitioning pattern. In fact, with the exception of a few overlappings, the groups highlighted in Figure 4.1 are complementary. Employing all the dimensions available and the algorithm *g-means*, some of these overlappings disappear and it is possible to obtain a good clusterisation. The latter is a partition of the space for which is possible to obtain a relative assignment of solvers. Therefore, the defined feature space, together with the Euclidean distance, could be a good map between problem structure and performance. Consequently, through the definition of a *similarity measure* as the combination of this feature space and a distance metric, it is possible to execute unsupervised learning on the data, grouping the instances in such a way that instances of the same cluster prefer the same algorithm.

From the results presented in the following chapters, it emerges that, having a clusterisation, an instance often changes clusters during the solving process. Moreover, the branching rule preferred by the clusters are often different. Therefore, if the feature space together with the distance metric, represents a good map from problem structure to solving performance, switching the branching rule when the instance changes cluster may give a significant improvement to the overall solving time.

4.1 Feature Space

The feature set captures the structural difference between distinct MIP problems and also between subproblems of the same instance. Likewise, it is essential that these features don't become too expensive to compute. To do this, the collected information is composed of statistic values related to the current subproblem, as was similarly done in [16]. Specifically, the features are:

- Percentage of variables in the subinstance;

Feature number	Feature description
1	% of vars in OBJ
2	% of C vars
3	% of I vars
4	% of B vars
5	% of C vars in OBJ
6	% of I vars in OBJ
7	% of B vars in OBJ
8	% of equality constraints
9	% of inequality constraints
10..13	nVars in each constraint: Average, Std, Min, and Max
14..17	nVars C in each constraint: Average, Std, Min, and Max
18..21	nVars I in each constraint: Average, Std, Min, and Max
22..25	nVars B in each constraint: Average, Std, Min, and Max
26..29	In how many constr is each variable: Average, Std, Min, and Max
30..33	In how many equality constr is each variable: Average, Std, Min, and Max
34	nVars current problem / original problem
35	nVars in OBJ current problem / original problem
36..39	Average, Std, Min, and Max infeasibility value
40	Depth at the current node

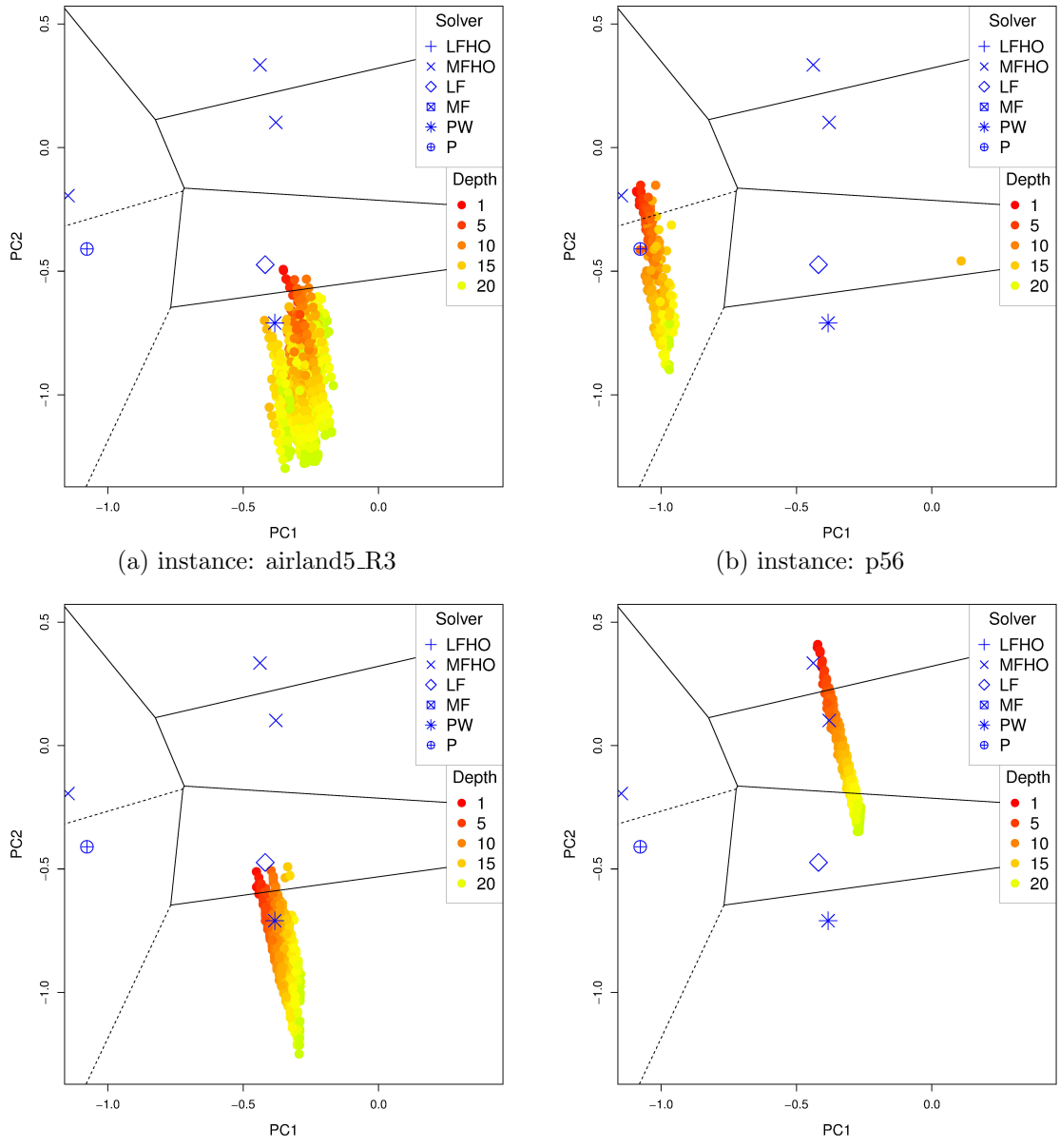
Table 4.1: Features description. The letters C, I, and B refers respectively to continuous, integer, and binary. *OBJ* indicates the objective function. *nVars* means "number of variables". *Constr* means "constraint".

- Percentage of variables in the objective function of the subinstance;
- Percentage of equality and inequality constraints;
- Statistics (min, max, avg, std) of how many variables are in each constraint;
- Statistics of the number of constraints in which each variable is used;
- Depth in the branch-and-bound tree.

Wherever a feature has to do with the problem variables, it is separately computed for each type of variable: i.e., continuous, general integer, and binary. Therefore, the resulting set is composed of 40 features. Table 4.1 shows the complete list of features.

4.2 Clustering the Instances

The feature space together with the Euclidean distance metric offers a measure of "similarity" between instances. Using this information, it is possible to exploit unsupervised learning techniques in order to group our data. In particular, having a representative set of MIP instances, equally partitioned in training and testing set, a clusterisation of the feature space is obtained from the training instances. The idea is that similar instances, that belongs to the same cluster, are likely to "prefer" the same solver [16]. Since the goal is to obtain a spatial representation that describes the possible evolutions of the instances during the solving problem, the clusterisation is performed, as previously proposed in [16], using an extended dataset (*extDataset*) that contains the original instances and a sample of their subinstances. A subinstance is represented by the feature vector computed at a (non-root) node during the branch-and-bound solving process. The considerations and results presented in this section refer to the dataset exhaustively described in Section 5.4. In particular, the three employed sets of instances are collected from our training set using, respectively, the commercial software IBM CPLEX [10], *extDatasetC*, and using the open-source software SCIP [9], *extDatasetS* and *extDatasetSr*.



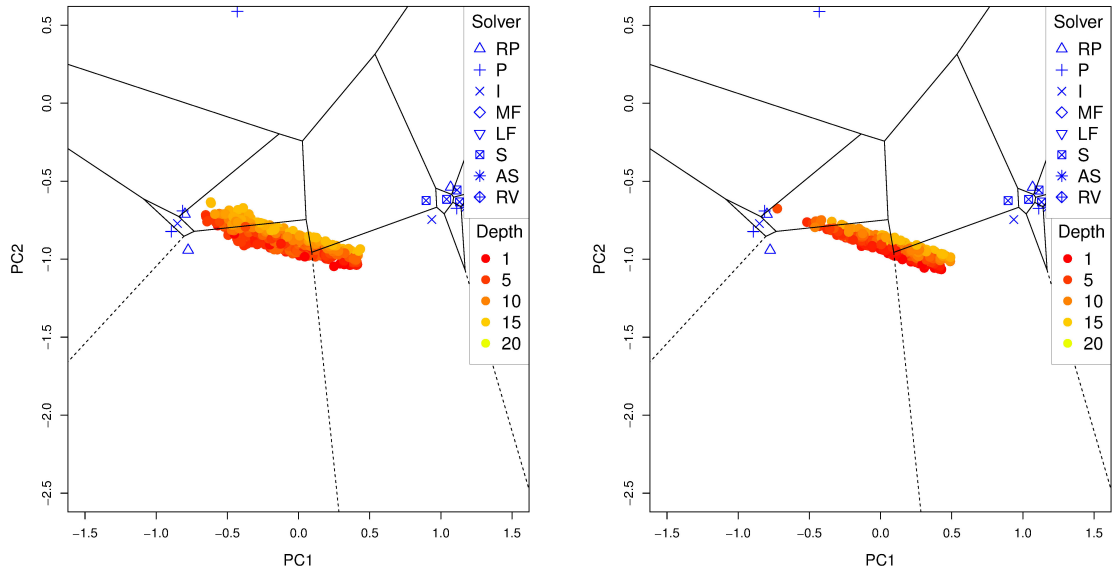
(a) instance: airland5_R3

(b) instance: p56

(c) instance: pmedcap_p3

(d) instance: regions-goods200-bids1000

Figure 4.2: Evolution of the ISAC solving process using CPLEX on four distinct instances. PCA of the clusterised feature space. For each instance, ISAC selects the solver assigned to its original cluster.



(a) instance: regions-goods100-bids500

(b) instance: regions-goods200-bids1000

(c) instance: mik.250-1-100.2

(d) instance: p51

Figure 4.3: Evolution of the ISAC solving process using SCIP on four distinct instances. PCA of the clusterised feature space. For each instance, ISAC selects the solver assigned to its original cluster.

Dataset	% subinstances
extDatasetC	56.7%
extDatasetS	51.9%
extDatasetSr	57.6%

Table 4.2: Percentage of subinstances that are in a different cluster from the one of their original instance. This information is computed on the extended datasets (instances + subinstances): *extDatasetC*, *extDatasetS*, and *extDatasetSr*, the first realised using CPLEX and the others using SCIP, and using respectively 6, 8, and 5 branching heuristics

A subinstance could have a significant distance from its original problem (the root node of the solving process). Therefore, there is a relevant chance that the branch-and-bound process moves an instance enough to change cluster. In fact, according to Table 4.2, more than 50% of the collected subinstances are in a different cluster than their root node. Figure 4.2 and Figure 4.3 show the solving evolution of ISAC in the feature space of a few representative instances, using respectively CPLEX and SCIP. In the pictures, the lines represent the cluster bounds, each point an instance at a specific depth of the solving tree, and each blue symbol indicates a cluster centre and the heuristic assigned to it. From these figures, we can deduce that the solving process moves the problem to a different cluster with a high probability. Furthermore, Figure 4.2 and Figure 4.3 show that the solving process usually brings a gradual change. There are a few exceptions, probably because of the effect of feature filtering and of operations like *restart* or *backtrack*, for which the change could be drastic. In this case, the structure of the problem obtained could be very different from the one of the previous one. For example, Figure 4.2 (b) and Figure 4.3 (d) have a few outliers that have a relevant distance from the other points. Even if the experiments show that this anomaly doesn't affect the approach introduced here, a better understanding of its causes could be an interesting direction for future research.

Given a subinstance, the general idea is to determine to which cluster it belongs and, having an assignment of heuristics for the clusterisation, to apply the relative solver. Knowing that the evolution in the branch-and-bound tree is usually gradual, this idea can be relaxed in practice, checking the current position and cluster just at

extDatasetC	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**				
inst+subinst	19	6	5	4	4	26	19	7	5	5				
inst	13	15	-	-	8	48	-	16	-	-				
extDatasetS	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
inst+subinst	11	2	9	2	24	25	2	7	7	2	2	3	2	2
inst	10	-	15	-	-	55	-	7	12	-	1	-	-	-
extDatasetSr	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**		
inst+subinst	11	13	29	17	2	3	3	1	6	1	1	12		
inst	10	-	70	-	-	-	-	-	-	-	-	20		

Table 4.3: Distribution of instances and subinstances in the clusterisation, expressed in percentage values. This information is computed for each extended dataset, using the clusterisation obtained with *g-means*.

selected depths, while for the other nodes the solver assigned to their parents can be applied.

Table 4.3 shows the distribution of instances and subinstances for each clusterisation, given the corresponding extended dataset. It is important to underline that there is no relation between clusters of different extended dataset (for example, cluster 6 on *extDatasetC* and cluster 6 on *extDatasetS* have a void intersections). For each extended dataset the distribution of all its elements (instances and subinstances) is shown among the clusters and in which clusters the original instances are contained.

From the information in Table 4.3 it emerges that the extended datasets tend to partition the space also describing the dynamic aspects of the MIP solving process. For example, cluster 7 of *extDatasetC* contains 19% of the whole data, but none of the original instances belongs to it. This means that the subinstances in cluster 7, that are "similar" since they are grouped together, could prefer a certain solver different from the one selected for the original instances. Furthermore, the subinstances of cluster 7 derives from instances that belong to several other clusters, therefore, in general, with a different preferred solver. The same considerations can be made on cluster 5 of *extDatasetS* and on cluster 4 of *extDataserSr*, that contains respectively the 24% and the 17% of the whole data but they have no original instance inside them.

The information discussed about Table 4.3 is further represented in Appendix A. The presented figures are bidimensional projections of the feature space, similarly to Figure 4.2 and 4.3. In particular, for each one of the studied extended datasets, the distribution of the instances and subinstances that "starts from the same cluster". A set of subinstances satisfies this condition if the root nodes of their elements belong to the same cluster. These figures show that in almost every case there is a relevant number of subinstances that belong to a cluster different from their original one, i.e., the cluster which contains their original instance, root node of the branch-and-bound solving process, giving a further proof of the high frequency of this event. Therefore, the idea of switching heuristics has reasons for being applied several times during the solving process, giving the opportunity to use the optimal heuristics for a specific subinstance, with the goal of obtaining a relevant speed-up.

4.3 Methodology and algorithm

The specifics of DASH (Dynamic Approach for Switching Heuristics) are described in Figure 4.4 and Algorithm 4. Modelled after the ISAC (Instance-Specific Algorithm Configuration) approach [11], DASH assumes that instances with similar features share the same structure and so will yield to the same algorithm. Therefore, these groups of instances are identified during an offline clustering procedure. DASH is provided with the current subinstance, the heuristic employed by the parent node, the centres of the known clusters, and the list of available heuristics. Because determining the features can be computationally expensive and because switching heuristics at lower depths of the search tree has a smaller impact on the quality of the search, DASH only chooses to switch the guiding heuristic up to a certain depth and only at predetermined intervals, employing the parent's heuristic in all other cases. When a decision needs to be made, the approach computes the features of the provided subinstance and determines the nearest cluster based on the Euclidean distance. In theory, any distance metric can be used here, but in practice the Euclidean works well in the general case. In the end, DASH, employs the heuristic that has been determined best for that cluster.

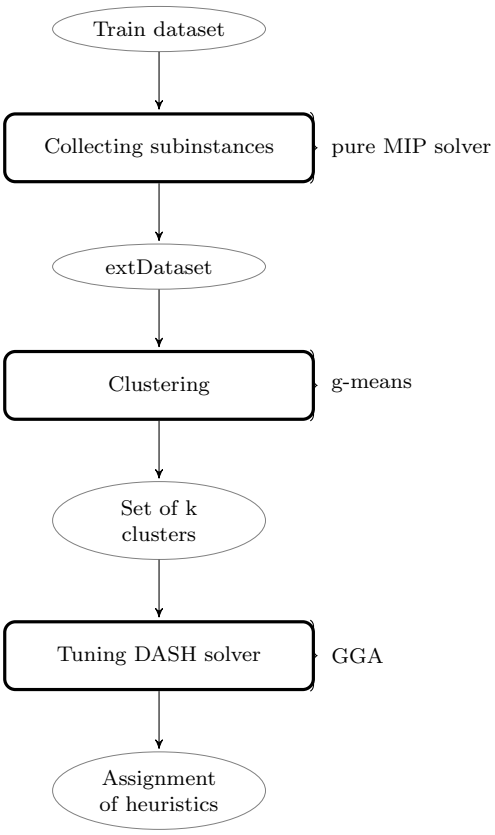


Figure 4.4: Offline part of the Dynamic Approach for Switching Heuristics (DASH), that outputs the information needed in the online part by the DASH solver.

As can be inferred from this algorithm, which describes the online part of the whole approach, the key component that determines the success or failure of DASH is the correct assignment of heuristics to the clusterisation. The offline procedure shown in Figure 4.4, similar to the one first described in [16], aims to find an optimal assignment of heuristics. For each instance in the training set, an assortment of subinstances is computed. These are observed when using each single heuristic in a solving process without any switch. This extended problem set (*extDataset*) allows us to get a better overview of the type of subinstances DASH will be encountering, as opposed to just using the original training instances. Computing the features of

Algorithm 4 DASH solver - called before branching

```
1: procedure DASH(subinstance, parent, centres, heuristics)
2:     if depth < maxDepth and depth % interval == 0 then
3:         x ← featuresComputation(subinstance)
4:         for all c in centres do
5:             distance_i ← EuclideanDistance(x, c)
6:         end for
7:         cluster ← argmin(distance)
8:         heuristic ← heuristics_cluster
9:     else
10:        heuristic ← parent.heuristic
11:    end if
12:    ExecuteBranching(subinstance, heuristic)
13: end procedure
```

the extended problem set, it is possible to cluster the instances using *g-means* [22], a general clustering approach that automatically determines the best number of clusters for the dataset in question (see Section 2.2.2).

Once all the subinstances are clustered, it must be determined which heuristic is best in which scenario. However, an important caveat to this is that the decision of using a heuristic for a certain cluster also affects all other decisions. This is because DASH can switch heuristics several times, and the types of subinstances observed after applying one heuristic will likely be different than when another one has been applied. Therefore, the parameter tuner GGA (Gender-based Genetic Algorithm) [33] is employed to simultaneously assign heuristics to all clusters, using the original instances as the training set. GGA performs several parallel executions of the DASH solver, employing different parameterisation. These parameters are:

- Assignment of heuristics for the clusterisation;
- Maximum depth value for the application of DASH (*maxDepth*);

- Interval of application of DASH. This parameter indicates at which depths a heuristic switch is enabled to occur.

Furthermore, this tuner permits to specify one or more random seeds for each of the training instances. Therefore, in order to have more possible training combinations, three different seeds have been randomly chosen for each instance. The final output of this offline procedure consists of a clusterisation of the feature space, represented by the cluster centres, and an assignment of solvers, one for each cluster. This data is then used in the online part of DASH, as shown in Algorithm 4, with the goal of determining which solver should be applied for the current subinstance.

4.4 Chapter Summary

This chapter presented motivations and components of the proposed dynamic approach for switching heuristics, DASH. The approach is divided in an offline part and an online algorithm. Firstly, it defines a feature space for MIP problems. Secondly, a heterogeneous dataset is built, collecting instances of several MIP datasets. Then, a portfolio of solving heuristics is selected and used for extending the dataset with a sample of subinstances. Next, the extended dataset is clustered into groups of instances that have similar features. Finally, an automatic parameter tuner is used for selecting a solver for each cluster and parameters related to the activation of DASH during the online part of the execution.

Chapter 5

Experimental Setup

In order to set the stage for DASH, three things are necessary. Firstly, there must be a descriptive *feature set* that can correctly distinguish between different classes of instances, while doing so with a minimal overhead. Secondly, we must have a diverse *set of heuristics* each of which performs well on different kinds of instances. Finally, there must be a *heterogeneous domain*, with a large number of benchmark instances. Since the feature set has already been introduced, this chapter touches on the remaining two components, together with the description of the machines used and the definitions related to the numerical analysis.

5.1 Measurements

The presented experiments commonly aim to compare different heuristics and to state how good they are "on average". The arithmetic mean is a comparison indicator which is easy to compute. However, since the values for comparison differ heavily in their magnitude (they usually lie in a range of 0 to 1,800 seconds), the arithmetic mean would only depend on the largest values. Therefore, a measurement more appropriate for this case is the shifted geometric mean (setting 10 seconds of shifting). In competitions, the time-out value usually lies in a range of 0 to 5,000 seconds. Therefore, the experiments have a time-out value of 1,800 seconds, which lies in the same interval and is also large enough to let at least one solver finish its process before having reached the time limit.

Since the experiments have a finite time-out value, it is useful to measure the

results also with an alternative comparison indicator, which takes into account that an instance could remain unsolved. This measurement, called Par10, is the standard penalised average measurement where when a solver times-out it is penalised as having taken 10 times the time-out value. Furthermore, the presented data includes the percentage of instances solved.

The results are evaluated comparing them to several benchmark values:

- Virtual Best Solver (VBS): the lower bound of what is achievable with a perfect portfolio that for every instance always chooses the solver that results in the best performance. In particular, VBS chooses from the available solvers those that use each single branching heuristic (pure solvers), in addition to this VBS_RAND uses the available random switched solvers, and VBS_DASH chooses from the pure solvers and our DASH solver. A random switched solver is an algorithm that, using an uniform probability distribution, at each node of the branch-and-bound tree selects a random heuristic from the available portfolio.

- Best Single Solver (BSS): the desired upper bound, obtained by solving each instance with the solver based on a single branching heuristic, whose average running time is the lowest on all the dataset.

- Instance-Specic Algorithm Conguration (ISAC): this is the pure ISAC methodology obtained with the set of features and clustering described in Section 3.3.3.

5.2 Technology

In order to realise DASH (Dynamic Approach for Switching Heuristics) and to show its effectiveness, the chosen strategy is based on a customisable MIP solver, a scripting language that provides libraries for statistical and graphical analysis, and a scripting language that permits automation of tests and analysis.

The chosen MIP solvers are the state-of-the-art commercial software CPLEX version 12.5 [10] and the open-source software SCIP version 3.0.1 [9, 41]. These solvers

are competitive with other free solvers like CBC [42], GLPK [43], MINTO [44], and SYMPHONY [45] and also with commercial solvers like XPRESS [46]. CPLEX and SCIP, to the best of our knowledge, are considered the best available, respectively, commercial and open-source solvers.

In order to obtain reliable results, CPLEX and SCIP have been executed in the single core mode. The experiments were run on dual Intel Xeon E5430 quad-core processors (2.66Ghz) computers with 12GB of DDR-2 FB-DIMM 667MHz memory.

The languages used for the solver implementation are Python and C and the scripts that execute multiple tests and collect the output data of the solvers have been written using Bash scripting. Moreover, the graphical and numerical analysis, have been done using the scripting language R.

5.2.1 CPLEX

In the CPLEX implementation, the only modified part is the built-in branching strategy, by implementing a branch callback function based on Algorithm 4. CPLEX does not give access to its source code, but it offers the opportunity to implement new heuristics through a callback, that is a function executed regularly at a specific point of the solving process (for example, before the selection of the variable for the branching operation). Because all the tested approaches require this branch callback to be enabled, the comparability of the results is guaranteed[1]. To the best of our knowledge, CPLEX does not offer the opportunity to change among its own heuristics during the solving process (the parameter that set the branching heuristic to apply is *mip.strategy.variableselect*). An alternative idea is to create a copy of the subproblem and to solve it with the new configuration. However, this approach slows down the solving process considerably, with the additional drawback of changing the solving path in a random way, even when disabling all the relevant pre-solving and heuristics.

The employed solution is to implement all the branching heuristics in the same callback function, with the main advantage of having comparable results. This function implements both Algorithm 4 and the branching heuristics described in the

[1]Note that CPLEX switches off certain heuristics as soon as branch callbacks, even empty ones, are being used so that the entire search behaviour could be different from the default version.

following section. The major drawback of using the callback is that CPLEX changes its parameter configuration automatically and it does not allow the use of the same heuristics of its default version. Moreover, the branch callback does not disable the CPLEX branching variable selection. In this way, CPLEX offers in the callback the choice to change strategy or to apply a new one, with the drawback of executing everytime the native CPLEX branching heuristic selected once at the beginning of the solving process. In order to minimise the overhead, the selected heuristic is the one with minimum computation time (Least Fractional Rounding (LF)). Even if it is not possible to compare the numerical results of the CPLEX default execution, the experiments show that DASH gives a relevant improvement on the heuristics implemented in the branch callback.

5.2.2 SCIP

SCIP [9, 41] is an open-source framework created to solve Constraint Integer Programs (CIPs) [9], which denotes an integration of Constraint Programming and Mixed Integer Programming. Since MIPs are a sub-category of CIPs, and since this thesis is related to MIP-solving, SCIP will be referred to as a MIP-solver.

SCIP was developed by Achtelrberg et al. [9]. It is structured as a framework that works mainly with external plugins. Its current version offers a bundle of MIP-solving plugins, e.g., pre-solvers, cut separators, and primal heuristics. Furthermore, the structure of the software easily allows the creation of a new plugin, in fact among the "native" plugins there are example source codes that are very helpful in the realisation of a new source file. The software is implemented in C, therefore the plugin that realises DASH has also been written in this language. The main source file, which manages the execution of the solver with the configuration of interest and organises the output properly, is instead written in C++.

While CPLEX has several limitations about the implementation, the open-source solver SCIP offers much more flexibility. In particular, SCIP assigns to each one of its built-in branching heuristics (called *branching rules*) a priority. Then, it selects the branching rule with the highest priority just before each branching operation. If it

fails, e.g. if it has reached the depth limit, the heuristic with the second highest priority (and so on) is applied. Furthermore, it is possible to change the priorities during the solving process and therefore to dynamically switch between native branching rules. While for CPLEX DASH is realised as a branching heuristic that changes its behaviour depending on the features of each studied node, in the case of SCIP, it is implemented as a new heuristic that decides if and when a branching rule switch will occur, setting properly their priority values.

5.3 Branching Heuristics for CPLEX experiments

In order to realise and test the Dynamic Approach for Switching Heuristics, a portfolio of six branching rules has been implemented for CPLEX. These heuristics are specific implementations of the generic branching rules described in Section 2.1.2.

Most Fractional Rounding (MF) One of the simplest MIP branching ideas is to select the variable that has a relaxed LP solution whose fractional part is largest, and to round it first. The driving reason behind this is to make decisions on variables that deterministic analysis is least certain about. Therefore, this heuristic strives to find infeasible solutions as quickly as possible.

Least Fractional Rounding (LF) Alternatively to MF, this technique selects the variable that has a relaxed LP solution whose fractional part is closest to an integer value, and it rounds it first. This is done to gently nudge the deterministic reasoning in whatever direction it is currently pursuing, with a smallest chance of making a mistake.

Least Fractional and Highest Objective Rounding (LFHO) This heuristic is based on the same motivation behind the Less Fractional branching. The idea is to branch on a variable with a small fractionality (fr) and a high objective value (obj). Such a variable can be found by an iteration that looks for the minimum value of fr, but updating the variable only if obj does not decrease. This means that, when branching on a variable k in [1,n], the following property

is guaranteed:

$$\forall i \in [1, n],\ fr_k \leq fr_i \ or\ obj_k \geq obj_i \tag{5.1}$$

Most Fractional and Highest Objective Rounding (MFHO) A modification of the previous approach is also used, but this time the focus is on the most fractional variables. In this case the guaranteed property is:

$$\forall i \in [1, n],\ fr_k \geq fr_i \ or\ obj_k \geq obj_i \tag{5.2}$$

Pseudo Cost Branching Weighted Score (PW) This heuristic is based on the pseudo costs, numerical values that estimate the variation in objective value for rounding up or rounding down, called respectively up-pseudocost and down-pseudocost. The pseudo costs of a variable can be combined in a score function (5.3) that returns a numeric value. This result is used to guide the branching, for which the variable that maximises this score is chosen. Further details can be found in [7].

$$score(q^-, q^+) = (1 - \mu) * min(q^-, q^+) + (\mu) * max(q^-, q^+),\ \ \mu = 1/6. \tag{5.3}$$

Pseudo Cost Branching Product Score (P) This approach is based on the same idea as PW. The difference lies in the score function, that is now the product of the two pseudo costs:

$$score(q^-, q^+) = q^- * q^+. \tag{5.4}$$

5.3.1 Branching Heuristics for SCIP experiments

The software SCIP supports a dynamic switch among the built-in branching rules during the solving process. Therefore, the portfolio of heuristics chosen for this set of experiments consists of the SCIP native branching rules. Among the eight pure

heuristics, three realise the same ones previously described for CPLEX: Most Fractional Rounding (MF), Least Fractional Rounding (LF), and Pseudo Cost Branching (P). The others are presented above.

Reliability Branching (RP) The pseudo cost of a variable is considered to be unreliable until it has been updated a number of times. With this heuristic, "unreliable" variables are selected for strong branching to initialise the pseudo costs with enough updates to make them more reliable. This technique, described in Section 2.1.2, is implemented using $\eta = 8$ as *reliability* parameter. The latter is a default parameter which is commonly chosen for this branching heuristic (see [7]).

Inference History Branching (I) The inference history of a variable is a record of how many inferences have been discovered as a result of branching on this variable in the past. These inferences take the form of counts of variables whose domains have been effected by LP bounds propagation or domain propagation that might have happened during pre-solving / insolving.

Full Strong Branching (S) In Full Strong Branching each variable of several (by default: 8) most promising variables have branches created for them, each branch's LP relaxation is solved and the branches of the variable which results in the best objective is proceeded with.

Full Strong Branching on all Variables (AS) In Full Strong Branching on all Variables a more exhaustive approach is taken where all possible branches are explored as before. This should find the best possible branch at each node but is very expensive.

Random Variable Branching (RV) This Random Variable heuristic involves choosing the branching variable randomly among the candidates at the current node, using a uniform probability distribution.

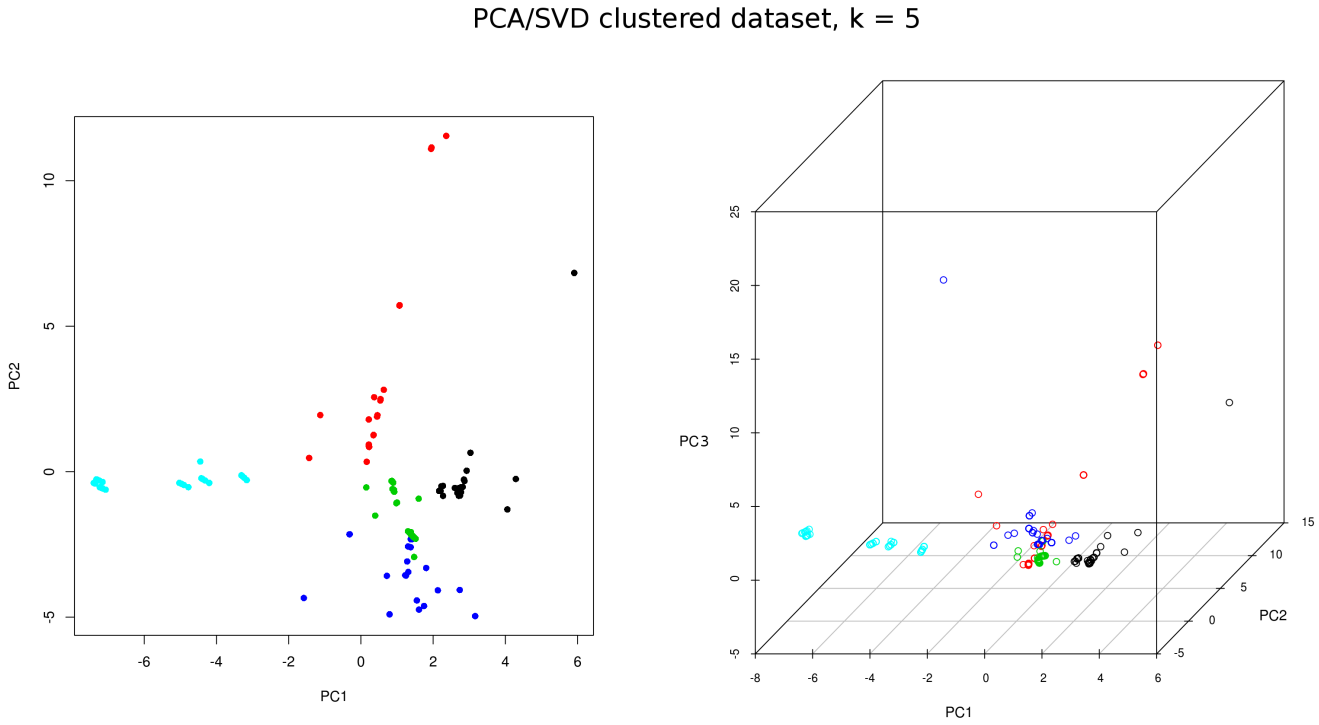


Figure 5.1: PCA of the whole dataset (training and testing sets), after a clustering operation using *g-means*. The resulting number of clusters (k) is 5. Each instance is represented as a point and coloured after its cluster.

5.4 Dataset

The analysis and the numerical results presented in this thesis refer to a single dataset, that consists of several different groups of problems. This dataset is made up of 341 instances, divided in 180 instances for the training set and 161 for the testing set. These instances have been collected using the following datasets: *miplib2010* [47], *fc* [48], *lotSizing* [49], *mik* [50], *nexp* [51], *region* [52], and *pmedcapv*, *airland*, *genAssignment*, *scp*, *SSCFLP* were originally downloaded from [53]. Each of the selected instances can be solved by at least one of the available pure heuristics in less than the 1,800 seconds of time-out. Furthermore, the dataset doesn't include instances that

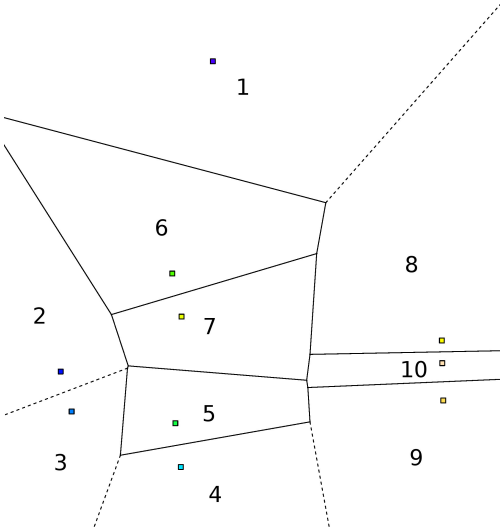
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Cluster 1	20	-	-	25	-	25	-	30	-	-	-	-
Cluster 2	-	45	-	-	-	14	41	-	-	-	-	-
Cluster 3	-	-	-	-	1	5	-	-	18	62	14	-
Cluster 4	-	-	49	-	-	7	-	-	-	-	-	44
Cluster 5	-	-	-	-	98	2	-	-	-	-	-	-

Table 5.1: Given the clusterisation obtained using *g-means* and the whole instances dataset, the table shows the distribution (percentage) of the instances in the clusters. The problem types are: 1: airland, 2: fc, 3:GenAssignment, 4: LotSizing, 5: mik, 6: miplib2010, 7: nexp, 8: pmedcap, 9: region100, 10: region200, 11: scp, 12: SSCFLP

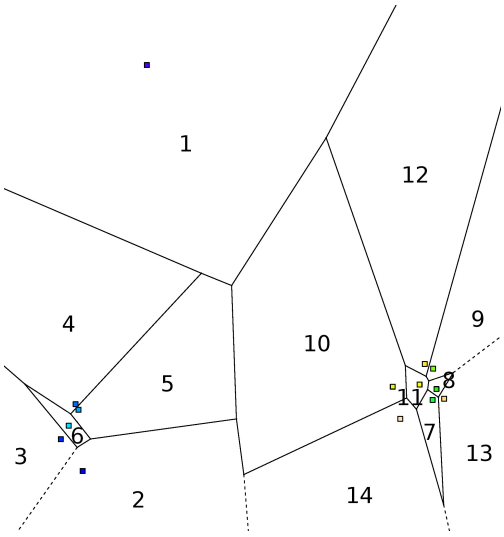
can be solved too easily; therefore:

- each instance will not be solved completely in the preprocessing phase;
- an instance solving time will be greater than 1s for at least one of the available heuristics.

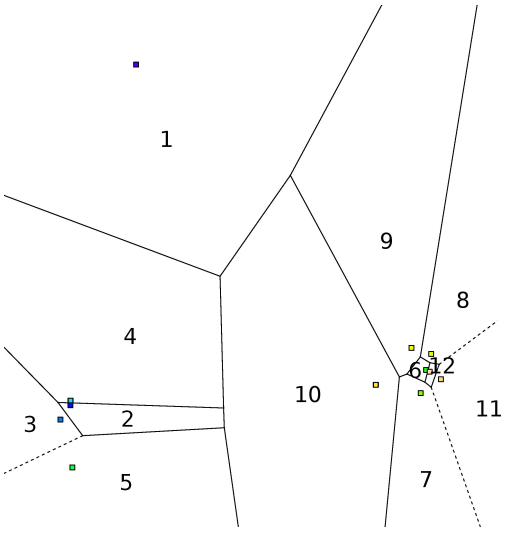
Figure 5.1 shows a bidimensional projection of the training set. The latter is clustered using *g-means*, and the obtained distribution of instances per cluster is shown in Table 5.1. Each row is normalised to sum to 100%. Thus for Cluster 1, 20% of the instances are from the *airland* dataset. From this table, a first observation is that there are not enough clusters to perfectly separate the different datasets into unique groups. However, this is not a problem as it is not the desired result. Instead the focus is in capturing similarities between instances, not splitting benchmarks. Looking at Table 5.1, the *region100* and *region200* instances are grouped together. Furthermore, Cluster 4 logically groups the *LotSizing* and the *SSCFLP* instances together. Finally, the instances from the *miplib*, those instances that are supposed to be an overview of all problem types, are spread across all clusters. This clustering therefore demonstrates that the dataset both has a diverse set of instances and that the employed features are representative enough to automatically notice interesting groupings.



(a) Dataset: *extDatasetC*



(b) Dataset: *extDatasetS*



(c) Dataset: *extDatasetSr*

Figure 5.2: Clustering obtained using *g-means* on, respectively, *extDatasetC*, *extDatasetS*, *extDatasetSr*. These datasets contain all the training instances and a collection of their subinstances, obtained using CPLEX or SCIP, as described in Section 5.4. The figures show also the cluster centres and the cluster identification numbers.

Using the procedure described in Section 4.3, given a set of branching heuristics and the instances dataset, it is possible to collect a sample of subinstances and to build an extended dataset. In particular, this thesis employs three extended datasets:

- *extDatasetC*: this extended dataset consists in all the original training instances, together with a subsample of subinstances collected using CPLEX and all the 8 branching heuristics (6 pure heuristics and 2 random switched) implemented in the branch callback, for a total of about 16,000 feature vectors.

- *extDatasetS*: it contains the original training instances and a subsample of subinstances collected using SCIP and all the 8 available branching rules, for a total of about 45,000 feature vectors.

- *extDatasetSr*: similar to the previous one, with the only difference of having used just 5 heuristics for collecting the subinstances. In particular the heuristics removed are the 2 that performed best on the training set, using the geometric mean as performance measure: reliability branching (RP) and pseudo cost branching (P). Moreover, the random variable branching rule has also been removed. The dataset obtained is of about 30,000 feature vectors.

Finally the three datasets have been reduced to 10,000 feature vectors each, because of memory limitations that the chosen implementation of *g-means* doesn't cover. The subsampling performed is mainly a random sampling operation with the constraint, given an instance, to keep the feature vectors at depth 0 and 1 and to keep, if possible, at least one other element for each branching rule applied.

The reason behind having distinct extended datasets relies on the fact that the two DASH implementations for CPLEX and SCIP use a different set of heuristics. Therefore, the behaviour of an instance during the solving process is probably different. In particular, the distribution of subinstances could change in such a way as to result in a different clusterisation. As it is shown in Figure 5.2, the SCIP and the CPLEX versions have completely different clusterisations. On the contrary, it is clear that the two groupings obtained for the SCIP versions are very similar. However, small difference in the clustering could cause significant variations in the overall

performance. In fact just turning off 3 heuristics, 2 small clusters are not identified anymore and the overall structure slightly changes, and this could lead to totally different path in the solving tree.

Next, the three obtained clusterisations are studied in order to obtain information about the feature space. In particular, the features are ranked using the *information gain ratio* [54, 55], obtaining a possible features importance index. From this analysis, the most important variables seem to be the statistics on the number of constraints and of the number of variables in each constraint (in Section 4.1 these are indicated as features 10..13, 26..29, 30..33). On the contrary, features specific to continuous, integer, or binary variables seem to be less important. An interesting result regards the feature *depth*, that turns out to be one of the least important. Assuming that the *information gain ratio* is a good choice for ranking the features in this case, this result means that it is not important at which depth a subproblem is, but the similarity with the other instances can be measured just using the other features, that are related on the subproblem structure. In fact, it is reasonable to think that two distinct solving processes, on the same original problem, could arrive sometimes at the same subproblem p_r, but at different depths. The conclusion is that p_r should be solved in the same way in both cases, without considering the difference in depth.

5.5 Chapter Summary

This chapter presented the implementation-specific details of DASH and introduced the technology and measurements employed for the experiments. The analysis and results have been performed on a heterogeneous dataset. Because of the different magnitude of the results on this dataset and because of the presence of a time-out value for the solvers, each of the chosen measurements for the comparisons puts the focus on a distinct aspect of the analysis. Finally, this chapter described also how the heuristics employed in the experiments are implemented, and it defined the distinct portfolios used.

Chapter 6

Numerical Results

With the described methodology, the main question that needs to be addressed is whether switching heuristics can indeed be beneficial to the performance of the solver. To test this, each of the implemented heuristics was run without allowing any switching (pure solvers), for each of the instances in our test set. Moreover, the random switched branching heuristics, specific to CPLEX and SCIP, were run in order to show what kind of advantages there are for the employed dataset. The numerical and graphical results of the DASH solver are then shown and analysed.

In the CPLEX case, the random switched solvers have been realised in the branch callback. The first one switches between all heuristics (*RAND 1*), while the second (*RAND 2*) switches only among the top four best heuristics (MFHO, MF, PW, and P). Instead the solver SCIP offers a native random branching rule, which selects the branching variable using a uniform probability distribution (*RANDvar*). Since the focus is on the branching rule switching, an additional random switched heuristic (*RANDheur*) has been implemented that randomly changes the branching rule, choosing from all the native non-random heuristics at every node of the branch-and-bound solving process. The results are summarised in Table 6.1, Table 6.3, and Table 6.4.

It can be observed that neither of the random switching heuristics described above perform very well by themselves, compared to the best single solver (BSS), that is the solver that performs best on average sticking to a single branching heuristic. However, observing the solving times of the virtual best solver (VBS), i.e., the solver that for each instance applies the heuristic that results in the best performance in the portfolio, and of VBS_Rand, i.e., the virtual best solver that includes the random switched heuristics, it emerges that VBS_Rand has better performance than VBS.

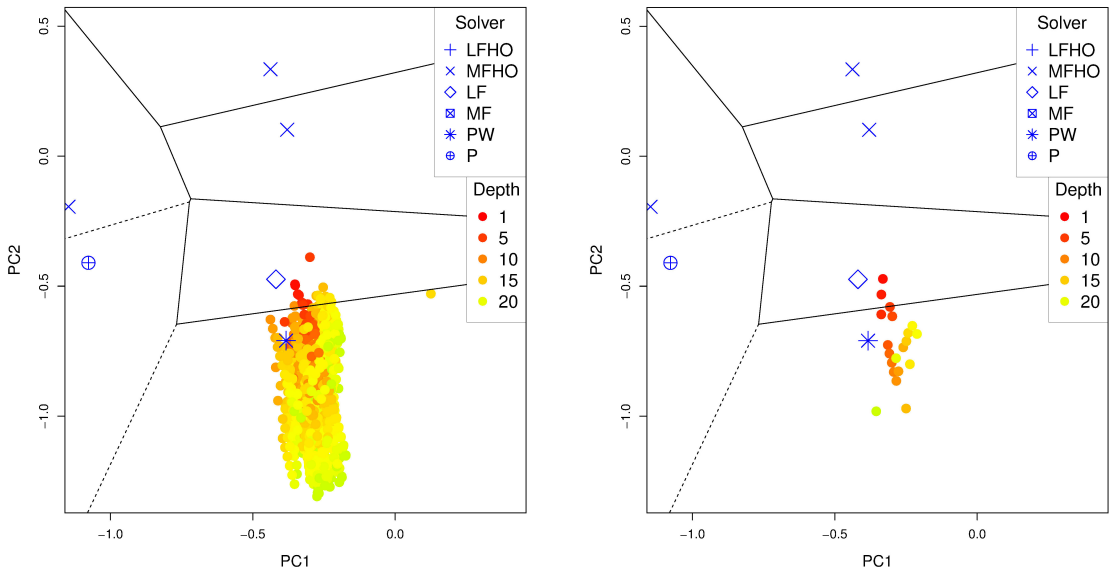
Thus, the performance can be further improved beyond what is possible when always sticking to the same heuristic. The question therefore now, becomes, if we can get improved performance just by switching between heuristics randomly, can we do even better if we do so intelligently?

6.1 CPLEX

In order to find out if the described DASH approach can improve a simple random switching heuristic, it is essential to set a few parameters of our solver and explore this parameter space, with the goal to determine an optimal assignment which maximises the performance. Particularly, the two main parameters specify until what depth (*maxDepth*) and with which frequency (*interval*) the heuristic switching will be enabled (for example, *maxDepth*=20 and *interval*=3 means that DASH will be active just for the nodes at the following depths: 1, 4, 7, 10, 13, 16, 19). Next, the dataset *extDatasetC*, that includes both the original training instances and the possible observed subinstances, is clustered. Using *g-means*, there are a total of 10 clusters formed, represented in Figure 6.1 projecting the feature space into two dimensions using Principal Component Analysis (PCA) [40]. Here, the cluster boundaries are represented by the solid lines, and the best heuristic for each cluster is identified by a unique symbol at its centre (see Section 5.3. These figures also show the typical way in which features change as the problem is solved with a particular heuristic. The nodes are coloured based on the depth of the tree, with (a) showing all the observed

Solver	**Par10**	**Avg**	**GeoMean**	**%Solved**
BSS	1321	315	54	93.8
RAND 1	4414	590	139	77.0
RAND 2	5137	609	135	72.7
VBS	326	225	42	99.4
VBS_Rand	217	217	41	100

Table 6.1: Solving times on the testing set using CPLEX with the branching heuristics implemented using the callback methodology.



(a) Multi-path evolution

(b) Single-path evolution

Figure 6.1: Position of a subinstance in the feature space based on depth using CPLEX. The solved instance is taken from the dataset *Airland*.

subinstances and (b) that of a single branch.

What this figure shows is that the features change gradually. This means that there is no need to check the features at every decision node. Therefore, the subinstance features are checked at every 3rd node. Similarly, the figure and those like it, show that using a depth of 10 is reasonable, as in most cases the nodes don't span more than two clusters.

GGA is the algorithm used to tune the parameters of DASH, computing the best heuristic for each cluster. The tuning algorithm runs the DASH solver several times, therefore each execution has a time-out value. Similarly to other genetic algorithms, GGA works with a population that evolves from one generation to another. When a new generation is reached, the algorithm returns the best results obtained in the previous one, that is a possible complete parameter set for the DASH solver. The time needed for the tuning phase is very sensitive to the time-out value, therefore the latter can't be too large. The chosen value for the CPLEX DASH solver is 300 seconds.

Table 6.2 presents the results, with the comparison to the best single solver (BSS), to the virtual best solver (VBS), and to a vanilla ISAC approach that, for a given instance, chooses the single best heuristic and then does not allow any switching. The comparison is performed using the arithmetic mean, the shifted geometric mean, and the Par10 average value, i.e., an arithmetic mean where the timed-out solving times are penalised as having taken 10 times the time-out value. Moreover, the percentage value of solved instances is also presented. From this data the observation is that DASH is able to perform much better than its more rigid counterpart. However, there is the possibility that switching heuristics might not be the best strategy for every instance. The proposed solution is called DASH+, which first clusters the original instances using ISAC and then allows each cluster to independently decide if it wants to use dynamic heuristic switching. The idea is to let ISAC decide for which instances DASH has to be executed. Ideally, it will choose the instances that will benefit from the proposed approach, giving an improvement that should be relevant on average. The effectiveness of this idea is proven by the numerical results shown in Table 6.2, in fact DASH+ offers a significant speed-up compared to DASH.

Taking a lesson from [56], which shows that often the features are not equally important, an additional idea to achieve better overall performance is to perform a feature selection operation. This thesis utilises the information gain filtering technique, often used in decision trees. In particular, this method is based on the calculation of entropy of the data as a whole and for each class. The feature filtering is applied to ISAC and DASH+. The two alternative techniques are referred to, respectively,

Solver	**Par10**	**Avg**	**GeoMean**	**%Solved**
BSS	1321	315	54.0	93.8
ISAC	1107	302	51.7	95.0
ISAC_filt	892	289	50.8	96.3
DASH	956	251	46.2	95.7
DASH+	858	255	45.6	96.3
DASH+filt	643	241	44.9	98.1
VBS	326	225	41.7	99.4
VBS_DASH	286	185	36.2	99.4

Table 6.2: Solving times on the testing set using the DASH solver on CPLEX.

as ISAC_filt and DASH+filt, and they both improve the results. In particular, the resulting solver DASH+filt performs considerably better than everything else.

Finally, Table 6.2 shows the performance of a virtual best solver if allowed to use DASH (VBS_DASH). Even though the current implementation cannot overtake VBS, future refinements to the portfolio techniques will be able to achieve performances much better than techniques that rely purely on sticking to a single heuristic.

6.2 SCIP

The positive results obtained using the MIP-solver CPLEX motivate the implementation of DASH as a SCIP plugin, aiming to give further proofs about the effectiveness of DASH. The idea is to show that the improvement is implementation-independent and that it is still relevant with a different set of branching rules. In particular, SCIP offers the opportunity to work directly on the native heuristics, giving the opportunity to show the improvement on a state-of-the-art solver, using its default version without any significant change.

According to Table 6.3 and Table 6.4 there is a very small gap between virtual best solver (VBS) and best single solver (BSS), i.e., respectively what is achievable switching among single branching heuristics and what is obtained using the best single branching heuristic. In this case, the reason relies on a branching heuristic, reliability branching on pseudo cost values, which has results that considerably outperform the others. It is then much more difficult to improve this BSS and prove the effectiveness of DASH, since it is likely to give better improvement when distinct clusters prefer

Solver	**Par10**	**Avg**	**GeoMean**	**%Solved**
BSS	1296	187	32	93.1
RANDvar	5333	603	106	70.8
RANDheur	3791	480	76	79.6
VBS	986.6	158.9	25.9	94.9
VBS_Rand	986.5	158.7	25.8	94.9

Table 6.3: Solving times on the testing set using SCIP in the *allBranch* case (using all the native branching rules).

Solver	Par10	Avg	GeoMean	%Solved
BSS	3055	392	67.6	83.6
VBS	1596	265	40.4	91.7

Table 6.4: Solving times on the testing set using SCIP in the *no2BSS* case (without using the random heuristics and the two solvers that perform best on the training set).

different heuristics. With the values shown in Table 6.3, it is probable that almost all the clusters prefer the BSS, that essentially means no switching at all.

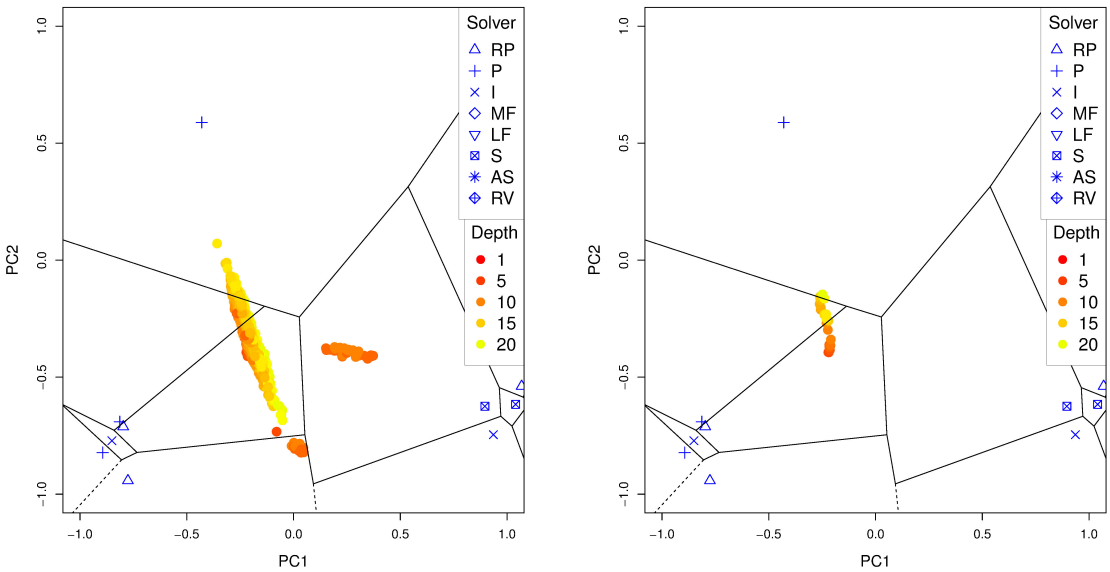
In order to provide more results that can prove the strength of the new approach introduced in this thesis, the analysis for the SCIP cases is divided in 2 parts:

- *allBranch*: it uses a portfolio that contains all the native branching rules of SCIP. In this case the BSS and VBS have similar performance (see Figure 6.3);

- *no2BSS*: it uses all the branching rules, except the random one and the two that perform best on the training set, i.e., respectively Reliability Branching (RP) and Pseudo Cost Branching (P). In the remaining portfolio, the gap between BSS and VBS is much larger than in the *allBranch* case. The employed heuristics are then Inference History Branching (I), Most Fractional Rounding (MF), Least Fractional Rounding (LF), Full Strong Branching (S), Full Strong Branching on all Variable (AS), and Random Variable Branching (RV).

As already done while using CPLEX, a clustering operation is performed on the extended datasets, i.e., *extDatasetS* and *extDatasetSr*. The obtained clusterisations have, respectively, 14 and 12 clusters formed, as shown in Figure 5.2.

Figure 6.2 shows the evolution of the solving process using the version *allBranch* of DASH, representing each subinstance as a coloured point, and using a colour scheme that indicates the depth of the subinstance in the solving tree. The representation is a projection obtained using the principal component analysis (PCA) as a dimensionality reduction technique in a clustered space, where the solid lines represent the cluster boundaries and the blue symbols refers both to a cluster centre and to its assigned solving heuristic (described in Section 5.3.1). Similarly to Figure 6.1, Figure 6.2(a)



(a) Multi-path evolution

(b) Single-path evolution

Figure 6.2: Position of a subinstance in the feature space based on depth using SCIP, version based on *extDatasetS*. The solved instance is taken from the dataset *Miplib2010*: *bienst2*.

shows all the observed subinstances and Figure 6.2(b) shows a single branch. In both cases the selected depth limit is 20. The solving process gradually moves the instance to other clusters, in particular from a cluster assigned to Reliability Branching (RP) to ones assigned to Full Strong Branching (S) and Pseudo Cost Branching Product Score (P). Another observation is that the direction of the spatial evolution is not unique, but the single branch is likely to continue in a single direction. It is also important to underline the presence of groups of subproblems that are in a different position (each group of a same branch), probably because of the application of preprocessing, backtrack, or restart techniques. Since their depth is between 4 and 10 and the solution has been found deeper in the branch-and-bound tree, those branches have been pruned.

In the CPLEX case, the max depth value and the depth interval of application (*maxDepth* and *interval*) have been set arbitrarily. In this case these two parameters are included among the configuration values determined by GGA during the tuning

Solver	Par10	Avg	GeoMean	%Solved
BSS	1242.8	178.6	30.2	93.1
ISAC	1242.8	178.6	30.2	93.1
ISAC_filt	1242.8	178.6	30.2	93.1
DASH	1142.1	182.1	31.4	94.1
DASH+	1241.8	177.6	30.1	93.4
DASH+filt	1373.3	190.8	30.7	92.7
VBS	986.6	158.2	25.9	94.9
VBS_DASH	862.8	153.3	25.0	95.6

Table 6.5: Solving times on the testing set using the DASH solver on SCIP using the portfolio *allBranch*.

operation, with the drawback of increasing the search space, and then to make the tuning process longer. However, these two parameters are strongly related to the performance, it could be therefore important to explore reasonable values in order to find a better solution.

The time-out values assigned to the tuning algorithm are 300, 600, and 900 seconds, however the presented data are related to the 600 seconds version, since it returned the optimal solutions in both cases. The genetic tuning algorithm GGA returns an assignment of parameters for each of the obtained generations, i.e., branching rules assigned to the clusterisation, *maxDepth* and *interval* control the activation of the DASH algorithm. In particular, the best DASH solver obtained for the portfolio *allBranch* uses $maxDepth = 14$ and $interval = 7$, while for the portfolio *no2BSS* the obtained best solver has $maxDepth = 12$ and $interval = 6$. Both these settings enables an heuristic switch just at the root node and at other two depths, differently from the configuration used for CPLEX, which enables DASH more frequently. The result obtained in this Section confirms that the movement of an instance during the solving process is gradual, therefore a more frequent application of DASH could increase the overhead without offering a significant speed-up.

Similarly to what was already done for CPLEX, Table 6.5 and Table 6.6 presents the results, with the comparison BSS, VBS, and ISAC, respectively using the portfolio of heuristics *allBranch* and *no2BSS*. Among the presented data, VBS represents

Solver	Par10	Avg	GeoMean	%Solved
BSS	3119.2	399.5	68.4	83.6
ISAC	2558.2	339.1	49.7	86.3
ISAC_filt	2426.1	318.3	46.7	87.0
DASH	2415.2	309.4	45.4	87.0
DASH+	2171.4	306.7	45.2	88.5
DASH+filt	2164.7	300.0	44.2	88.5
VBS	1596.3	265.4	40.4	91.7
VBS_DASH	1540.2	258.2	38.9	92.1

Table 6.6: Solving times on the testing set using the DASH solver on SCIP using the portfolio *no2BSS*.

a lower bound that ideally can be improved using new approaches, VBS_DASH represents the lower bound using DASH, and BSS and ISAC are the solving times to improve. According to the reported values, using the portfolio *allBranch* DASH does improve the results in terms of percentage of instances solved, therefore in terms of Par10 average value. This is reasonable, since there is not a significant gap between BSS and VBS. Using instead the portfolio *no2BSS* there is a relevant improvement regarding both the number of instances solved and in terms of solving times. These results are confirmed by Figure 6.3 and Figure 6.4, where the evolution of performance while tuning is represented, executing the different tuned version of DASH on the testing set, compared to the best single solver (BSS) and to the virtual best solver (VBS) solving times. In particular, the two horizontal lines, green and blue, represent respectively the BSS and the VBS average solving time. As already shown by Table 6.2, Table 6.5, and Table 6.6, while using the portfolio *allBranch* the improvement regards mainly the number of solved instances, therefore the effect is captured by the Par10 average value, while using the portfolio *no2BSS* there is also a speed-up in terms of geometric mean. Furthermore, Figure 6.3 and Figure 6.4 show that, using 600 seconds of time-out, the tuning operation does reach its lower-bound before the 20th/25th generation.

6.3 Chapter Summary

This chapter showed the possible enhancements that can be achieved by employing DASH on MIP problems. An improved version of DASH is also introduced, that uses a vanilla ISAC approach (Instance-Specific Algorithm Configuration) with the aim of applying the proposed approach just on the instances that will benefit from it. The experiments were done using three different portfolios of branching rules and using the two state-of-the-art MIP-solvers CPLEX and SCIP. In all cases, solvers trained using the DASH or DASH+ approaches outperformed both the algorithm in the portfolio and ISAC. It is then shown that, having just a few solvers that outperform the others on the whole dataset, there will be a less diverse assignment of heuristics to the clusterisation, therefore low chance of switching and of having a speed-up. On the contrary, when the portfolio has many solvers with performances similar in their magnitude, then the tuning will return a diverse assignment of heuristics and the switch will be possible. In this case, the average solving time can be significantly improved.

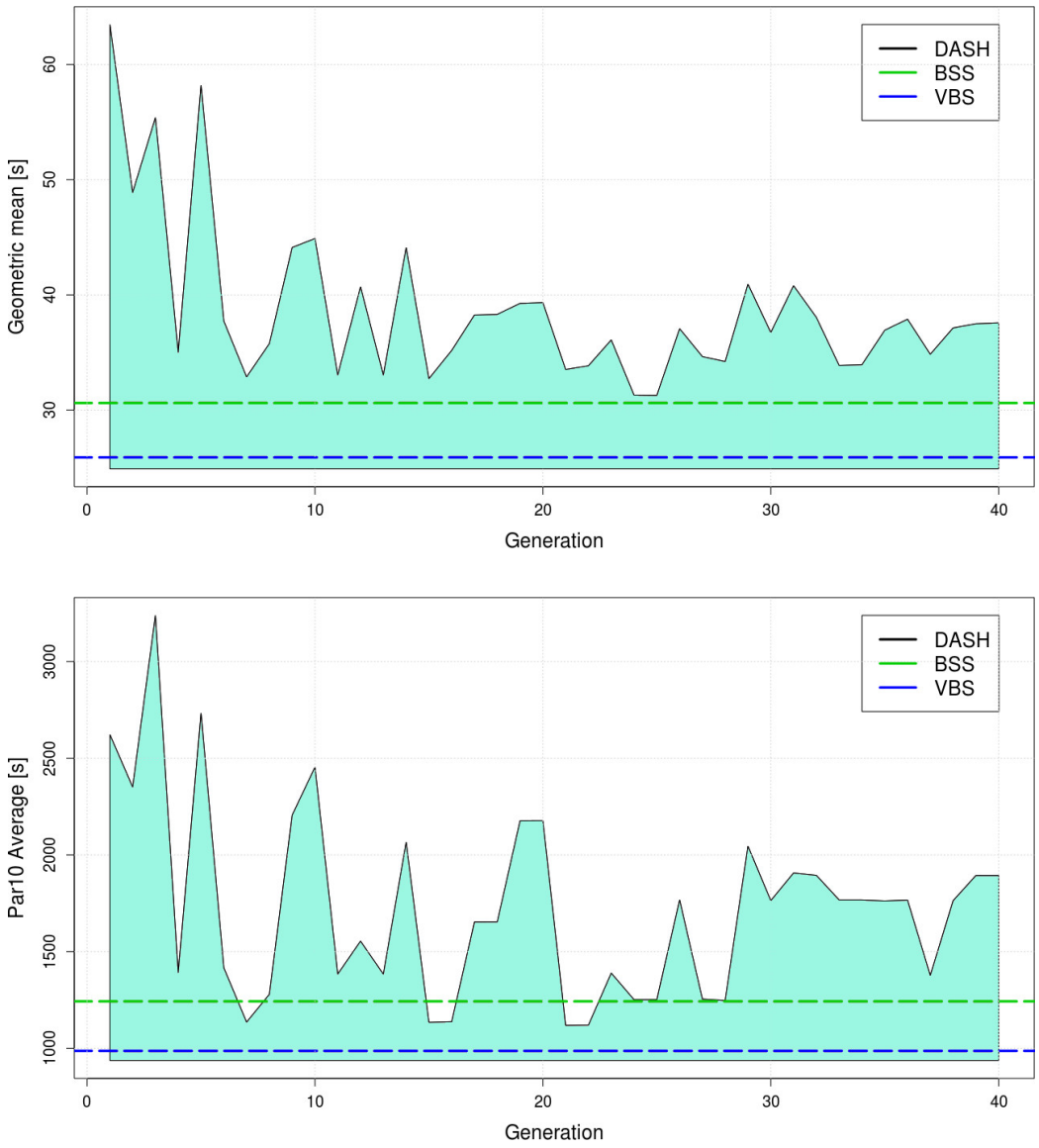


Figure 6.3: Evolution of the solving time of DASH while the tuning operation evolves using the portfolio *allBranch*. The solving time is expressed both as geometric mean and Par10 average.

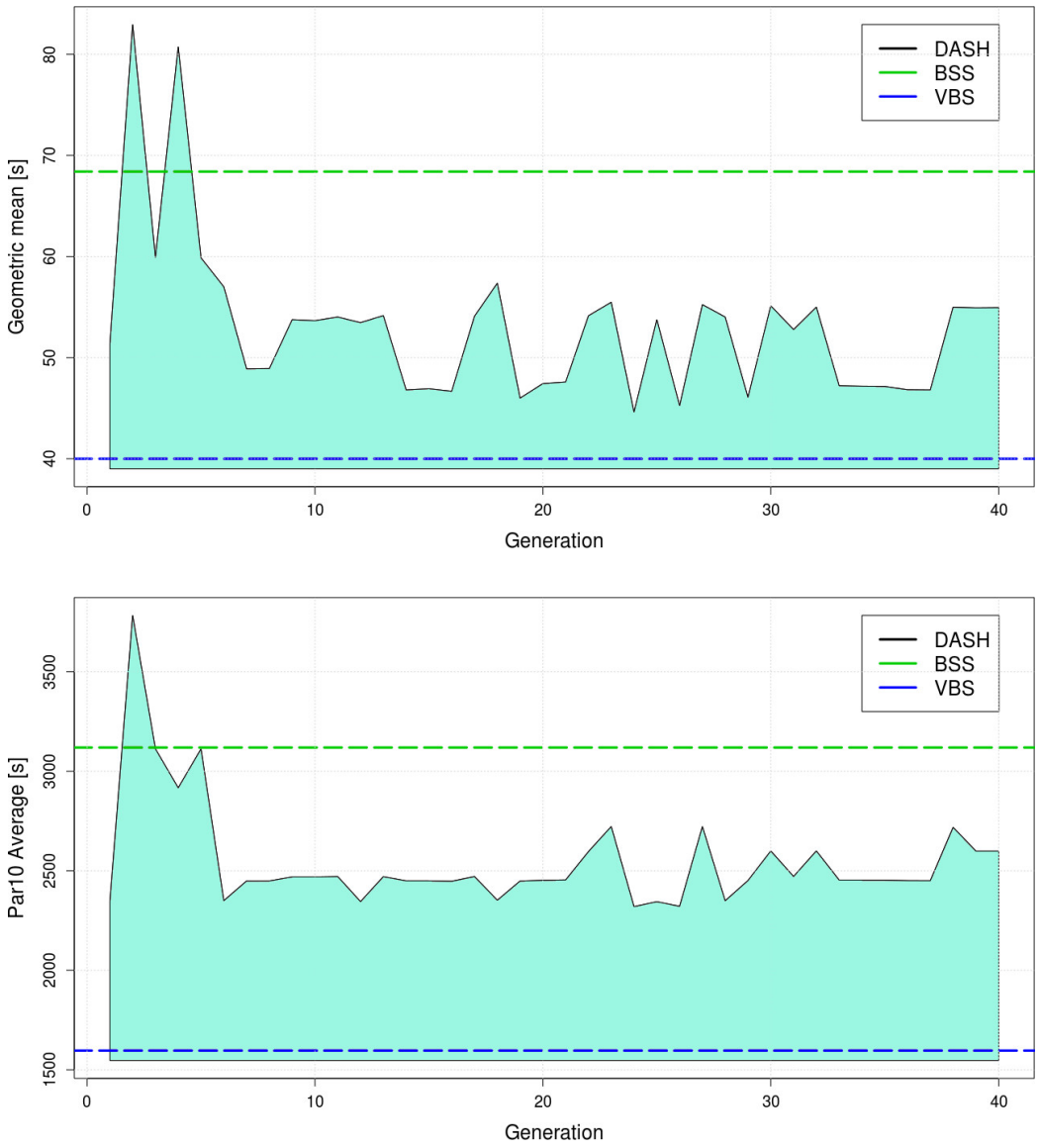


Figure 6.4: Evolution of the solving time of DASH while the tuning operation evolves using the portfolio *no2BSS*. The solving time is expressed both as geometric mean and Par10 average.

Chapter 7

Conclusion

This thesis introduces a Dynamic Approach for Switching Heuristics (DASH). Using MIP as the running example, it is shown how to automatically determine when a subproblem observed during a branch-and-bound search is significantly different from what has been observed before, and therefore warrants a change of tactics used while solving it. Employing a diverse set of instances, the dissertation demonstrates that significant performance improvements are possible if a solver does not use the same guiding heuristic for the whole solving process.

This work expands the approach recently introduced in [16] from the set partitioning problem with problem dependent heuristics, to the much more general problem of MIP. As already done in [16], the introduced methodology is based on ISAC, Instance-Specific Algorithm Configuration [11], a high performance algorithm selection method.

The dissertation gives definitions and a brief introduction into MIP and branch-and-bound (B&B). Furthermore, the several heuristics employed are introduced in the second chapter and described with the implementation details in the fourth chapter. Moreover, several related works are presented, first about algorithm selection and then about unsupervised learning.

The DASH methodology realisation is motivated by a phase of data analysis, mainly exposed in chapter four. The first task was to identify a set of features that best describes the structure of a MIP problem. In particular, in the chosen feature space it is possible to distinguish between distinct instances, and also between two subinstances of a same original problem. Furthermore, the features can be easily computed, since they are based on structural information that doesn't require a relevant

computational effort. This is a very important aspect since the overhead introduced by DASH depends mainly on the features' computation. It is then showed that there is a relation between the feature space and the preferred branching heuristic. This relation has been determined using the clustering algorithm *g-means* on large and heterogeneous datasets. These datasets are composed by instances of many different MIP datasets, together with a subsample of their subinstances obtained using each one of the available heuristics. Since there are three employed algorithm portfolios, one using the MIP solver CPLEX and the other two using SCIP, there are three datasets.

This approach has been shown to be beneficial especially for two of the three employed datasets. In particular, DASH can offer a significant speed-up to the solving process when there is a large gap between the Best Single Solver (BSS) and the Virtual Best Solver (VBS), i.e., respectively the solver based on single branching heuristics, whose average running time is the lowest on all the dataset, and the lower bound of what is achievable with a perfect portfolio that for every instance always chooses the one that results in the best performance. In both situations, DASH gives a highly relevant speed-up both compared to the BSS and to ISAC. The results obtained using DASH+ and DASH+filt shows a further improvement that brings the performance nearer to that of the VBS. The third dataset, instead, has a small gap between BSS and VBS. Since this gap is an indicative value of how much it is possible to improve, even assuming that DASH does not make mistakes, the possible speed-up is even smaller because of the overhead. However, the outcome is a further proof of the effectiveness of DASH. In fact, the performances are very similar to the BSS but with a larger number of solved instances, that is very close to what is achievable with VBS.

In its entirety, this thesis showed that DASH is an effective methodology, demonstrating that it is possible to train a model that dynamically adapts to the structural changes of an instance during the solving process, switching the applied heuristic when needed. Furthermore, this approach is further improved through the combination of ISAC and DASH (DASH+), and finally the application of feature filtering techniques (DASH+filt). Finally, the methodology has also been realised as a SCIP plugin, with the goal of making it easily available to the community.

This thesis introduces a new methodology and it shows its effectiveness for MIP problems. Indeed, there are many remaining opportunities for future work. First, among all the available heuristics, this dissertation works just with the branching heuristic, that is directly related to the branch-and-bound process. However, there could be other heuristics suitable for DASH. Therefore, the portfolio of solvers could be expanded using combinations of different heuristics, with the risk, however, of increasing the search space of the tuning algorithm GGA too much, obtaining much longer times for the tuning.

A second direction for further research could be extending the approach to other problem spaces, with the final goal of obtaining an even more general methodology. This probably requires much more effort, since changing the problem space means collecting instances for a new dataset and to perform analysis on its structure in order to motivate further studies. Furthermore, the coding work could be much greater than for the previous proposal, since changing the problem space could mean having to also change the general solver (recall that this thesis relies on results obtained with the MIP-solvers CPLEX and SCIP). DASH is here described as strictly related to the branch-and-bound (B&B) methodology, so it could be a good idea to try extending it to problem domains that can be solved using the same technique (like non-linear programming and Max-SAT). However, it could be possible to apply the same idea in other ways.

Further research could also be done for the instance-oblivious tuning. In particular, the time-out value set for the tuning operation using GGA has always been smaller than the time-out value used for the tests, mainly because the tuning operation requires huge amount of CPU time. It is therefore important to investigate new techniques that can find usable parameterisations within a reasonable timeframe. Another future task would be the automation of the whole offline training process. At the moment, gathering the solving times, the clustering, and the tuning operations are automated, but still independent parts of DASH. In order to make this new appoach easy to use for a normal user, it would be important to automate the whole process and to group it in a single command.

In summary, this dissertation improves the currently available algorithm selection

methodology, applied to the MIP problem space, and hopefully further research will enhance and expand its applicability.

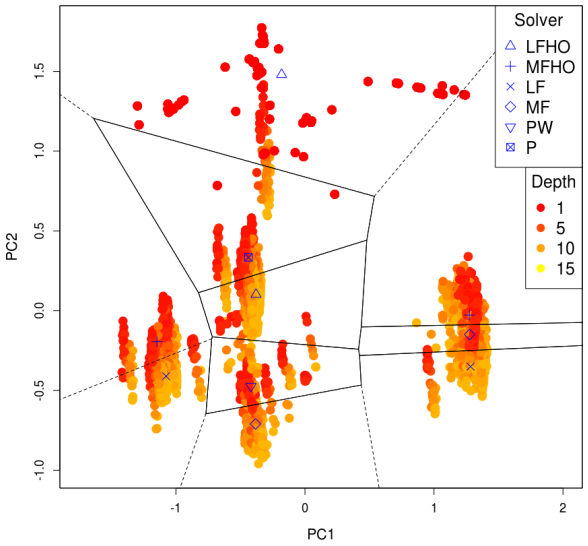
Appendices

Appendix A

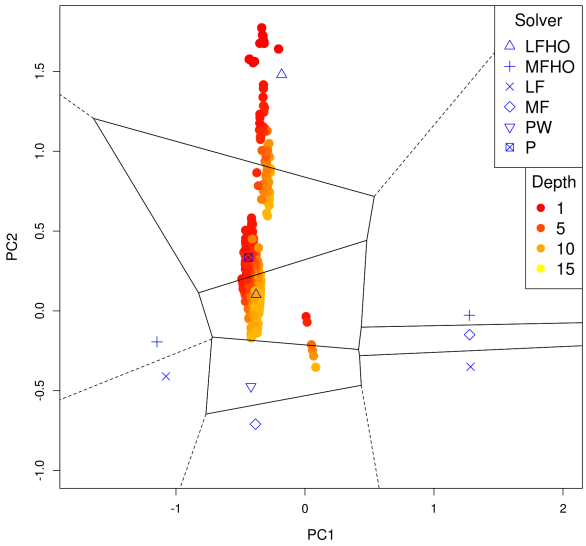
Feature space analysis

The presented figures are bidimensional projections of the feature space, obtained using Principal Component Analysis (PCA) [40]. In particular, for each of the employed portfolio of solvers an extended dataset has been collected. As presented in Section 4.2, the available extended datasets are *extDatasetC*, *extDatasetS*, and *extDatasetSr*. For each of these datasets, the distribution of instances and subinstances that "starts from the same cluster" is shown in the following figures. A set of subinstances satisfies this condition if the root nodes of their elements belong to the same cluster. For example, Figure A.1(a) shows all the instances in *extDatasetC*, while Figure A.1(b) represents all the instances that have their root node in cluster 1 (for the clusterisation indexes see Figure 5.2).

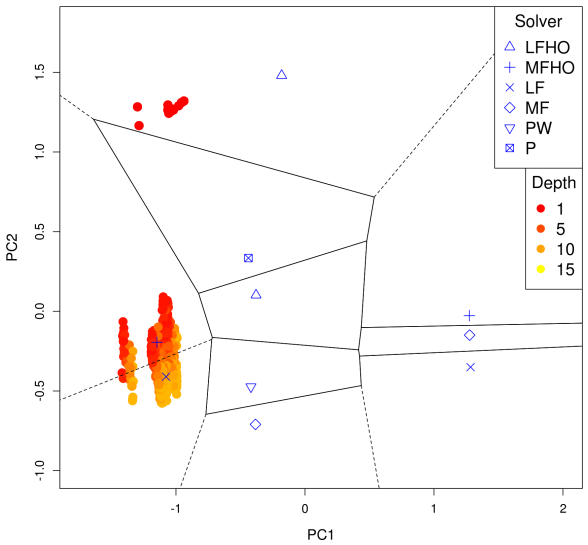
These figures show that in almost every case there is a relevant number of subinstances which belong to a cluster different from their original one (i.e., the cluster which contains their original instance, the root node of the branch-and-bound solving process), giving further proof of the high frequency of this event. Therefore, the idea of switching heuristics has reasons for being applied several times during the solving process, giving the opportunity of using the optimal heuristic for a specific subinstance, with the goal of obtaining a significant speed-up.



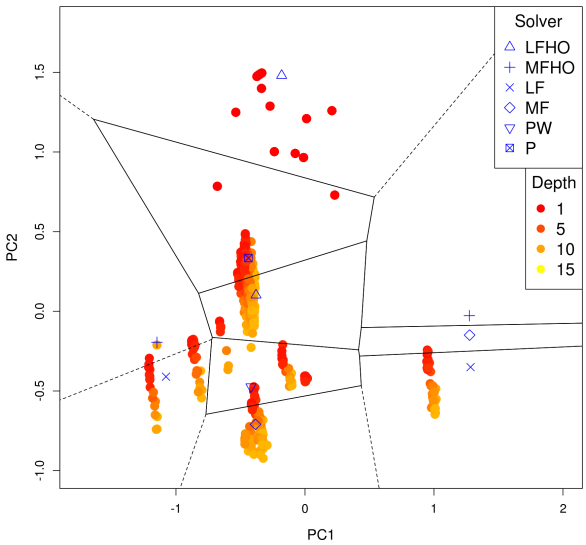
(a) Complete dataset



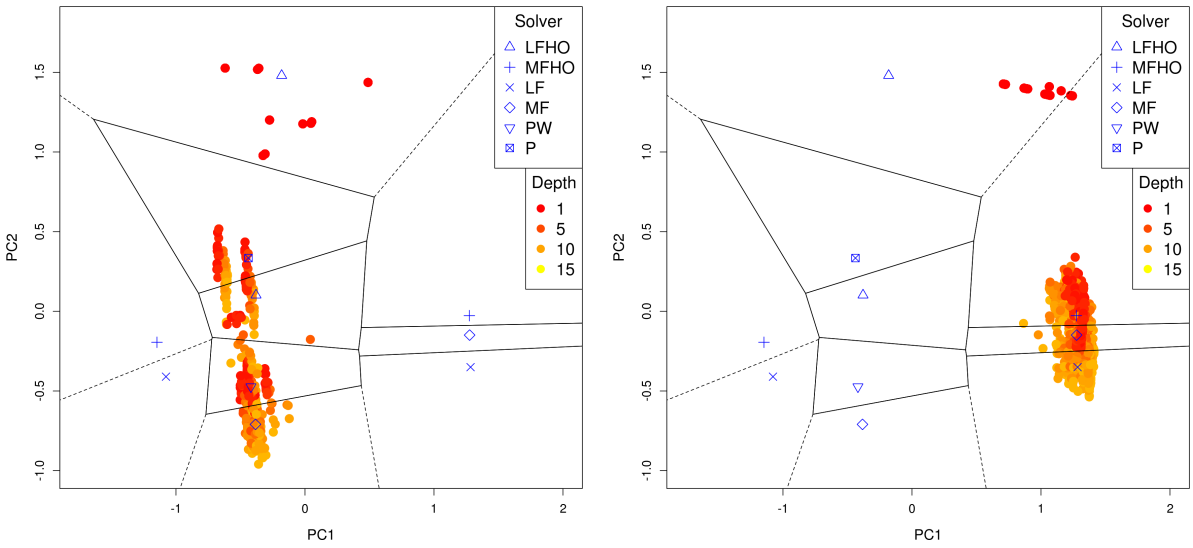
(b) Root cluster: 1



(c) Root cluster: 2



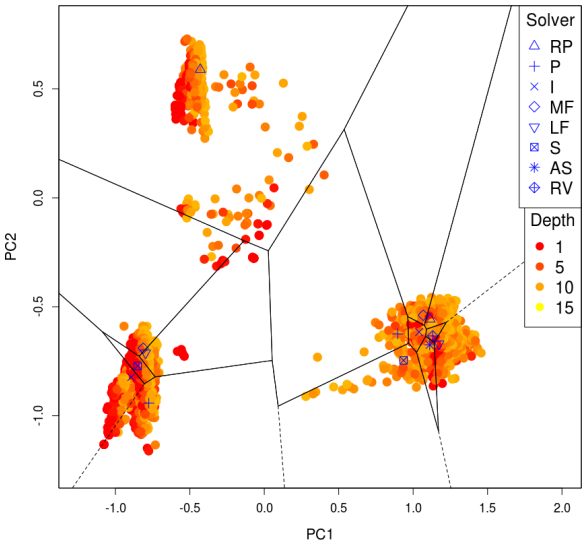
(d) Root cluster: 5



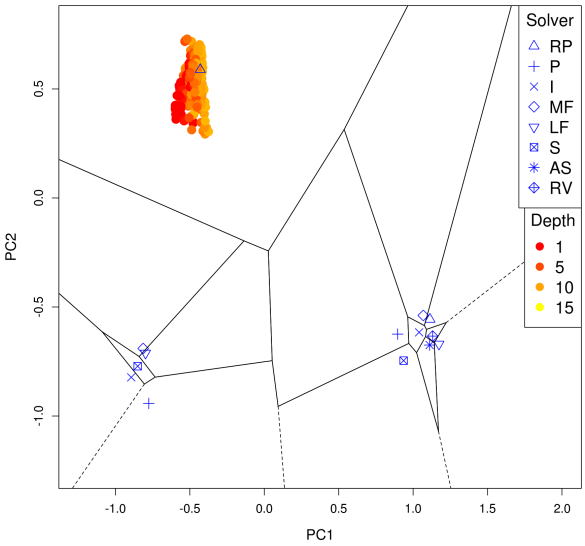
(e) Root cluster: 6

(f) Root cluster: 8

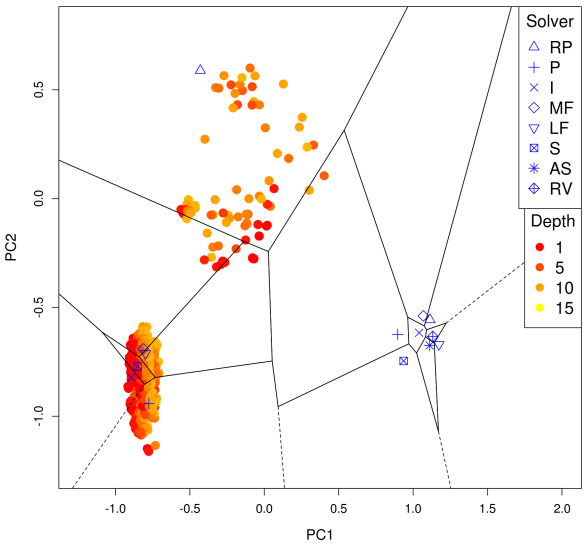
Figure A.1: *extDatasetC* distribution in the clustered feature space. The representation is a PCA bidimensional projection. Figure (a) represent the whole dataset, the others the instances whose root node is in a specific cluster.



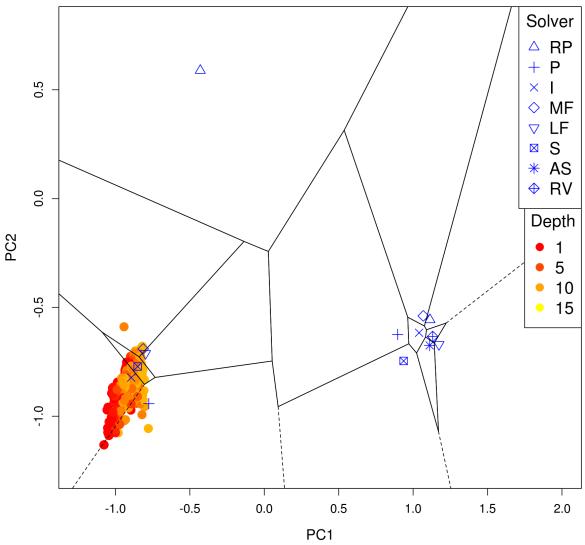
(a) Complete dataset



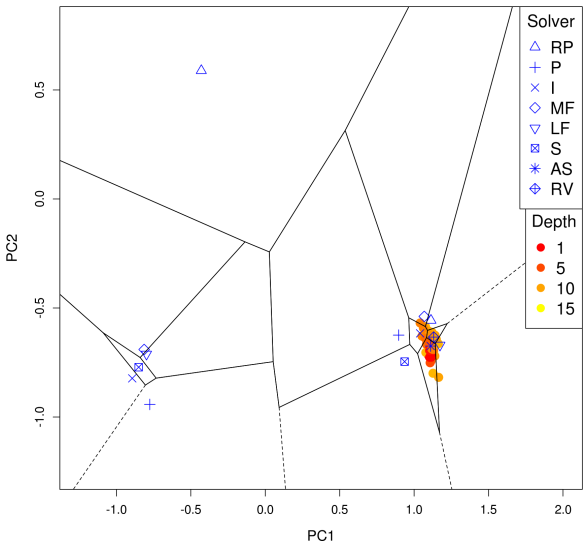
(b) Root cluster: 1



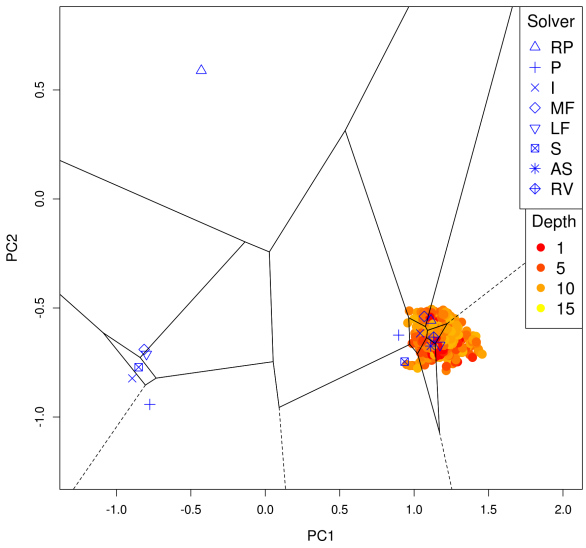
(c) Root cluster: 3



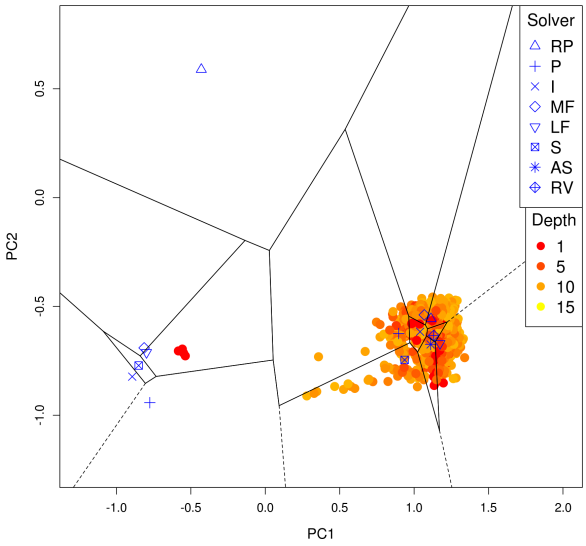
(d) Root cluster: 6



(e) Root cluster: 8

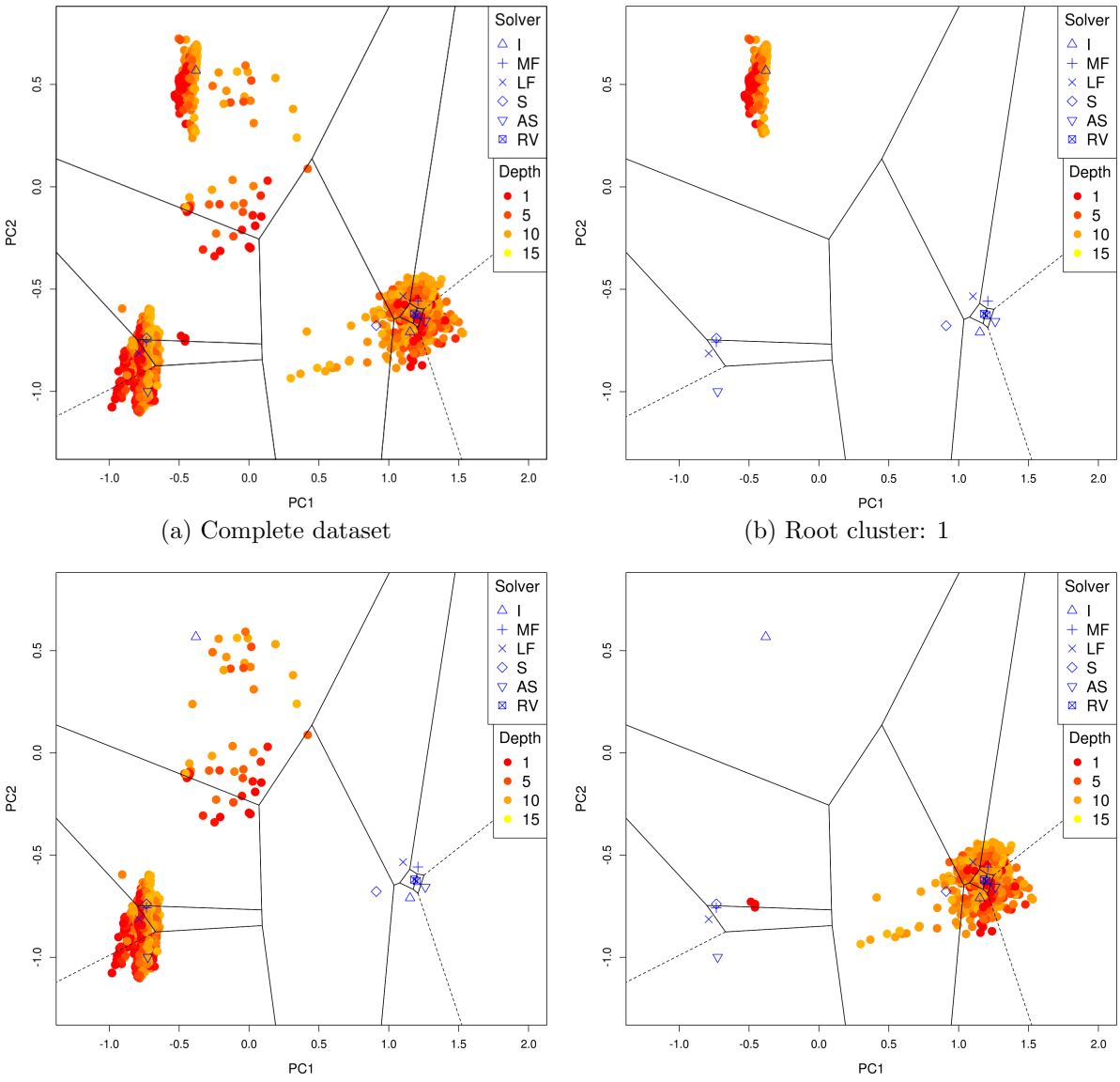


(f) Root cluster: 9



(g) Root cluster: 11

Figure A.2: *extDatasetS* distribution in the clustered feature space. The representation is a PCA bidimensional projection. Figure (a) represent the whole dataset, the others the instances whose root node is in a specific cluster.



(a) Complete dataset

(b) Root cluster: 1

(c) Root cluster: 3

(d) Root cluster: 12

Figure A.3: *extDatasetSr* distribution in the clustered feature space. The representation is a PCA bidimensional projection. Figure (a) represent the whole dataset, the others the instances whose root node is in a specific cluster.

Bibliography

[1] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm selection and scheduling, CP (2011) 454–469.

[2] L. A. Wolsey, Y. Pochet, Production Planning by Mixed Integer Programming, Springer, Berlin, 2006.

[3] A. Balakrishnan, T. Magnanti, P. Mirchandani, Annotated Bibliographies in Combinatorial Optimization, Wiley, New York, 1997, Ch. Network Design, pp. 311–334.

[4] E. Zurel, N. Nisan, An efficient approximate allocation algorithm for combinatorial auctions, Proceedings of the 3rd ACM conference on Electronic Commerce (2001) 125–136.

[5] A. Caprara, M. Fischetti, P. Toth, Modeling and solving the train timetabling problem., Operations Research 50 (5) (2002) 851–861.

[6] X. Cai, C. J. Goh, A fast heuristic for the train scheduling problem., Computers & OR 21 (5) (1994) 499–510.

[7] T. Achterberg, T. Koch, A. Martin, Branching rules revisited, Operations Research Letters 33 (2004) 42–54.

[8] J. T. Linderoth, M. W. P. Savelsbergh, A computational study of search strategies for mixed integer programming, INFORMS Journal on Computing 11 (1997) 173–187.

[9] T. Achterberg, Constraint Integer Programming, Ph.D. thesis, Technische Universität Berlin (2007).

[10] IBM, *IBM CPLEX* v12.5, `http://www14.software.ibm.com/webapp/download/preconfig.jsp?id=2010-07-23+03%3A35%3A42.559321R&S_TACT=1&S_CMP=` (2013).

[11] S. Kadioglu, Y. Malitsky, M. Sellmann, K. Tierney, Isac –instance-specific algorithm configuration, ECAI (2010) 751–756.

[12] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, B. O'Sullivan, Using case-based reasoning in an algorithm portfolio for constraint solving, AICS.

[13] L. Xu, F. Hutter, J. Shen, H. H. Hoos, K. Leyton-Brown, Satzilla2012: Improved algorithm selection based on cost-sensitive classification models, sAT Competition (2012).

[14] J. R. Rice, The algorithm selection problem, Advances in Computers 15 (1976) 65–118.

[15] C. P. Gomes, B. Selman, Algorithm portfolios, Artif. Intell. 126 (1-2) (2001) 43–62.

[16] S. Kadioglu, Y. Malitsky, M. Sellmann, Non-model-based search guidance for set partitioning problems, AAAI.

[17] A. Schrijver, Theory of linear and integer programming, John Wiley & Sons, Inc., New York, NY, USA, 1986.

[18] G. L. Nemhauser, L. A. Wolsey, Integer and combinatorial optimization, Wiley-Interscience, New York, NY, USA, 1988.

[19] L. Wolsey, Integer Programming, Wiley Series in Discrete Mathematics and Optimization, Wiley, 1998.

[20] M. Collautti, Y. Malitsky, D. Mehta, B. O'Sullivan, Snnap: Solver-based nearest neighbor for algorithm portfolios, ECML/PKDD (2013) 435–450.

[21] S. P. Lloyd, Least squares quantization in pcm, IEEE Transactions on Information Theory 28 (2) (1982) 129–136.

[22] G. Hamerly, C. Elkan, Learning the k in k-means, NIPS.

[23] T. W. Anderson, D. A. Darling, Asymptotic theory of certain goodness of fit criteria based on stochastic processes, Annals of Mathematical Statistics 23 (1952) 193–212.

[24] Fselector r package, `http://cran.r-project.org/web/packages/FSelector/index.html`.

[25] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 67–82.

[26] K. Leyton-Brown, E. Nudelman, G. Andrew, J. Mcfadden, Y. Shoham, A portfolio approach to algorithm selection, IJCAI (2003) 1542–1543.

[27] L.-B. Kevin, N. Eugene, A. Galen, M. Jim, S. Yoav, Boosting as a metaphor for algorithm design, in: R. Francesca (Ed.), Principles and Practice of Constraint Programming CP 2003, Vol. 2833 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 899–903.

[28] Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Parallel sat solver selection and scheduling, in: CP, 2012, pp. 512–526.

[29] M. J. Streeter, S. F. Smith, New techniques for algorithm portfolio design, Robotics 10 (2008) 519–527.

[30] L. Xu, F. Hutter, H. H. Hoos, K. Leyton-Brown, Satzilla: Portfolio-based algorithm selection for sat, J. Artif. Intell. Res. (JAIR) 32 (2008) 565–606.

[31] Sat competitions, `http://www.satcompetition.org/`.

[32] R. Riesbeck, C. Schank, Inside Case-Based Reasoning, Lawrence Erlbaum, 1989.

[33] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, CP (2009) 142–157.

[34] M. Yuri, M. Deepak, O. Barry, Evolving instance specific algorithm configuration., in: M. Helmert, G. Rger (Eds.), SOCS, AAAI Press, 2013.

[35] K. M. Ting, An instance-weighting method to induce cost-sensitive trees, IEEE Trans. on Knowl. and Data Eng. 14 (3) (2002) 659–665.

[36] E. Nudelman, K. Leyton-Brown, H. H. A. Devkar, Y. Shoham, Understanding random sat: Beyond the clauses-to-variables ratio, in: Proceedings of CP, 2004.

[37] E. Nudelman, K. L. Brown, H. H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: Beyond the Clauses-to-Variables ratio, in: M. Wallace (Ed.), Principles and Practice of Constraint Programming - CP, 10th International Conference, Toronto, Canada, Vol. 3258 of Lecture Notes in Computer Science, Springer, 2004, pp. 438–452.

[38] D. H. Leventhal, M. Sellmann, The accuracy of search heuristics: An empirical study on knapsack problems., in: P. Laurent, T. M. A. (Eds.), CPAIOR, Vol. 5015 of Lecture Notes in Computer Science, Springer, 2008, pp. 142–157.

[39] H. Samulowitz, R. Memisevic, Learning to solve qbf, in: AAAI, 2007, pp. 255–260.

[40] H. Abdi, L. J. Williams, Principal component analysis (2010).

[41] *SCIP* version 3.0.1, `http://scip.zib.de/doc/html_devel/index.shtml` (2013).

[42] I. COIN-OR Foundation, *CBC*: Coin-or branch and cut, `https://projects.coin-or.org/Cbc`.

[43] F. S. Foundation, *GLPK*: Gnu linear programming kit, `http://www.gnu.org/software/glpk/glpk.html`.

[44] *MINTO*: Mixed integer optimizer, `http://coral.ie.lehigh.edu/~minto/index.html`.

[45] C. L. at Lehigh University, *SYMPHONY*, `https://projects.coin-or.org/SYMPHONY/`.

[46] F. F. I. Corporation), *XPRESS* solver engine, `http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx`.

[47] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, K. Wolter, MIPLIB 2010 - Mixed Integer Programming Library version 5, Mathematical Programming Computation 3 (2011) 103–163.

[48] A. Atamturk, Flow pack facets of the single node fixed-charge flow polytope, Operations Research Letters 29 (2001) 107–114.

[49] A. Atamturk, J. C. Munoz, A study of the lot-sizing polytope, Mathematical Programming 99 (2004) 443–465.

[50] A. Atamturk, On the facets of the mixed–integer knapsack polyhedron, Mathematical Programming 98 (2003) 145–175.

[51] A. Atamturk, G. L. Nemhauser, M. W. P. Savelsbergh, Valid inequalities for problems with additive variable upper bounds, Mathematical Programming 91 (2001) 145–162.

[52] K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, ACM Conference on Electronic Commerce (EC-00).

[53] A. Saxena, Mip benchmark instances, `http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm` (2010).

[54] T. Mori, M. Kikuchi, K. Yoshida, Term weighting method based on information gain ratio for summarizing documents retrieved by ir systems, in: Journal of Natural Language Processing, 9(4):3–32, 2001.

[55] T. Mori, Information gain ratio as term weight: the case of summarization of ir results, in: Proceedings of the 19th international conference on Computational linguistics - Volume 1, COLING '02, Association for Computational Linguistics, Stroudsburg, PA, USA, 2002, pp. 1–7.

[56] C. Kroer, Y. Malitsky, Feature filtering for instance-specific algorithm configuration, ICTAI (2011) 849–855.