Albert-Ludwigs University of Freiburg

Master Thesis

PER INSTANCE ALGORITHM CONFIGURATION

Author: André Biedenkapp | Supervisor: Dr. Marius Lindauer
First examiner: Prof. Frank Hutter | Second examiner: Prof. Dr. Bernhard Nebel

A thesis submitted in fulfillment of the requirements for the degree of Master of Science
in the Research Group Machine Learning,
Faculty of Engineering, Department of Computer Science

Submitted on 2nd August 2017

DECLARATION

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg,	
Place, Date	Signature

I want to thank my friends and family who always supported me in my endeavor to get my Master's degree, as well as my proofreaders for reading every word at least twice. I want to thank my supervisor for getting me interested in the topic and keeping me motivated and I want to thank my lovely girlfriend for keeping up with my antics during these short six months.

Abstract

It is often necessary to adjust the parameters of an algorithm for a given problem domain in order to achieve peak performance. This is the case for many solvers in a multitude of AI problems such as mixed integer programming (MIP), propositional satisfiability (SAT) solving and AI-planning. General purpose algorithm configuration procedures have been developed to find performance-optimizing configurations. While impressive performance gains can be achieved using such procedures, they are limited to finding a single configuration over a set of instances. If this set consists of dissimilar instances, a single configuration might not suffice to solve the instance set effectively. Here we present a novel *per instance algorithm configuration* tool that is able to find multiple configurations and learn to select which configuration to use for new instances. This system is able to identify homogeneous subsets of instances and iteratively refine these. In our experiments we demonstrate the systems capabilities and show that it is able to reach similar and even better performance as state-of-the-art systems.

Zusammenfassung

Damit ein Algorithmus seine Höchstleistung erzielen kann, ist es oft von Nöten seine Parameter an die Problem Domäne anzupassen. Dies ist der Fall für eine Reihe verschiedener Algorithmen aus unterschiedlichsten Problembereichen der Künstlichen Intelligenz. Beispiele hierfür sind Algorithmen zur Lösung von Erfüllbarkeitsproblemen oder gemischt-ganzzahliger Optimierungsproblemen. Um einen Anwender vor der mühsamen Aufgabe der manuellen Einstellung der Parameter zu wahren, wurden Konfigurationsverfahren entwickelt, die dazu in der Lage sind eindrucksvolle Ergebnisse zu erzielen. Solche Verfahren sind darin limitiert, dass sie nur eine einzige performante Konfiguration finden können. Auf Instanz Mengen welche aus stark unterschiedlichen Instanzen aufgebaut sind können diese Verfahren keine Konfiguration finden die alle Instanzen gut lösen kann. In dieser Arbeit stellen wir ein neues *per instance algorithm configuration* System vor, welches dazu in der Lage ist mehrere Konfigurationen zu finden und zu lernen welche Instanz von diesen gut gelöst werden. Dieses System ermittelt homogene Teilmengen und verfeinert diese iterativ. In unseren Experimenten zeigen wir das dieses System ähnliche und sogar bessere Ergebnisse erzielen kann, als die zurzeit besten Verfahren.

Contents

Chapter 1

Introduction

Hard combinatorial problems, such as *satisfiability (SAT) problems*, *AI planning* or *Mixed Integer Programming (MIP)* pose developers with the challenge to develop algorithms that are capable of efficiently solving a given problem instance.

In an industry setting such a cost metric could be *required running time*. For example, a company that is able to solve more problems than competitors in a specified amount of time is clearly more attractive to customers who need a lot of problems solved.

On the other hand, if running time is not of importance, a more suitable cost metric could be solution quality. In a planning task this could mean that a user is not only interested in finding a path from start to finish but that the resulting plan is as short as possible.

Modern algorithms are required to be applicable to different domains. For example, a problem instance could be in the domain of finding a holiday location. In such a scenario there would be few variables that have to adhere to many constraints as opposed to an instance in the domain of loading a vehicle with wares. This would involve many variables that have to adhere to fewer constraints, such as overall weight and destination.

Nowadays it is custom to design highly parameterized algorithms in order to create a flexible solver that is capable of working on multiple different problem domains. This mindset is also expressed by the programming paradigm *Programming by Optimization* (Hoos, 2012), where developers are encouraged to expose possible parameters of their algorithms such that they avoid early commitment to only one possible path in the execution of the resulting algorithm and to seek alternative design choices. Thus developers will create a flexible end product that is capable of adapting to different problem tasks.

The number of parameters and their possible values however make it difficult even for human experts to determine well performing settings for a given problem task. To alleviate humans of the time consuming task of searching for well performing configurations, automatic configuration procedures such as *ParamILS* (Hutter et al., 2009), *SMAC* (Hutter et al., 2011), *GGA* (Ansótegui et al., 2009) or *irace* (López-Ibánez et al., 2011) have been developed.

These procedures excel at the task of finding high-performance areas of the configuration space of an algorithm. To determine which configurations might perform well for the given algorithm, these procedures need to be run on instance sets that are representative of the problem domain.

On homogeneous sets a tool has to identify a configuration that performs well on only a few instances, since the algorithms performance on those instances will be similar. For heterogeneous sets, the instances can be in some sense orthogonal, meaning that a configuration that solves one instance best might be the worst on a different one and vice versa.

For homogeneous instance sets, these algorithm configuration procedures are able to easily identify a configuration or a set of configurations that is able to solve the whole instance set well (with respect to the specified cost metric). However the more heterogeneous the instance set is, the more difficult the task becomes for algorithm configuration procedures (Schneider and Hoos, 2012).

	x	y	$m(A, \pi_i)$
π_0	**0**	**0**	**0.59**
	0	1	1.52
	1	0	5.24
	1	1	33.57
π_1	0	0	18.85
	0	1	3.96
	1	**0**	**1.99**
	1	1	6.47

Table 1.1: *Values for m. The best configurations for each instance are highlighted in bold.*

To make this example more clear, let A be an algorithm with two parameters x and y that each can take the values $\{0, 1\}$. Further, let Π be a heterogeneous set of instances consisting of two instances π_0 and π_1. The arbitrary cost metric m has to be minimized by the configuration tool. The values for m are listed in Table 1.1. On instance π_0 configuration $x = 0, y = 0$ whereas on π_1 configuration $x = 1, y = 0$ performs the best. A classic algorithm configurator would try to optimize the mean cost over the whole set.

For our purposes, the default configuration of algorithm A in this setting is $x = 1, y = 0$. This configuration would result in a mean cost of 3.62. A configuration tool would return configuration $x = 0, y = 1$ as the best configuration with a mean cost of 2.74. However, if a tool would be able to incorporate the knowledge about the homogeneity of the set and therefore optimize π_0 and π_1 separately, the mean cost would be 1.29.

This example is slightly exaggerated to better show the impact an instance-aware approach could have. However in practice, we will likely have to deal with instance sets of different degrees of heterogeneity. It is therefore necessary to at

least be able to identify the homogeneity of a set of instances to automatically apply *per instance algorithm configuration* (*PIAC*) instead of the classical configuration approaches.

Furthermore a per instance approach is related to algorithm selection (Rice, 1976). PIAC procedures not only have to configure an algorithm on a per instance basis but also have to select which of the found configurations should be applied to new instances. Thus solutions to the PIAC problem merge these two fields by first searching for well performing configurations that are capable of efficiently solving different parts of an heterogeneous instance set; thereby creating a portfolio of configurations. Secondly, a model has to be learned that is able to select the best configuration for a given instance out of the created portfolio.

The contributions of this thesis are as follows, we will first discuss the current state-of-the art of per instance algorithm configuration systems. We were inspired by these systems to develop our own PIAC system. As part of it, we introduce an exploration phase into the PIAC work-flow. We take the idea of *profile expected improvement (PEI)* (Ginsbourger et al., 2014; Bossek et al., 2015) and use it as an acquisition function to guide the exploration of an algorithms configuration space. We further present a modified version of *Cost-Sensitive-Hierarchical Clustering (CSHC)* (Malitsky et al., 2013). With it, our PIAC system is able to take into account a configurations performance, when splitting an instance set into subsets. This allows our approach to group instances together that are solved well by the same configuration. We further empirically evaluate our new system and compare its capabilities to that of the current state-of-the-art PIAC systems.

In Chapters 2 and 3, we will discuss aspects of related work of per instance algorithm configuration. Chapter 4 introduces already existing PIAC systems. Further, in Chapter 5 we present our PIAC system. We perform an empirical evaluation of our PIAC system and established methods in Chapter 6. In Chapter 7 we show our conclusion and discuss possible future work.

Chapter 2

Algorithm Configuration

In this chapter we present algorithm configuration. It is one of two parts that form *per instance algorithm configuration*. In the first section we will discuss benefits of using algorithm configuration and show impressive performance gains that were achieved, compared to using a default parameter setting. Further, we will discuss the general algorithm configuration work-flow. In the second section we will present the idea of empirical performance models as they play a crucial role in well established configuration procedures.

2.1 Benefits of Algorithm Configuration

The behavior of state-of-the-art algorithms is controlled by their (multitude of) parameters and the values these can be set to. Consider the commercial *CPLEX*[1] solver; a solver for *mixed integer programming* problems. It has 80 configurable parameters that heavily influence the search mechanism. For example, the parameter **backtracking tolerance** controls how often *CPLEX* will backtrack. This parameter can take values in the interval $[0, 1]$. Low values increase the amount of backtracking which turns *CPLEX*'s search more into a breadth-first search whereas increasing this parameter changes the search behavior more into a depth-first search.

For a problem instance in which the search tree is very shallow but wide, lower values for this parameter should theoretically result in *CPLEX* finding a solution faster. Whereas problems whose search trees are deep and very thin should benefit from a larger value for the backtracking tolerance. It however is not as straight forward as it might seem to set this value. In order to find the correct setting of this parameter, a user needs to have prior knowledge about the potential search trees over a set of instances. Without such prior knowledge, a user will have to reason about the influence of the parameter on the performance of *CPLEX* on a small subset of instances to find the correct setting. This might still be

[1] `www.ibm.com/software/commerce/optimization/cplex-optimizer/`

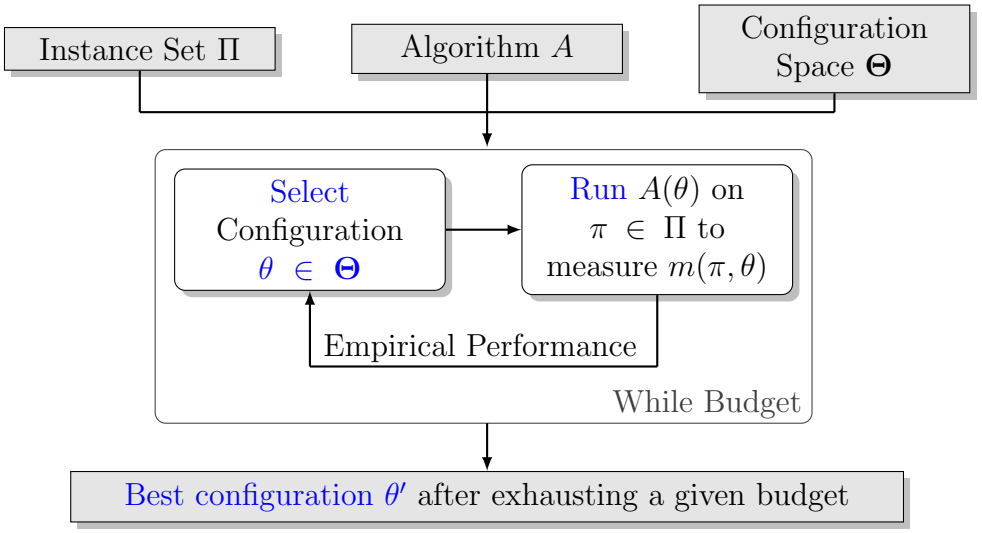


Figure 2.1: *Algorithm configuration (AC) work-flow. Given an instance set* Π*, algorithm* A *and its configuration space* $\mathbf{\Theta}$*, the AC process will determine a well performing configuration by (1) measuring As performance and using this information to (2) select a promising new configuration. The AC process will alternate between (1) and (2) until a given budget such as wallclock-time is exhausted.*

possible for users with only one parameter to tune, but with a growing number of parameters this task quickly becomes impossible. Hutter et al. (2009) presented the configuration procedure *ParamILS* and showed that *CPLEX* greatly benefits from being automatically configured, reporting a speedup factor of 2 on one dataset and 23 on a second one.

Fawcett and Hoos (2016) were able to show that the improvement between two configurations can most often be explained by only a few parameters. They give as example a 437-fold speedup of the SAT solver *Spear* (Babic and Hutter, 2007) on software verification instances, that was almost exclusively (99.7%) achieved by a single parameter. Biedenkapp et al. (2017) report speedup factors after changing only the most important parameter that range from 1.2 for *CPLEX* on the Red-cockaded Woodpecker (RCW2) instance set (Ahmadizadeh et al., 2010) to 332.6 for *Spear* on software verification instances. It is however not easy to identify which parameter will have the most influence prior to configuration. In order to set the most important parameters correctly, a user will have to look at all parameters and determine which combinations perform well. Algorithm configuration does this implicitly on the fly by predicting which potential candidate configurations will result in a potential improvement over the best seen so far.

Definition 1 *Given a set of training instances* Π*, an algorithm* A *with its possible parameter settings* $\mathbf{\Theta}$ *and a cost metric* $m\colon \mathbf{\Theta} \times \Pi \to \mathbb{R}$*, the problem of algorithm configuration is to find a parameter configuration* $\theta^* \in \mathbf{\Theta}$ *that minimizes the cost metric* m *across the instance set* Π*. That is,* $\theta^* = \arg\min_{\theta \in \mathbf{\Theta}} \sum_{\pi \in \pi} m(\theta, \pi)$.

Figure 2.1 shows the general (automated) algorithm configuration workflow and Definition 1 defines the algorithm configuration problem. Given a set Π of instances π, an algorithm A and the parameter configuration space $\mathbf{\Theta}$, the algorithm configuration work-flow iteratively selects a configuration θ and an instance π on which to run A parameterized on θ. In each iteration the empirical performance is measured $m(A(\theta), \pi)$ and kept track of to reason about which configuration to choose next. Whereby the system also keeps track of the so far best seen configuration. In the next iteration, the selected configuration can either be a so far unseen configuration that might improve over the current best or a configuration that has already been seen. The latter will give the system more confidence about the estimated performance of the configuration and lets it determine if a challenger configuration is better than the current best or not.

Automated configuration might benefit developers not only by giving their algorithms a speedup in finding solutions but also allows them to find better ones or to find solutions where handcrafted configurations fail.
For this example consider a planning algorithm that tries to find a plan that uses a minimal number of actions to reach a goal. This toy planner has one parameter that specifies which metric should be used as a heuristic of the length of a resulting plan. For the purpose of the example, the distance to travel is chosen as heuristic h. The planner further has an action set consisting of turn $\pm 15°$, forward n-meter and stop. This toy planner first samples a set of trajectories / paths, evaluates the potential solution quality, according to h and then builds the plan for the potential best trajectory.

Figure 2.2 shows a few example paths that the algorithm might consider. If the Euclidean distance is used as heuristic, then the planning algorithm would prefer the diagonal blue line and would construct the plan for this trajectory. Due to its restricted action set, the plan would be long and consist of a multitude of turns and forwards to approximate the trajectory of the blue line (in a similar fashion to the beginning of the magenta step-like line). If however the heuristic would be the Manhattan distance, then the algorithm would not prefer the blue line and might choose the red line, which results in a shorter plan.

Another important aspect of algorithm configuration is that it enables users to better compare the performance of two algorithms or evaluations of algorithms. If a user compares two algorithms just by their default configuration on the same instance set, the result might show one algorithm to slightly outperform the other. That does not definitively show the superiority of one of the algorithms. The default setting of one of the two algorithms might just be more suited for the chosen set on which they are compared. This would result in an unfair comparison and might lead to wrong conclusions. If however both algorithms are optimized, using the same configuration procedure, with the same budget, the result is more meaningful since both algorithms are tuned on the benchmark and can live up to their full potential.

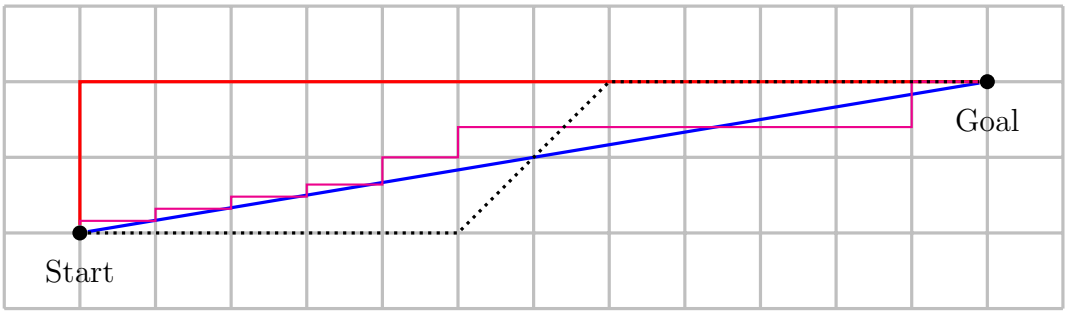


Figure 2.2: *Using the Manhattan distance as heuristic, all paths have equal length. The example algorithm discussed in the text, does not choose the diagonal blue line by default.*

2.2 Empirical Performance Models

In modern algorithm configuration systems such as *GGA++* (Ansótegui et al., 2009) and *SMAC* (Hutter et al., 2011), empirical performance models (EPMs) play the crucial role of guiding the search for promising configurations in high performance areas. Hyperparameter optimization procedures for machine learning algorithms, that are based on Bayesian optimization also use a simplified version of an EPM that does not consider instances for the predictions, (see Shahriari et al., 2016) and they have also been used as surrogates for efficient benchmarking of algorithm configuration procedures (Eggensperger et al., 2015). Hutter et al. (2014c) studied different approaches to train a regression model to predict the performance (e.g., running time) of a parameter configuration on a given instance. They showed that EPMs can be efficiently used to predict the performance of algorithm configurations on a set of problem instances. In particular for $\mathcal{NP}$-hard problems, such as Mixed Integer Programming (MIP), SAT and AI planning.

The data collected during the optimization of a target algorithm can also give valuable insight into the behavior of the target algorithm. This data can be reused to fit an EPM $\hat{m} : \mathbf{\Theta} \times \Pi \to \mathbb{R}$ which can be queried for the performance of the algorithm on given instances. These EPMs can also be used to assess the importance of parameters on the instance set used for optimization. Forward selection (Hutter et al., 2013) and *fANOVA* (Hutter et al., 2014a) are two examples that construct an EPM from the data collected during an optimization run in order to assess how important an algorithms parameter is to achieve an optimal performance. Ablation (Fawcett and Hoos, 2016) is another method for assessing parameter importance. Ablation however does not use an EPM to determine the final values. It can be used after configuration to determine which parameters between a default configuration and the target configuration had the most influence in the final result.

The AI community has used algorithm configuration procedures to achieve state-of-the art performance on a multitude of applications. Thornton et al. (2013) developed Auto-WEKA and Feurer et al. (2015) Auto-Sklearn, which are able to simultaneously select a learning pipeline and set its hyperparameters. This strain of automated machine learning (AutoMl) enables non-Experts to use machine-learning systems out of the box without having to spend resources to ascertain what method and its hyperparameters work for their problem domain. AC has further been successfully applied to problems in AI planning (Fawcett et al., 2011), SAT solving (Hutter et al., 2017) and MIP (Hutter et al., 2010).

Despite the successful use of algorithm configuration in practice Eggensperger et al. (2017) observed fundamental issues with experimental setup of algorithm configuration experiments. As recommendations when and how to apply algorithm configuration to obtain the best results they state that algorithm configuration in and of itself works best on homogeneous benchmarks. On these, a configuration procedure can easily generalize the configuration performance over the whole instance set. On heterogeneous sets the problem becomes more difficult as configurations that solve some instances very well, might completely fail to find a solution on others. Moreover, they suggest to use at least 300 instances for configuration on a heterogeneous set but suggest that more than 1000 will lead to a more robust result.

In this chapter we presented and motivated the algorithm configuration paradigm. We showed its capabilities and discussed the standard algorithm configuration work-flow. In the next chapter we will present algorithm selection, as it together with algorithm configuration is one of the building blocks of *per instance algorithm configuration*. Algorithm selection is a widely used tool to solve problem instances in a more efficient way, similarly to algorithm configuration, nevertheless the methodology to achieving this goal is quite different.

Chapter 3

Algorithm Selection

In this chapter we present algorithm selection, as it is the second main pillar of per instance algorithm configuration. We will present the algorithm selection work-flow and discuss algorithm selection using two prominent systems to show possible solutions to the algorithm selection problem. The AI community deals with a lot of different problem domains, such as Answer Set Programming (ASP), Satisfiability (SAT) solving, Constraint Satisfaction Problems (CSP) or Quantified Boolean Formulas (QBF).

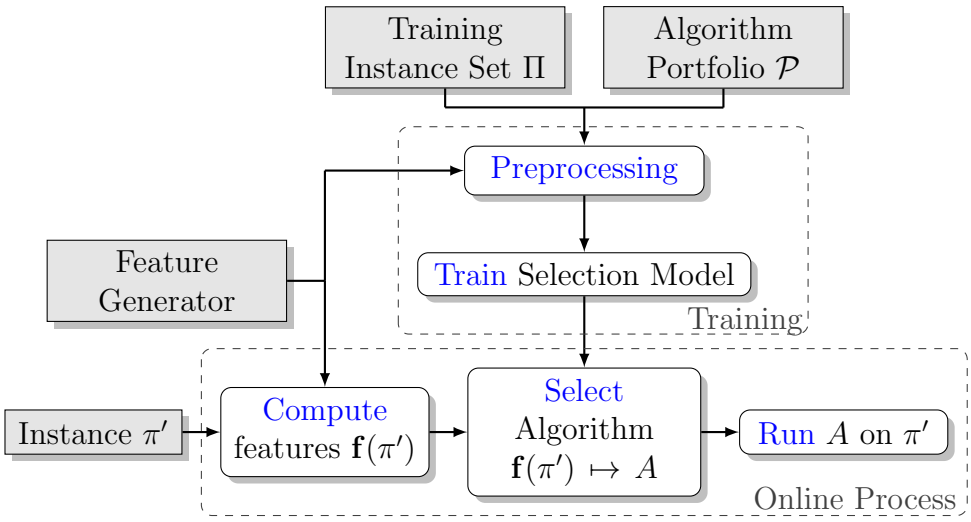


Figure 3.1: *Simplified algorithm selection (AS) work-flow. In order to select the best algorithm A out of a portfolio $\mathcal{P}$ of algorithms, a training phase is necessary in which a selection model is constructed. Given a training instance set Π, portfolio $\mathcal{P}$ and a feature generator, the selection model is trained to select the algorithm that solves an instance best, given the portfolio performance values. For unseen instances, features are computed which are used to select which algorithm to run on the instance.*

Definition 2 *Given a set of training instances* Π*, a portfolio of algorithms* $\mathcal{P}$ *and a cost metric* $m\colon \mathcal{P} \times \Pi \to \mathbb{R}$ *the algorithm selection problem is to learn a selector* $\mathcal{S}\colon \Pi \to \mathcal{P}$ *that is able to select an algorithm* $A \in \mathcal{P}$ *for a given instance* $\pi \in \Pi$ *such that* A *performs best on the instance* π *at hand. That is,* $A = \arg\min_{A' \in \mathcal{P}} \sum_{\pi \in \pi} m(A', \pi)$.

Progress in these areas can be largely attributed to the development of many solvers with complementary solving strategies. Regardless, there does not exist one dominant solver that is superior on all types of problem domains. Algorithm selection (Rice, 1976) takes advantage of this fact and uses instance characteristics, so called features, to select a promising algorithm to solve an instance.

Figure 3.1 depicts a simplified variant of the algorithm selection work-flow and Definition 2 defines the algorithm selection problem. To solve the problem, a selection model has to be learned in an offline training phase. It expects a portfolio of algorithms to choose from and a set of features that can be used or computed via a feature generator. During the training process, the model learns to decide which of the available algorithms to choose, given the features of an instance. In the online selection phase, features for an unseen instance have to be computed. These are then fed into the selection model. The selection strategy determines which of the possible algorithms selected out of the available portfolio to solve an instance.

Selection strategies can take different shapes. A simple strategy consists of pairwise predicting whether an algorithm will outperform the other. The winner of this voting will then be used to solve an instance. Similar to the "winner" selection strategy, a weighted variant can be used to decide which algorithm is to be used for solving. The weighting could be the difference by how much an algorithm outperforms another, informing the selection method not only of the fact that an algorithm is outperformed by another, but also of the significance of it. Meaning that if the difference between the performance values is large, the model will learn that an algorithm is not only outperformed but also that if the wrong algorithm were to be selected, poor performance will be achieved on that instance. A scheduling strategy could also be learned where multiple algorithms can be selected and applied to the instance in a schedule.

The first version of SATzilla (Xu et al., 2008) uses regression models to predict the performance values for all algorithms in the portfolio. It then selects the algorithm with the best predicted performance value to solve an instance. In a later version, Xu et al. (2012) use pairwise cost-sensitive classification models to predict how strongly an algorithm is outperformed. Voting over these predictions leads to the choice of the algorithm to use. SATzilla also pioneered pre-solving schedules. These are instance independent, static schedules that can save a selection method from having to compute expensive instance features if the pre-solving schedule was able to solve an instance.

3S (Kadioglu et al., 2011) is a k-nearest neighbor approach that first determines the k-nearest neighbors in the training set and chooses an algorithm that outperformed others on this small set to solve a new instance. The selection employs a distance-based weighting, taking into account how dissimilar the k neighbors are to a new instance. 3S further uses a pre-solving schedule based on mixed integer programming. Both systems are more than capable of outperforming single solver based systems and won SAT competitions.

Further systems combine these techniques or extend them. For example, a combination of clustering and a regression model approach was presented by Collautti et al. (2013). Yun and Epstein (2012); Lindauer et al. (2015a) presented methods to select portfolios (e.g. to run in parallel) instead of a single best algorithm and Amadini et al. (2014) presented an approach that selects schedules instead of single algorithms. AutoFolio (Lindauer et al., 2015b) combines algorithm configuration and selection by automatically configuring algorithm selectors to further improve a selection approach.

In this chapter we presented the algorithm selection work-flow and different selection methods. We presented two prominent examples of selectors and shortly presented further extensions to the algorithm selection setting. In the following chapter we will present the *per instance algorithm configuration* problem that combines algorithm configuration with algorithm selection.

Chapter 4

Per Instance Algorithm Configuration

As seen in the previous two chapters, algorithm configuration is a powerful tool to improve the solution quality or running time of an algorithm. Regardless, it only tries to find one parameter setting over a set of instances. This works well if the instance set is very homogeneous (Schneider and Hoos, 2012), i.e. the instances all bear similar characteristics. For heterogeneous sets however, a configuration procedure might only be able to find a parameter setting that improves at best slightly over a default configuration. A portfolio based solution would be more suited to solve instances from a heterogeneous set, where a selector should choose from complementary configurations to solve a given instance. *Per instance algorithm configuration* (PIAC) is concerned with finding a set of configurations that are well suited for solving heterogeneous sets of instances and learning a model that is capable of deciding which configuration to apply to unseen instances. The PIAC problem is defined as follows:

Definition 3 *Given an algorithm A with its possible parameter settings $\mathbf{\Theta}$, a set of training instances Π, and a cost metric $m\colon \mathbf{\Theta} \times \Pi \to \mathbb{R}$, the problem of per instance algorithm configuration is to learn a selector $\mathcal{S}\colon \Pi \to \mathbf{\Theta}$ that is able to select a configuration $\theta \in \mathbf{\Theta}$ for a given instance $\pi \in \Pi$ such that θ performs best on the instance π at hand. That is, $\theta = \arg\min_{\theta' \in \mathbf{\Theta}} \sum_{\pi \in \pi} m(\theta', \pi)$.*

So far few attempts at creating PIAC systems have been done. The existing methods try to either partition the instances into distinct sets on which algorithm configuration procedures can be used to find well performing configurations or directly searching for complementary configurations on the whole instance set. Thus all PIAC systems rely on a distance metric. For a partitioning of the instances this metric exists in the feature space, whereas it exists in the performance space when searching for complementary configurations.

We will first discuss two methods that determine a partitioning in the feature space before showing a method that searches for complementary configurations

according to an algorithm's behavior in the performance space on a given instance. We will further discuss advantages and shortcomings of these methods as these inspired us to create our own PIAC system that we will present in a later chapter.

4.1 Stochastic Offline Programming

Malitsky and Sellmann (2009) presented *Stochastic Offline Programming* (SOP). SOP is an iterative approach. Its partitioning of the instances works by iteratively adjusting a distance metric in the feature space that decides into which partition an instance belongs. It was designed to optimize algorithms of a particular structure. Target algorithms would need to choose a heuristic out of a distribution of heuristics that was either designed by hand or optimized by SOP. Figure 4.1 shows the training process of SOP where the configurations represent the distribution of heuristics.

SOP starts with an arbitrary distance metric (e.g. Euclidean Distance) and uses this metric with k-means clustering (Lloyd, 1982) to separate the instance set Π into k distinct partitions $\Pi_1 \subsetneq \Pi, \ldots, \Pi_k \subsetneq \Pi$. After being randomly initialized, it alternates between two steps. At first, it determines into which partition an instance belongs by taking the argmin of all distances to the partition centers. In the subsequent step, the centroids are updated to lie directly in the center of all points in that partition. The steps are repeated until the partitions have converged. On each of the resulting partitions, a configuration procedure is used to determine an optimized configuration θ_i. These configurations then are used to update the distance metric, by computing the performance difference of applying θ_i on each instance of all other partitions. The distance for all instances in Π_j to the centroid in Π_i then is the maximum of the performance difference of applying θ_i on Π_j instead of θ_j. Now the distance metric has to take into account the penalization it would get for packing an instance into a partition whose optimal configuration can not or only badly solve that instance.

Figure 4.2a gives an example for of a 1D 3-means result and Figure 4.2b shows how a possible updated metric would look like for instances in partition A. For this example the instances in A are solved best with the optimized configuration of partition A, second best with the configuration of partition C and worst with the optimized configuration of partition B.

In the next iteration, the updated metric is used to partition the instance set again. The whole process runs until no adjustment of the metric is necessary and all instances are packed into the partition to which it has the minimal distance.

For each partition, SOP returns the optimized configuration and the centroid. It additionally returns the final distance metric. Unseen instances are solved by computing their features, using these to determine the distance value to each centroid and applying the configuration of that centroid on the instance.

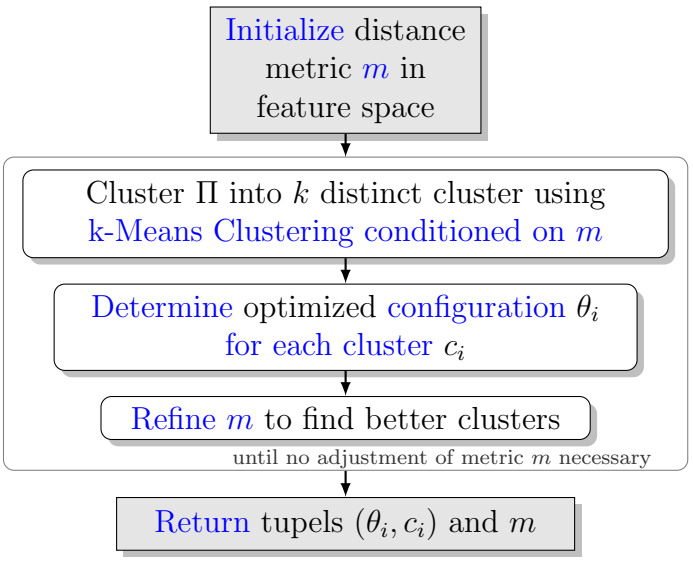


Figure 4.1: *Stochastic Offline Programming: SOP is an iterative process. It first initializes a metric m in the feature space $\mathcal{F}$. It clusters an instance set Π into k distinct clusters using k-Means Clustering conditioned on m. On each of the resulting clusters, optimized configurations θ_i are determined. These are used to determine the quality of the clusters, i.e. if instances in cluster c_i are better solved with θ_j instead of θ_i. The process iterates over all steps until the metric m does not have to be adjusted anymore. Unseen instances are solved with the configuration of the cluster to which the instance is closest.*

The authors of SOP reported positive results using their method to optimize the solution quality of a heuristic for set covering. Comparing to a virtual best solvers (VBS) performance they showed that there is room for improvement. The VBS is a theoretical solver that is capable of perfectly selecting the correct configuration for each instance. They note that one flaw of their method was that they did not normalize the feature values for their experiments and thus some features dominated the whole feature vector. They further acknowledged that leaving the k of k-means as a parameter to choose for the user as suboptimal. They performed additional experiments in which they compared their "standard" SOP approach to SOP with normalized features, using g-means clustering (Hamerly and Elkan, 2004, see Algorithm 1) and SOP using both enhancements. The results showed that these improve the performance (on the same benchmarks), but only slightly.

SOP is easily parallelizable where each optimization run could be performed on a single core as well as computing the loss for each partition. However, if the instance set is large, a great chunk of time and resources has to be allocated into refining the distance metric, since the method in SOP more or less needs to validate the whole instance set with k different configurations to refine the metric

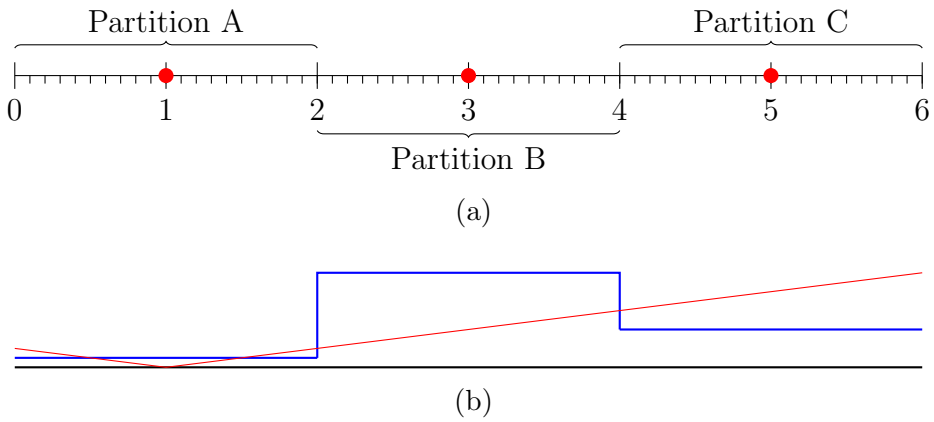


Figure 4.2: *(a) shows a simple 1D partitioning obtained using 3-means with the Euclidean distance metric. (b) shows distances to centroid of partition A. The red diagonal is the classical euclidean distance, the blue line is an adjusted metric. It is the maximum performance difference of applying θ_i on Π_j where $i \neq j$. This penalizes k-means for packing instances into partitions and decreasing the partitions mean performance thereby.*

accurately.

4.2 Instance Specific Algorithm Configuration

Kadioglu et al. (2010) improved over SOP with *Instance Specific Algorithm Configuration* (ISAC). They start by replacing design choices of SOP with ones they think are crucial to achieving better performance. The two main aspects they want to replace are the usage of unnormalized features and the user choice of k in k-means. In order to avoid dominance of some features in the feature vector, they normalize the features to fall into the range $[-1, 1]$. To take the burden of finding the correct k of a users shoulders, they also replace k-means clustering by g-means (the same enhancements the SOP developers tested in a later iteration.). Another problem they identified is that the modification of a distance metric for the feature space is not straight forward when trying to optimize running time of an algorithm on a heterogeneous instance set. Configurations that are not very well suited for a set of instances might run a very long time before terminating or being cut off early. Timeouts thus would only allow to compute a lower bound on the distance as presented in SOP. The authors further say that when trying to optimize running time of an algorithm in a similar fashion to SOP they observed a lot of timeouts when trying to update the distance metric. This results in a poor metric that can not be used to accurately pack instances together that behave similar under a given configuration, which takes a lot of time and resources.

Thus they drop the iterative approach of SOP and go with a one-shot strategy.

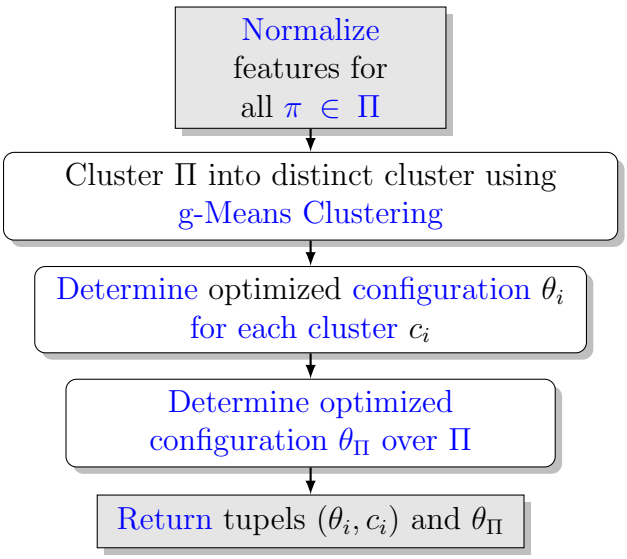


Figure 4.3: *Instance Specific Algorithm Configuration: ISAC is a one-shot PIAC approach. It uses g-means clustering to cluster the instance set into distinct sets only once. On each of the clusters optimized configurations are determined using an existing algorithm configuration procedure. Additionally, as backup an optimized configuration over the whole set is computed. Unseen instances are solved with the configuration of the cluster to which it is closest.*

The whole ISAC process is depicted in Figure 4.3. Before the partitioning of the instances into distinct sets is performed, all features are normalized. The preprocessed features are then used to compute the centroids and partitions using g-means clustering. Algorithm 1 shows the workings of g-means clustering. This algorithm is able to determine the number of clusters automatically.

Hamerly and Elkan (2004) propose that a good partition is approximately Gaussian distributed around the cluster center. The algorithm first considers all instances to belong into the same partition. In each iteration, one of the existing partitions is taken to determine if this partition is already split far enough to be considered Gaussian. To test this, the partition is split using 2-means clustering into two smaller partitions. All instances in the partition are then projected onto a line that runs through the centers of the two new partitions. This yields a one dimensional distribution of points. This distribution can now be tested for its degree of normality using the Anderson-Darling-Test (Anderson and Darling, 1954). If the test fails, this means that the distribution does not resemble a normal distribution and the partition gets split into the two computed partitions.

For each of the resulting partitions an optimizer is used to determine a well performing configuration. Additionally, as backup an optimized configuration over the whole instance set is computed. ISAC then returns the optimized configurations, the corresponding centers and the backup configuration.

Algorithm 1: g-Means Clustering (Hamerly and Elkan, 2004)

Data: Instance Set Π, Distance metric m
Result: Partitioning $\mathbf{H}$ of Π, Centroids $c_i \in C$

$i, g \leftarrow 0, 0$
$\mathbf{H}, C \leftarrow k\text{-means}(\Pi, m, 1)$ **while** $i \leq g$ **do**
  $\mathbf{H}', C' \leftarrow k\text{-means}(\mathbf{H_i}, m, 2)$
  $v \leftarrow c'_1 - c'_2$
  $\mathbf{H}'' \leftarrow \frac{\sum_{i=0}^{|\Pi|} \pi_i v_i \mathbb{1}_{\pi_i \in \mathbf{H}'}}{\sum_{i=0}^{|\Pi|} v_i^2 \mathbb{1}_{\pi_i \in \mathbf{H}'}}$ `// Project all instances in` $\mathbf{H}'$ `onto a line`
  if ***Anderson-Darling-Test(***$\mathbf{H}''$***)failed*** **then**
    $\mathbf{H_i}, C_i \leftarrow \mathbf{H}'_1, C'_1$
    $k \leftarrow k + 1$
    $\mathbf{H_k}, C_k \leftarrow \mathbf{H}'_2, C'_2$
  else
    $i \leftarrow i + 1$
return $\mathbf{H}, C$

When using ISAC to solve unseen instances, first the instance features are computed. The same scaling and translation factors as during the training phase are used to normalize the features to be in the range $[-1, 1]$. To determine which configuration to use in order to solve the instance, the distance to each cluster center is computed. The configuration of the center that is closest is used to solve the unseen instance. If the instance is too far away from the centers, the backup configuration will be used to solve the instance. Following Kadioglu et al. (2010) we deemed an instance as being too far away from a cluster center in our experiments, if an instance was more than the average distance plus two standard deviations of all points in a partition away from the center.

The backup configuration provides ISAC with a robust configuration to fall back on. This should be helpful as ISAC does not have to risk using a configuration that is not suitable for solving the instance but can use a configuration that was evaluated to behave well over the whole heterogeneous subset.

Figure 4.4 shows a range of results obtained using k-means clustering, using the Euclidean metric to determine similarity. For the example the true number of clusters is 6. Figure 4.4a shows the results obtained using 2-means clustering. For this result it is clear from the image that the obtained cluster should be further split into subsets. Figures 4.4b and 4.4d show results for $k = 4$ and $k = 8$. The resulting partitions look more promising. Figures 4.4c and 4.4e show the results for $k = 6$. For figure 4.4e the value for k was automatically determined using g-means clustering. The resulting partitions are very similar to the ground truth depicted in figure 4.4f. This example shows that g-means clustering can find the correct value of k easily if the feature space is informative enough. If this space

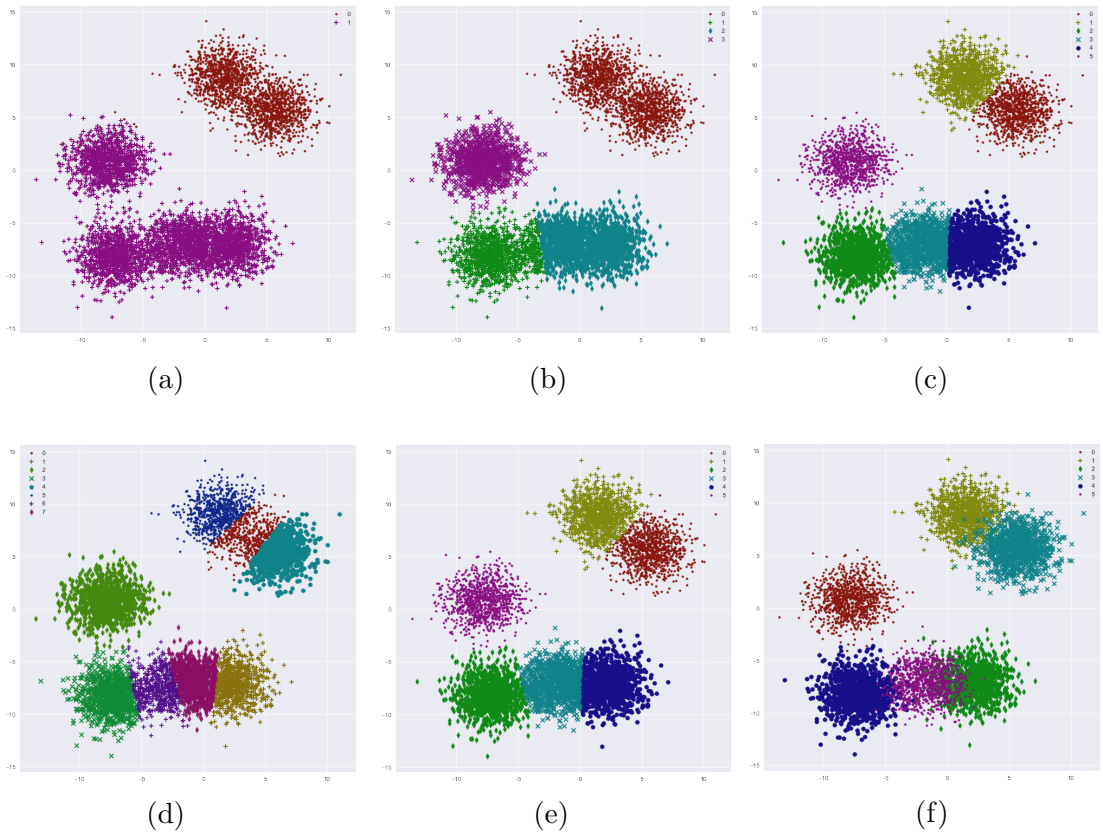


Figure 4.4: *Example results using k-means clustering on artificial data. (a) - (d) show results of k-means clustering for $k = 2, 4, 6, 8$. (f) shows the ground truth data. The result shown in (e) was obtained using g-means clustering, automatically determining that $k = 6$ is the correct value for this example.*

is low dimensional and very informative, *g*-means is the easiest way of finding the correct k without any additional user input.

Not being an iterative method allows ISAC to spend very little time in trying to find the optimal split and spending virtually all the allocated budget on only optimizing the partitions, which could allow it to find better performing configurations than its iterative competitors. However this is only the case if the resulting splits are sufficiently homogeneous.

One criticism that concerns ISAC is that it determines the partitioning in the feature space only. The partitioning in the feature space is very dependent on the distance measure provided. It might result in instances that look similar, but do not behave similar, to be packed together. More specifically, the problem is that the presented clustering methods do not consider how instances might behave under a specified configuration.

This could lead ISAC to pack instances together that have opposing behavior under a given configuration, leading to new (slightly less) heterogeneous subsets.

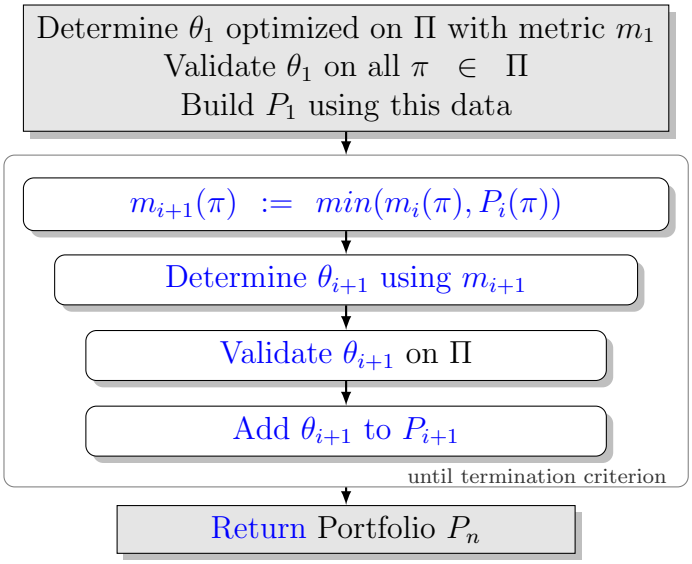


Figure 4.5: *Hydra: The goal of Hydra is to build a Portfolio of complementary configurations and does so in an iterative fashion. It first determines an optimized configuration using an off-the-shelve AC procedure. It then adjusts a performance metric with which it determines an optimized configuration that is complementary to the configuration already selected. A new configuration then is evaluated on the whole instance set. This is necessary in order to gather enough data to determine when a given configuration should be selected. Hydra repeats these steps until a predefined termination criterion is met. The portfolio is used to determine which configuration should be used to solve unseen instances.*

Since these sets consist of only a fraction of the original set, the task of finding a well performing or robust configuration is much more difficult for the optimization procedure even though the instances are very similar according to the metric used for clustering. After each iteration SOP adjusts its distance metric with respect to the performance values, which allows for an implicit use of performance values during the partitioning. During the partitioning however the distance metric is fixed and can not reason about the performance gains or losses that result from new partitions.

4.3 Hydra

Hydra was first introduced as a PIAC approach that does not partition the training instance set (Xu et al., 2010). In a similar fashion to SOP, Hydra is an iterative process that modifies a metric in each iteration. Unlike SOP however the metric is not defined on the feature space but on the performance space. Figure 4.5 shows the whole process of Hydra. In an initial step an optimized

	$m(\theta_0, \pi_i)$	$m(\theta_1, \pi_i)$	$m(\theta_2, \pi_i)$	$y_{\theta_0<\theta_1}$	$w_{\theta_0<\theta_1}$	$y_{\theta_0<\theta_2}$	$w_{\theta_0<\theta_2}$	$y_{\theta_1<\theta_2}$	$w_{\theta_1<\theta_2}$
π_0	1	5	3	1	4	1	2	0	2
π_1	4	2	6	0	2	1	2	1	4
π_2	9	1	3	0	8	0	6	1	2
π_3	1	3	5	1	4	1	4	1	2
π_4	6	5	4	0	2	0	2	0	1
π_5	3	8	2	0	2	0	1	0	6

Table 4.1: *Example of Hydras weighted pairwise selection to minimize an objective value. $y_{\theta_i<\theta_j} = 1$ means that θ_i outperforms θ_j. $w_{\theta_i<\theta_j}$ gives the associated weight. The larger the weight value the more certain one can be that the y value was not pure chance that θ_i outperformed θ_j, where the weight value shows by how much a configuration was outperformed by another.*

configuration θ_1 is searched that performs well over the whole instance set. This configuration additionally is validated on all instances in the set. This data then is used to build the initial portfolio P_1 that would always select θ_1.

In the iterative part of Hydra, at the beginning of each new round, Hydra modifies the metric that its configuration procedure uses to find a new well performing configuration. The metric m_{i-1} is modified to always return the better performance value (determined with the initial metric m_0) of the configuration under test θ_i and the current portfolio P_i performance. The configuration procedure then is applied on the whole instance set to find a new well performing configuration θ_{i+1} over the whole instance set, using the modified metric as its performance measure. As in the initial step, θ_{i+1} gets validated on all instances in Π. The collected data is then used to create a new portfolio P_{i+1}. Xu et al. (2010) used the first version of SATzilla (Xu et al., 2008) as the selector in their experiments.

This process is repeated until a specified budget (e.g specified timelimit or maximum number of iterations) is exhausted. In the end, Hydra returns the final Portfolio. To select a configuration to solve an unseen instance, Hydra uses SATzilla's selection method based on pairwise weighted random forests classifiers to pairwise predict which configuration will outperform the others. The configuration that is predicted to outperform the most other configurations is chosen to solve the new instance.

Table 4.1 gives a toy example to better understand the pairwise weighting that SATzilla uses to select which configuration to apply. The table shows if a configuration outperforms another, i.e. it results in a lower value, and the corresponding weight of that outcome. A large weight value shows that the choice between algorithms is more important than ones with small weights. No configuration dominates, i.e. is always better than all other configurations, showing

that a portfolio approach or instance specific approach results in a better mean performance than using an instance unaware or standalone approach.

On π_0, θ_0 outperforms θ_1 and θ_2, giving it a score of 2. θ_2 outperforms θ_1, giving it a score of 1. This leaves θ_0 with the largest score and it would be chosen to solve π_0 whereas π_5 would be solved using configuration θ_2.

Hydras method of adapting the performance metric never penalizes a configuration for being worse than the portfolio, since the portfolio would never choose this configuration to be used to solve an instance. Configurations are however rewarded for outperforming the portfolio. This method of finding new configurations allows for quite diverse configurations with opposing behavior to form a portfolio leading to a robust portfolio that can deal with heterogeneous instance sets. Further, the inclusion of configurations to the portfolio will never decrease the overall portfolio performance since bad configurations will never be chosen to be run on any instance. In each iteration it is likely that the overall performance increases if the configurator was able to find a new configuration that is able to solve instances better, which were previously only solved poorly or not solved at all. With an ever increasing portfolio size it is also to be expected that the improvement of adding a new configuration will decrease. This leads us to the assumption that Hydra optimizes a submodular function, which would give it tight performance guarantees (Nemhauser et al., 1978).

Hydra does not depend on a partitioning in the feature space. The optimization of configurations is always done on the whole set. This way, suboptimal groupings of instances into small sets that are only slightly less heterogeneous than the original set are avoided and a configuration can always be searched on a quite large set. This leaves the configuration procedure with more room to explore a configuration's real behavior over the whole set.

One of Hydras shortcomings concerns the optimization procedure. As Hydra always returns the best performance of the portfolio and a possible new configuration, a configuration procedure will have to be able to deal with a lot of plateaus, where it won't see any improvements. This makes the configuration step more difficult with each iteration. Further, Hydra requires more resources for the validation of each configuration.

Table 4.2 gives a toy example of how Hydra would construct a portfolio. Configurations θ_0 to θ_3 would be found in subsequent configuration runs after modifying the performance metrics. In the initialization phase, θ_0 with the best (lowest) mean performance was found. All other configurations on their own do not look promising. Nevertheless, each configuration is clearly the best one for a given instance. Thus, an adjusted performance metric that takes into account the current portfolio will not penalize θ_1 for having a worse mean performance than θ_0 but reward it for being better than θ_0 on at least one instance. Thus θ_1 gets included into the portfolio. The newly adjusted metric now won't punish promising configurations for bad performances on π_0 and π_3 but reward them for

	$m(\theta_j, \pi_i)$				$m(P_j, \pi_i)$			
Instance	θ_0	θ_1	θ_2	θ_3	P_0	P_1	P_2	P_3
π_0	5.0	1.0	4.0	8.0	5.0	1.0	1.0	1.0
π_1	6.0	4.0	5.0	2.0	6.0	4.0	2.0	2.0
π_2	4.0	7.0	2.0	9.0	4.0	4.0	4.0	2.0
π_3	1.0	8.0	9.0	9.0	1.0	1.0	1.0	1.0
μ_Π	4.0	5.0	5.0	7.0	4.0	2.5	2.0	1.5

Table 4.2: *Hydra Portfolio construction example. The middle gives example performance values for configurations θ_0 to θ_3 on four different instances. The last row gives the mean performance over the whole instance set Π. On the right you can see performance values for the constructed portfolios after n iterations.*

being better on π_1 and π_2.

In the latest version of Hydra, Xu et al. (2011) run n configuration procedures independently in parallel. Hydra might either add only one configuration out of the n found ones to extend its portfolio or multiple ones. If only one configuration is used to extend the portfolio, Hydra uses the configuration procedures internal estimate of the costs for each configuration to select which to add. The configuration whose improvement over the current portfolio is largest is then added. This is computationally cheap, as it does not require additional algorithm runs, but might choose a configuration that does not improve the portfolio. The reason for this is, that the independent configuration procedures might use the training instances in different orders and might evaluate configurations on a different set of instances. This might lead the procedure to estimate that a configuration is a good extension for the portfolio, when in reality, the configuration does not help on unseen instances. However, the used configuration procedure might also find different local optima. This would lead the parallel configuration runs to find diverse configurations.

In this chapter we motivated the per instance algorithm configuration problem and presented various existing systems that try to solve this problem in their own way. We discussed some advantages and shortcomings of the presented methods. Stemming from these observations we were inspired to create our own PIAC system which we will present in the next chapter.

Chapter 5

ISMAC

In this chapter we will present our main contribution: a novel PIAC approach. This approach was heavily influenced by the PIAC methods that were presented in the previous sections. Similar to ISAC and SOP our approach partitions the instances into different distinct partitions, similarly to Hydra and contrary to the other two methods it explicitly takes into account the performance that a possible configuration could achieve on the partitions to find a split in the feature space. Similar to SOP and Hydra it is also an iterative approach which tries to refine its partitioning after having seen real algorithm runs during its optimization pass. We will refer to our approach as *ISMAC* (short for Instance-specific *SMAC*).

5.1 Taking Performance into Account

We will first exemplify why we think that taking into account the configuration performance during the partitioning of instances is crucial to find better partitions. For ease of comprehensibility the toy example uses a centroid approach as it makes it easy to see how the partitions are formed. Table 5.1 shows the toy example consisting of nine instances. For each instance, two feature values $\mathbf{f}_i$ in the range $[0, 2]$ are listed. For the choice of taking instances π_2 and π_9 as centroids, the distances from each centroid to all other instances are listed $\mathrm{d}(\pi_i, \pi_{\{2,9\}})$. When only taking the distances to the centroids into account for the construction of the partitions, centroid π_2 spans a partition $\mathbf{h}_1$ of size five and centroid π_9 partition $\mathbf{h}_2$ of size four. Both centroids have mean distance 1 to all instances in their partition. However the resulting partitions $\mathbf{h}_1$ and $\mathbf{h}_2$ barely lead to an improvement over strictly using one of the configurations to solve the whole instance set. When the partitioning of the instances takes into account possible performances, the resulting partitioning $\mathbf{h}_i^+$ is qualitatively better and much closer to the virtual best solver. This can be observed in the last columns of the table. The mean performances are listed in the last row. When taking the performance into account when creating the partitions we are able to identify

a better split and see that π_2 and π_9 are not suited as centroids. For example, looking at the direct neighbors of π_2, we can see that even though π_1 and π_3 are similar to π_2 they are not solved best by the same configuration. Conversely π_5, another closest neighbor is solved best by the same configuration. Taking this information we can choose two new centroids, π_3 and π_4. The split that results from choosing these instances as centroids leads to an overall performance improvement. Not only are the instances grouped in such a way that they are similar with respect to their features, but also exhibit similar performances under a given configuration.

Figure 5.1 is a visual representation of Table 5.1. It shows the given partitions $\mathbf{h}_i$ and $\mathbf{h}_i^+$ as well as the resulting performances. Darker shades of red imply poor performance of a configuration on an instance. Figures 5.1a and 5.1b depict the performance of θ_1 and θ_2 respectively. The images in one row below show the discussed assignments. Figure 5.1c shows the assignment for choosing π_2 and π_9 as centroids, whereas Figure 5.1d shows a better assignment with π_3 and π_4 as centroids. We can see in Figures 5.1e and 5.1f that the split $\mathbf{h}_i^+$ is better as it shows less dark areas.

ID	$\mathbf{f}_1$	$\mathbf{f}_2$	d(π_i, π_2)	d(π_i, π_9)	$m(\theta_1, \pi_i)$	$m(\theta_2, \pi_i)$	$\mathbf{h}_1$	$\mathbf{h}_2$	$\mathbf{h}_1^+$	$\mathbf{h}_2^+$	VBS
π_1	0	0	1	4	9	1	1	-	-	1	1
$\pi_\mathbf{2}$	0	1	-	-	1	9	9	-	1	-	1
π_3	0	2	1	2	3	2	2	-	3	-	2
π_4	1	0	2	3	9	6	6	-	-	6	6
π_5	1	1	1	2	4	5	5	-	-	5	4
π_6	1	2	2	1	4	4	-	4	4	-	4
π_7	2	0	3	2	1	1	-	1	-	1	1
π_8	2	1	2	1	5	3	-	5	-	3	3
$\pi_\mathbf{9}$	2	2	-	-	4	9	-	4	4	-	4
μ	-	-	-	-	4.4	4.4	4.11		3.11		2.89

Table 5.1: *The left half of the table lists possible instances, their corresponding two feature values* $\mathbf{f}_i$ *and the distances to two centroids* $d(\pi_i, \pi_{\{2,9\}})$*. The right half lists the performances under two possible configurations* $m(\theta_i, \pi_i)$*, two resulting partitions when only considering distance to the centroids* $\mathbf{h}_i$ *and two resulting partitions when also taking the performances into account* $\mathbf{h}_i^+$*. The two centroids are* π_2 *and* π_9*. The last row shows the mean performance values. The overall mean performance when only using distance metrics is* 4.11*, compared to* 3.11 *when also considering performance values. The last column shows the virtual best solver, that perfectly chooses the best configuration per instance. It's mean performance is* 2.89 *which is only slightly better than that of the best partitioning in the feature space.*

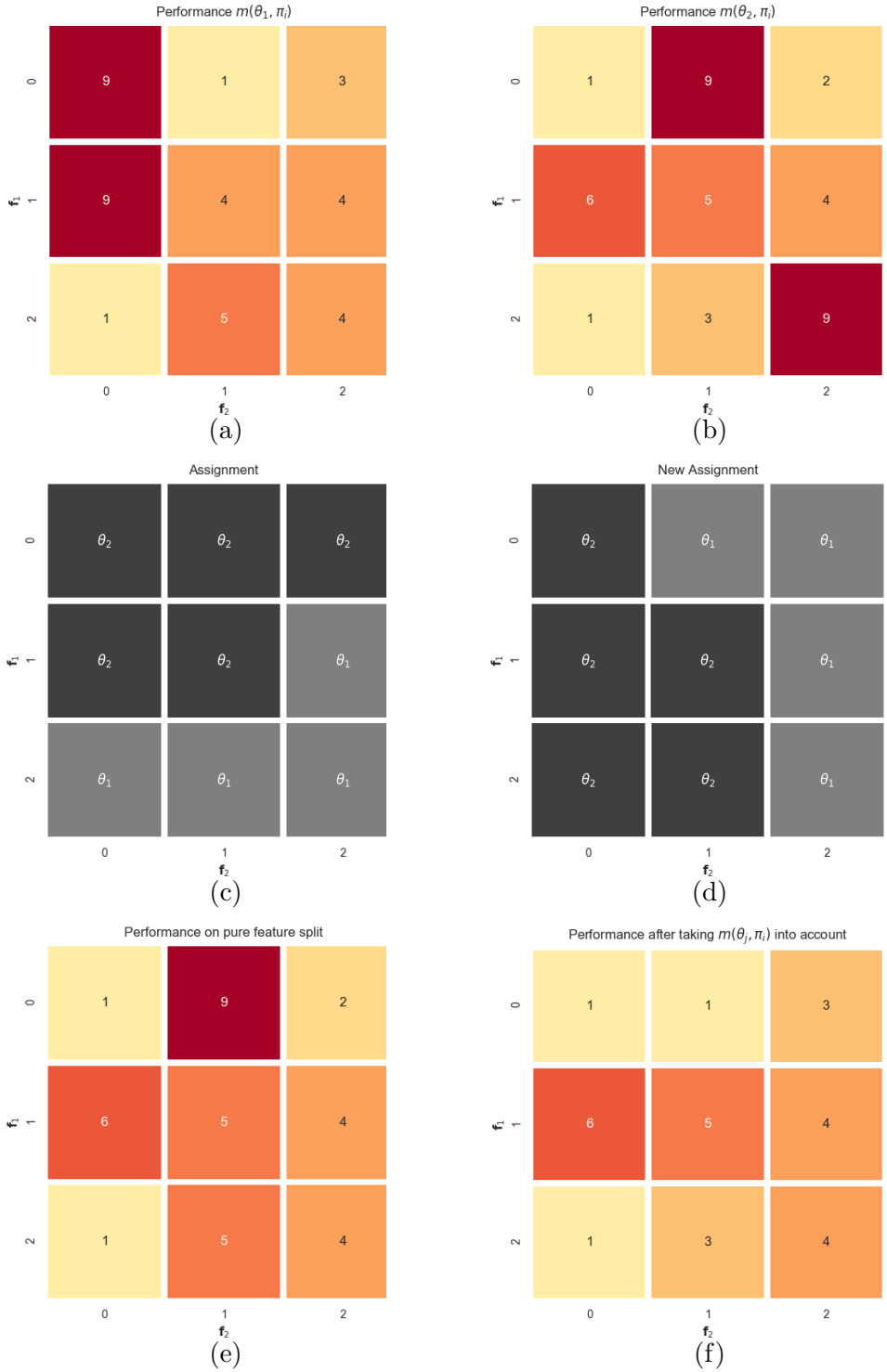


Figure 5.1: *Visual representation of Table 5.1. (a) and (b) show the performances of configurations θ_1 & θ_2. Darker shades of read imply poor performance. (c) depicts the assignment of configurations to instances when splitting purely feature based and (d) the assignment when taking individual performance into account. (e) and (f) show the resulting performances of these assignments.*

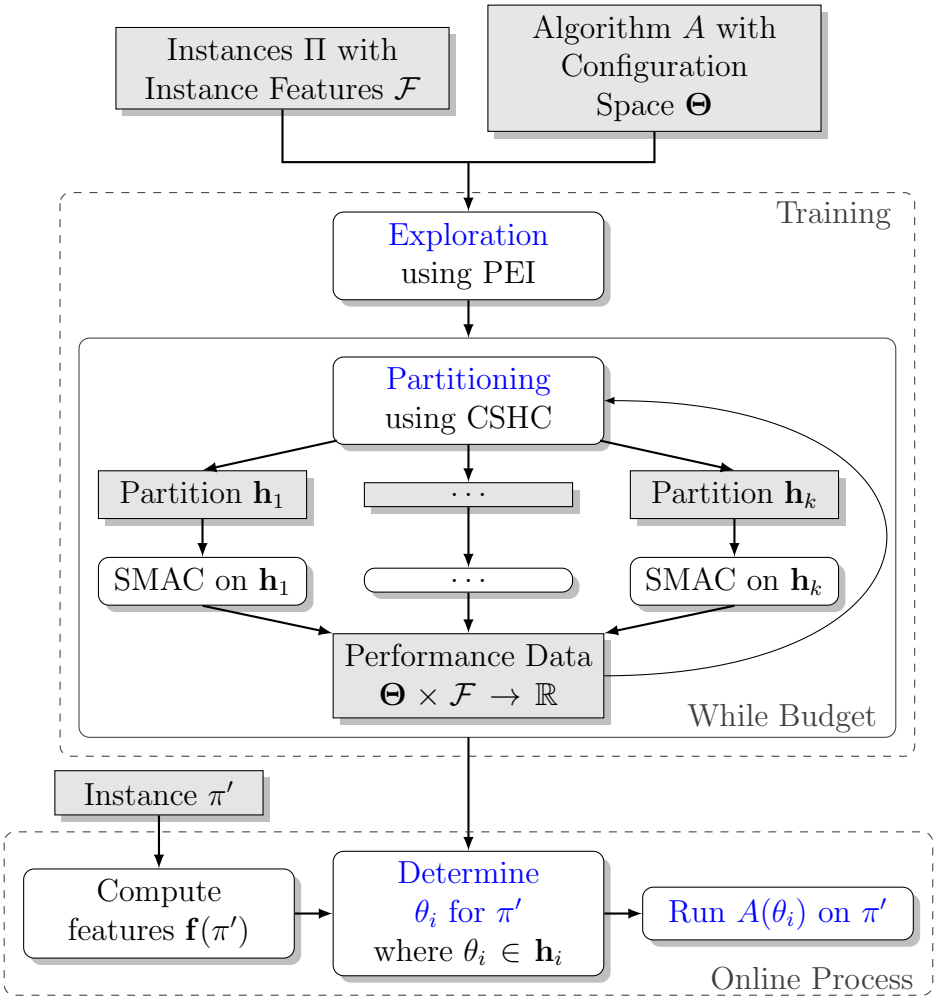


Figure 5.2: *To determine an optimal partitioning of the instance set ISMAC first explores the joint feature/configuration space and uses this information to form an initial partitioning of the instance set using CSHC. On each of the partitions SMAC is run to determine an optimal configuration. The collected performance data is used to refine the partitioning in the next iteration. The partitioning and configuration steps are repeated while a time budget remains. The partitioning found by CSHC forms a decision tree which is used in an online process to determine the best configuration for a before unseen instance.*

5.2 A novel approach

Figure 5.2 shows all aspects of ISMAC. Given a set of heterogeneous instances Π and the corresponding features $\mathcal{F}$ as well as an algorithm A and its configuration space $\mathbf{\Theta}$, ISMAC is faced with two possible options. It can either use an exploration procedure to explore the configuration space on a subset $\Pi' \subseteq \Pi$ of all instances available during the training process, or it skips this exploration and goes directly to the iterative process.

Depending on ISMAC having spend time exploring the configuration space or not, ISMAC constructs an EPM $\hat{m} : \mathbf{\Theta} \times \mathcal{F} \to \mathbb{R}$, trained on all observed algorithm runs, that is used to make predictions about the behavior of the target algorithm on all instances in the training set. If such an initial EPM was learned, it is used to partition the instances into n distinct partitions $\mathbf{H}$. The EPM can be further queried to cheaply find a potential well performing configuration on the individual partitions, which are used to determine the quality of the partitions. A partition might be split further if a new split is found that results in overall better quality of the resulting new partitions. If no previous exploration was performed, random partitions are drawn. An alternative to exploration or random initial partitions would be to use ISACs approach and use g-means clustering to find an initial partitioning. However we did not explore this option.

The partitioning is kept track of in a decision tree. The nodes contain the split criteria and the leafs contain all instances in one partition. The leafs further can be used to store the configurations that should be used to solve instances that belong into that partition.

An algorithm configuration procedure is then used to find an optimized configuration on the subset of instances. Each configuration run is executed in parallel and is given the same budget for configuration. The data of all configuration runs on each partition $\mathbf{h}_i$ is collected and merged together. It is then used to construct a new EPM such that in the subsequent iteration the resulting partitions are based on all the information observed in the previous iterations. This should increase the prediction accuracy of the performance of the target algorithm in different parts of the feature space. ISMAC iterates over these steps until a predefined budget is exhausted. This could either be maximum number of iterations or a time limit.

Similarly to Hydra our approach takes the previous iterations into account when searching for new configurations. Hydra does this by adjusting the performance metric to only consider the better performance of the portfolio and a new candidate configuration. As described in Section 4.3 this complicates the configuration process with each iteration. In our approach however the partitions depend on all the previously seen algorithm runs. The algorithm configuration procedure then is given a set of instances to find a promising configuration. Thus a configuration process does not have to deal with artificially uninformative areas,

as in Hydra.

The iterative nature of our approach also avoids the problems ISAC has to face due to being a one-shot approach. We can incrementally improve the partitions found with ISMAC by learning from observed target algorithm runs. Further, we avoid splitting purely in the feature space. The splits that result with our approach are performed in the feature space but are cost-sensitive and take into account the resulting performance a split might have under a specified configuration. The resulting partitions are thus more homogeneous in the performance space.

For unseen instances, ISMAC determines into which partition the instance belongs by reusing the split criteria determined in the final iteration. The decision tree will first determine into which learned partition an instance belongs and select the configuration that was determined to be the best on that partition, to solve this new instance.

5.3 Exploration

We introduced the exploration phase into ISMAC in order to give the partitioning of ISMAC a head-start. We expect an exploration phase to learn a rough approximation of the target algorithms behavior in different areas of the feature space. The goal is to find some instances that are representative of smaller homogeneous instance sets out of the large heterogeneous set. This would allows us to partition the instances into these homogeneous subset on which a configuration procedure can search for well performing configurations. Instances on which the algorithm behavior was explored could give ISMAC a good idea of which instances will be solved well by the same configuration. Following exploration, ISMACs initial partitioning of instances might thus already be of high quality and only few subsequent iterations would be needed to refine the partitioning.

One problem this exploration mechanism might suffer from is that the chosen random instances are not representative enough of the heterogenity of the whole instance set. This could lead to overexploration of instances that are all solved well by the same set of configurations and leave other parts of the instance set unexplored.

Imagine a set of heterogeneous instances that is constructed from two dissimilar homogeneous sets with one set consisting of 100 instances and the other of 50. Using uniform sampling to choose 50 instances out of the overall heterogeneous set favors the larger set as there are more instances that can be chosen from this set. Thus a good understanding of the behavior of the target algorithm on the larger instance set can be gained but not much will be learned of the smaller set. This might lead ISMAC to start with an odd grouping where only those instances that were explored heavily are packed into a well performing partition and the rest would result in random splits. This might be resolved by applying

a similarity measure to the sampled instances to avoid only looking at a small homogeneous part out of the whole heterogeneous set. After a few iterations however the use of the configurator should gradually fill in the missing information about the unexplored instances. Moreover, the number of instances used for exploration of the algorithm behavior depends on the heterogenity of the instance set. If only a small number of homogeneous subsets is expected to be part of the larger heterogeneous set, the number of instances for exploration could be small. This would avoid exploring on instances where the algorithms behavior is very similar. If however the number of instances is too small, parts of the instance set might not sufficiently be explored.

Another problem is that the exploration phase takes budget away from the overall optimization budget. It is not straight forward to find a good balance between initial exploration and later exploitation of that gathered information. Spending budget exploring only a tiny fraction of the whole instance set might not provide ISMAC with much useful initial information and thus might not improve over random splitting. Spending a lot of budget on exploration might give a good understanding of the instance set in the beginning but after some time might not gain new insights that were not observed already. Thus budget might be unnecessarily wasted without actually providing new valuable information.

Algorithm 2 shows an adaptation of the algorithm presented by Bossek et al. (2015) for which they used *Profile Expected Improvement* (PEI, Ginsbourger et al., 2014) to learn feature parameter mappings. In contrast to that, we use PEI as acquisition function to guide our exploration of a target algorithms behavior on the instance set. We adapt their algorithm such that we not only evaluate the configuration-instance pair with the highest expected PEI. We evaluate each configuration-instance pair, for which the configuration has the highest PEI value for the respective instance. This allows us to explore the algorithms behavior in multiple different areas of the instance set and not only the most promising improving one. When looking at only the most improving configuration for a single instance we might not learn about configurations that are unable to solve certain instances. Learning about which configurations are not well suited for a type of instance later allows us to avoid packing them together with instances that are solved well by the same configuration.

As input, the exploration phase expects a set of instances Π, an algorithm A and its configuration space $\mathbf{\Theta}$ as well as a performance metric m and a budget that determines when to stop exploration. It first draws a subset of instances $\Pi' \subseteq \Pi$ out of all available during training time, on which exploration will occur. The size of this exploration set can be set by the user. For each instance π_i in Π' a random starting configuration is chosen to explore the target algorithms behavior on that instance. Each of these configuration-instance pairs are evaluated by running the target algorithm A with the configuration on the instance. The resulting performance y^{π_i} is measured according to the given metric m. All

Algorithm 2: Exploration using PEI (Adaptation of Bossek et al., 2015)

Data: Instances Π, Configuration Space $\mathbf{\Theta}$, Algorithm A, Performance metric m, Exploration budget B, K #Instances used for exploration
Result: Trained EPM $\hat{m} : \mathbf{\Theta} \times \Pi \to \mathbb{R}$

$S, \Pi' \leftarrow$ sampleKRandomConfigurationInstancePairs($\mathbf{\Theta}$, Π, k)
$\mathcal{X} \leftarrow \emptyset$
$\mathbf{y} \leftarrow \emptyset$
$iteration \leftarrow 0$
while B **do** `// While budget not exhausted`
  if $iteration > 0$ **then**
    $S \leftarrow \emptyset$
    for $\pi_i \in \Pi'$ **do** `// Determine which` θ `to test next on` π_i
      $y^{\pi_i}_{min} \leftarrow \min(\mathbf{y}^{\pi_i})$
      $\theta'_i \leftarrow \text{optimizePEI}(\hat{m}, \pi_i, y^{\pi_i}_{min})$
      $S.\text{append}(\theta'_i, \pi_i)$
  for $\theta_i, \pi_i \in S$ **do** `// Empirically evaluate` θ_i `on` π_i
    $y^{\pi_i} \leftarrow m(A(\theta_i), \pi_i)$
    $\mathcal{X}.append(\theta_i, \pi_i)$
    $\mathbf{y}.\text{append}(y^{\pi_i})$
  $\hat{m} \leftarrow \text{fitModel}(\mathcal{X}, \mathbf{y})$ `// Update model`
return $\hat{m} : \mathbf{\Theta} \times \Pi \to \mathbb{R}$

performance values and the corresponding configuration-instance pairs are then used to construct an EPM.

The process then iterates over all instances in Π'. For each of these the EPM is used to search for a configuration that is deemed to improve over the best performance seen on the instance. The configurations predicted to improve over the current best are used in the subsequent iteration and run on their corresponding instances. Using this exploration strategy, we expect to find complementary configurations that solve different parts of the instance set best. Additionally ISMACs partitioning process does not have to perform any random splitting and can use data of real algorithm runs.

The exploration phase is not interested in finding one global optimum but in the learning about possible high-performance areas, thereby approximating the whole profile of the instance set. For this reason we cannot simply use the expected improvement as acquisition function. Ginsbourger et al. (2014) presented PEI as solution to this problem. They first define the Profile Improvement (PI)

for the k-th experiment out of a series of experiments as

$$PI(\pi, \theta) = \max\{0, M_{\pi,\theta} - \{m_{\min}, \hat{m}(\pi, \theta^*_\pi)\}\} \tag{5.1}$$

where $m_{\min}$ is the minimum observed performance value out of all experiments, $\hat{m}(\pi, \theta^*_\pi)$ is the estimated cost for the estimated best configuration on a given instance π and $M_{\pi,\theta}$ is the random variable over the posterior, defined by the EPM. In case of a gaussian process or a random forest, $M_{\pi_{\pi,\theta}}$ is normal distributed as $\mathcal{N}(\mu_{\pi,\theta}, \sigma^2_{\pi,\theta})$, with a mean prediction $\mu_{\pi,\theta}$ and the uncertainty $\sigma^2_{\pi,\theta}$. This allows them to define PEI as $PEI(\pi, \theta) = \mathbb{E}(PI(\pi, \theta))$. This gives the closed-form definition:

$$PEI(\pi, \theta) = \begin{cases} \sigma_{\pi,\theta}\left[(g(\pi,\theta)\Phi g(\pi,\theta)) + \phi g(\pi,\theta)\right], & \text{if} \quad \sigma_{\pi,\theta} \neq 0 \\ 0, & \text{else} \end{cases} \tag{5.2}$$

ϕ and Φ are the probability density and cumulative density of the standard normal distribution and $g(\pi, \theta) = \frac{m_{\min} - \mu_{\pi,\theta}}{\sigma_{\pi,\theta}}$.

5.4 Cost-Sensitive Partitioning

We already stated that we think it necessary to take into account the target algorithms performance when splitting the instances into new partitions. We thus decided to make use of *Cost Sensitive Hierarchical Clustering* (CSHC, Malitsky et al., 2013). As the name suggests CSHC is aware of costs during the partitioning process and, similarly to g-means clustering it is hierarchical. The number of resulting partitions is not predetermined and each partition might be split further, if the process determines it to result in lower costs.

Algorithm 3 shows a modified version of CSHC such that it splits an instance with respect to the performance loss of grouping them together and letting them be solved by a configuration that is predicted to perform well on the smaller set. It accepts an upper bound of the number of resulting partitions. As input it expects an instance set Π, a configuration space $\boldsymbol{\Theta}$ from which it can sample configurations and an empirical performance model $\hat{m}$ that is able to predict the performance value of the target algorithm for a given configuration on a specific instance. It is further given a maximum of allowed partitions k. CSHC starts with one partition in which all instances are packed. For each partition in the current set of partitions, CSHC draws R random splits that have to consist of at least n instances per resulting partition. In our experiments, we set R to 1 000, to not spend to much of the allocated budget on finding the splits. These splits might either be axis-aligned or non-axis-aligned. In the case of axis-aligned splits, a feature vector $\mathbf{f}$ of an instance in the partition is drawn at random. The value of a random feature in the vector is then used to determine the split. Instance-feature vectors whose feature value is greater than the split

Algorithm 3: Modified CSHC (Malitsky et al., 2013) for PIAC

Data: Instances Π, Configuration Space $\mathbf{\Theta}$, $\hat{m} : \mathbf{\Theta} \times \Pi \to \mathbb{R}$, Minimal partition size n, Maximum number of partitions k
Result: Partitioning $\mathbf{H}$ of Π

$l(\Pi, \theta) \coloneqq \sum_{\pi \in \Pi} \hat{m}(\theta, \pi) - \min_{\theta' \in \mathbf{\Theta}} \hat{m}(\theta', \pi)$
$\mathbf{H}' \leftarrow \Pi$
$\mathbf{H} \leftarrow \emptyset$
while $\mathbf{H} \neq \mathbf{H}'$ **do** `// while not converged`
  $\mathbf{H} \leftarrow \mathbf{H}'$
  $\mathbf{H}' \leftarrow \emptyset$
  foreach $\mathbf{h} \in \mathbf{H}$ **do**
    $\theta_{\mathbf{h}} \leftarrow \arg\min_{\theta \in \mathbf{\Theta}} \sum_{\pi \in \mathbf{h}} m(\theta, \pi)$
    $bestSplit \leftarrow \mathbf{h}$
    $bestLoss \leftarrow l(\mathbf{h}, \theta_{\mathbf{h}})$
    for $r \in \{1, 2, \ldots, R\}$ **do** `// Draw` R `random splits`
      $\mathbf{h_1}, \mathbf{h_2} \leftarrow \text{findRandomSplit}(\mathbf{h}, n)$
      $\theta_{\mathbf{h_1}} \leftarrow \arg\min_{\theta \in \mathbf{\Theta}} \sum_{\pi \in \mathbf{h_1}} m(\theta, \pi)$
      $\theta_{\mathbf{h_2}} \leftarrow \arg\min_{\theta \in \mathbf{\Theta}} \sum_{\pi \in \mathbf{h_2}} m(\theta, \pi)$
      if $(l(\mathbf{h_1}, \theta_{\mathbf{h_1}}) + l(\mathbf{h_2}, \theta_{\mathbf{h_2}})) \leq bestLoss$ **then**
        $bestSplit \leftarrow (\mathbf{h_1}, \mathbf{h_2})$
        $bestLoss \leftarrow l(\mathbf{h_1}, \theta_{\mathbf{h_1}}) + l(\mathbf{h_2}, \theta_{\mathbf{h_2}})$
    $\mathbf{H}' \cup \{bestSplit[0], bestSplit[1]\}$
    if $|\mathbf{H}'| < k$ **then**
      break
return H

value are packed into one partition, the rest in the other. Non-axis-aligned splits are generated as follows. The mean over all feature vectors in the partition is computed. This vector is then perturbed by a vector whose values are drawn at random from a normal distribution. This is used as support vector $\mathbf{f}_{sup}$. Another normal distributed vector is used as direction vector $\mathbf{f}_{dir}$. We compute into which partition an instance π_i belongs as follows, $(\mathbf{f}_{\pi_i} - \mathbf{f}_{sup}) \bullet \mathbf{f}_{dir} > 0$, where $\bullet$ is the dot-product. It is also possible to reduce the size of the vectors beforehand, by using *principle component analysis* (PCA, Pearson, 1901). As it is more unlikely to find a good random split in a high dimensional space, using PCA would reduce the dimensionality and increase the likelihood of finding a good random split.

In order to be cost-sensitive during this procedure, CSHC uses a loss function to determine if a found partition $\mathbf{h}$ is homogeneous or not.

$$l(\mathbf{h}, \theta_{\mathbf{h}}) \coloneqq \sum_{\pi \in \mathbf{h}} m(\theta_{\mathbf{h}}, \pi) - \min_{\theta' \in \mathbf{\Theta}} m(\theta', \pi) \tag{5.3}$$

where $\theta_{\mathbf{h}} = \arg\min_{\theta \in \mathbf{\Theta}} \sum_{\pi \in \mathbf{h}} m(\theta, \pi)$. The used loss is the performance difference of using a well performing configuration on a found partition and using a best configuration for each instance, individually. To determine well performing configurations we can use local search. Nevertheless, using the loss as presented would require real algorithm runs. This would make the method too expensive to be used in practice. We can make use of an EPM to predict the target algorithms performance and thus compute the loss inexpensively.

$$l(\mathbf{h}, \theta_{\mathbf{h}}) \coloneqq \sum_{\pi \in \mathbf{h}} \hat{m}(\theta_{\mathbf{h}}, \pi) - \min_{\theta' \in \mathbf{\Theta}} \hat{m}(\theta', \pi) \tag{5.4}$$

The modified loss is the predicted performance difference of applying the best found configuration of the partition to applying an individual best configuration to each instance in the partition. This loss thus reflects if a partition of instances is solved well by a given configuration or not.

Using the EPM we determine a well performing configuration on the original split $\theta_{\mathbf{h}}$ as well as configurations on the resulting two splits $\theta_{\mathbf{h_i}}$. To decide if a split is kept or not, the loss of the individual splits is summed and compared to the loss of the original partition. If the sum of losses is smaller, a split is deemed to be better than the original partition and the partition is split into the two smaller ones. If not, the original partition is kept.

Using local search on an EPM to search the best configuration θ on partition $\mathbf{h}$ could lead to problems. It might be possible that the EPM that was learned is not suited well for predicting the performance in unexplored of the configuration space. It might thus happen that through local search on the EPM, one configuration is predicted to be very well performing on the set of instances but in reality does not solve the instances well at all. This can lead to problems as these configurations that are predicted to solve a partition best are used as starting points for the configuration step. Further all actual algorithm runs that were performed on instances in a partition are also given to the configurator as starting information. This might lead the configurator along a wrong path in the search for a better performing configuration and might actually lead to worsening performance in subsequent steps. Therefore we propose a second variant that only predicts the performance for configurations that were actually evaluated during the configuration step on at least one instance. This decreases the uncertainty in the predictions of the target algorithms behavior, given the set of instances and the configuration that should be used to solve it.
CSHC is run until the allowed maximum number of partitions is reached or if no new split is found that results in a new and better partitioning. Since the

splitting results in two subsets of instances, the partitions can be stored in a tree structure where each node stores the corresponding splitting criteria and the leaf values store the partition and the best configuration found on that partition.

5.5 Regularization

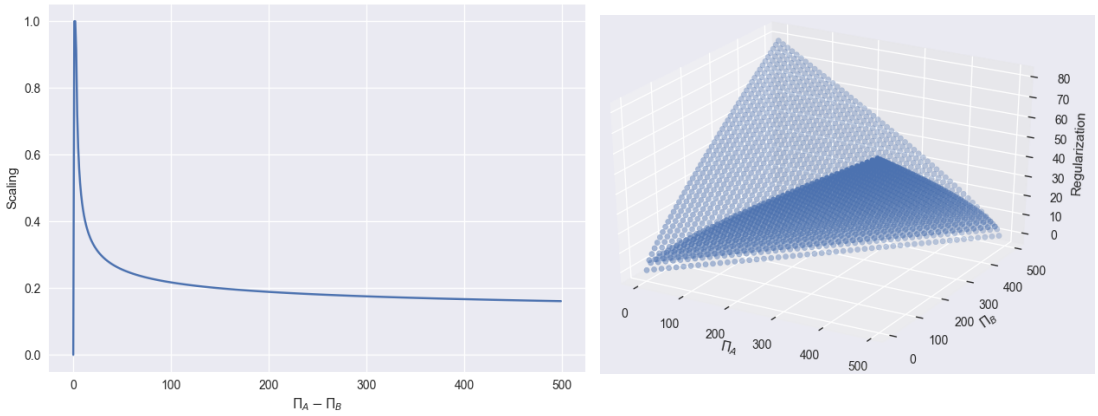
In preliminary results we observed that very imbalanced splits could occur where instances that were easy to solve were grouped into small partitions leaving the hard to solve instances in a large partition. The configuration procedure was able to find well performing configurations on those small partitions but failed to find a configuration that was able to tackle the last partition completely. We thus want to use regularization to avoid the problem of grouping only hard instances together and force ISMAC to pack them together with easier instances. These might be solved best by the same configuration, even though the individual performances differ due to the instance hardness.

When running ISMAC, a user is free to set the minimal number of instances that are required to form a partition. This can influence how many partitions CSHC can find. We observed that for low values, ISMAC will likely end up with a lot of partitions, if it does not find a clear split that prefers larger partitions. In later iterations, ISMAC gathered enough information about the behavior of the target algorithm on all types of instances in the set and might start to find larger groupings that result in an overall better partitioning. Similarly when allowing only relative large partitions, ISMAC will start with a very small number of partitions and in later iterations might be able to split these larger partitions into smaller ones, as long as the specified minimum allowed number of instances is not violated. The loss that is computed in Algorithm 3, line 15, can be computed on partitions of vastly different sizes if the number of required instances is set small. An alternative is to keep the partitions in each split balanced. We therefore introduced a regularization term that prefers balanced splits.

$$l(\mathbf{h_1}^*, \theta_{\mathbf{h_1}^*}) + l(\mathbf{h_2}^*, \theta_{\mathbf{h_2}^*}) + Reg \tag{5.5}$$

$$Reg \coloneqq |\#\Pi_i - \#\Pi_j| \cdot \min\left(1, \frac{1}{\ln\left(|\#\Pi_i - \#\Pi_j|\right)}\right) \tag{5.6}$$

Reg is the regularization term. It takes into account how imbalanced the partitions are and scales the value, such that the overall equation is not dominated by this term. Without the scaling the regularization term would quickly be the only factor used in finding good splits. With the scaling, the regularization will prefer balanced splits over imbalanced ones, only if they have similar losses. Figure 5.3 shows the scaling and resulting regularization values for an artificial example. The scaling does not alter the profile of the absolute function. However, it keeps the values low, to prevent the domination of the regularization.



(a) (b)

Figure 5.3: *Example of the regularization that can be used with ISMAC. (a) shows the scaling that is used. In this artificial example, both instance sets can have up to 500 instances. If the found split is not very imbalanced, the penalization is not that large and can have full effect. However if the imbalancedness between the sets gets large, the difference is scaled down much more. (b) shows the corresponding regularization values.*

5.6 Parallel Execution

ISMAC users can further specify how many partitions should maximally be created. Users could for example set this to the number of cores on their machine if no prior knowledge about the heterogenity of the instance set is given. Thus ISMAC might find any number of partitioning in the range of $(1, \#Cores)$. If no more than k' cores are needed, but k are allocated, ISMAC will never force a more fine-grained splitting. Setting this parameter together with the minimal number of instances per partition can result in different partitions, especially in earlier iterations when ISMAC has not seen many real algorithm runs. The configuration procedure will then be run in parallel on all available cores. If more cores are allocated then needed in a parallel environment, ISMAC leaves these cores idle.

5.7 Configuration and Budget Policy

We decided to use *SMAC* (Hutter et al., 2010), as it is able to search for promising configurations that consist not only of categorical but also of continuous parameters and was demonstrated to achieve great performance gains. Such an algorithm configuration procedures budget determines how long it is able to search for well performing configurations. Larger budgets allow to explore more configurations and can give a better estimate of how well a configuration will solve an instance set. Simillarly, ISMACs behavior depends on how a budget is allocated to find possible splits and optimized configurations. A user can also specify the configur-

ation budget. This budget is separate from the overall budget and only specifies how much budget is spend on the individual configuration passes. If this configuration budget is small ISMAC will run for more iterations in which it will look at more possible partitions. These will however be based on few observations where *SMAC* might not have been able to look in depth at possible high-performance areas. If relatively large chunks out of the whole budget are allocated for only the optimization runs, ISMAC might still be stuck with a suboptimal partitioning that makes the overall configuration on the partitions only slightly better and the final partition might not be as good as it could have been with a more balanced optimization budget. Instead of allocating equal budgets for splitting and configuration, ISMAC can be run with an incrementally increasing budget. In the first short iterations ISMAC can gather data that will allow for a better partitioning of the instances. This can be seen as an implicit online exploration at the beginning stages to become more certain of what makes a good partitioning. In later stages, when the most recent partitioning should be more robust, ISMAC is able to spend more time in searching for well performing configurations. This allows the configuration procedure to find better configurations due to a more exhaustive search. The final iteration will thus have formed a good partitioning based on all the information that was quickly collected in the beginning and the found configurations will perform well their respective partitions. The iterative budget is calculated as $b \cdot \left(2^{s \cdot (i-1)} - (1 - s)\right)$, where b is the configuration budget used for equal splits, i is the iteration number and s is the starting factor. If ISMAC would be allowed a budget of one hour with equal splits and the starting factor would be set to 0.25, the first iterative budget would only be 15 minutes long and the second $\approx$ 26 minutes long.

5.8 Summary

In this chapter we motivated the necessity to reason about the behavior in the performance space and not only in the feature space when partitioning a set of instances into smaller distinct sets. We presented ISMAC, a novel *per instance algorithm configuration* approach, that takes into account not only similarities in the feature space but also the performance values for potential configurations. We presented possible settings of the system and explored the theoretical implications of different settings of ISMAC.

Chapter 6

Experiments

In this chapter, we want to answer the following research questions. First we are interested in finding out how the parameters of ISMAC should be set to find good partitions of instances. Secondly we want to know how well ISMAC performs in comparison to the current state-of-the-art PIAC systems. We lastly want to determine how ISMAC behaves across different sets of instances.

6.1 Experimental Setup

All experiments were executed on a compute cluster equipped with two Intel Xeon E5-2650v2 8-core CPUs, 20 MB L2 cache and 64 GB of (shared) RAM per node, running Ubuntu 16.04.02 LTS 64 bit.

We implemented ISAC, Hydra and ISMAC using Python3.5[1] and all methods use the python implementation of *SMAC* (v3)[2] to optimize the target algorithm on the (sub)set(s) of instances.

To evaluate all the presented PIAC systems we followed Xu et al. (2010) and configured the solver *SATenstein* (KhudaBukhsh et al., 2009), with a runtime-cutoff of 5 seconds.

SATensteins behavior is controlled by a total of 41 parameters which span a discretized configuration space consisting of 4.82×10^{12} possible configurations. SATenstein was inspired by a wide range of state-of-the-art stochastic local search SAT-solvers and incorporates different components of these, leading to this large parameter space.

We evaluate ISMAC's capabilities in optimizing SATenstein on four different, heterogeneous instance sets and compare the optimized performance against performances obtained using ISAC, Hydra, standalone *SMAC* and the default configuration. The instance sets were constructed by Xu et al. (2010) to evaluate

[1] `https://www.python.org/download/releases/3.5.2/`

[2] `https://github.com/automl/SMAC3/tree/v0.5.0`

Hydras performance. We used the instance sets as given in AClib (Hutter et al., 2014b).

Table 6.1 shows the number of instances for the used sets and the training-test split. All methods were run for 10 hours. This means that standalone *SMAC* and each ISAC partition had 10 full hours to optimize over the given instance set.

Set Π	#Inst.	#Feats
HAND	380/190	124
INDU	250/250	66
RAND	897/448	124
BM	1527/888	66

Table 6.1: *Training-Test split for the used instance sets, as well as the number of features in each set.*

The iterative nature of ISMAC and Hydra demanded to split the budget over multiple runs. We allowed Hydra to run for 10 whole iterations where each optimization step had one hour to solve the task and neglected the overhead that was caused due to evaluating the new found configuration on the whole training set. Following Xu et al. (2011), we allowed Hydra to search for optimal configurations using n-parallel runs. We set n to 8, as this was the maximal number of partitions ISMAC was allowed to find in our experiments. Our Hydra implementation only added one configuration in each iteration to its portfolio. It used the configuration procedures internal estimate of the cost for a configuration to select which out of the n configurations to add.

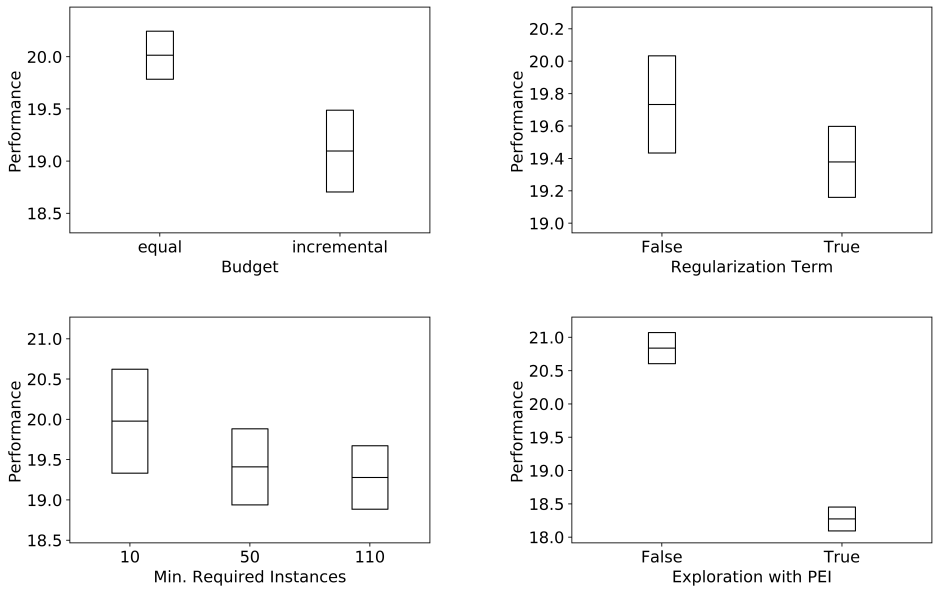


Figure 6.1: *fANOVA results for ISMAC-LS. The upper left shows the result for the different budget strategies. The top right image shows the result for using our regularization method. The bottom left shows the result for our second regularization method and the last image shows the influence of the exploration phase.*

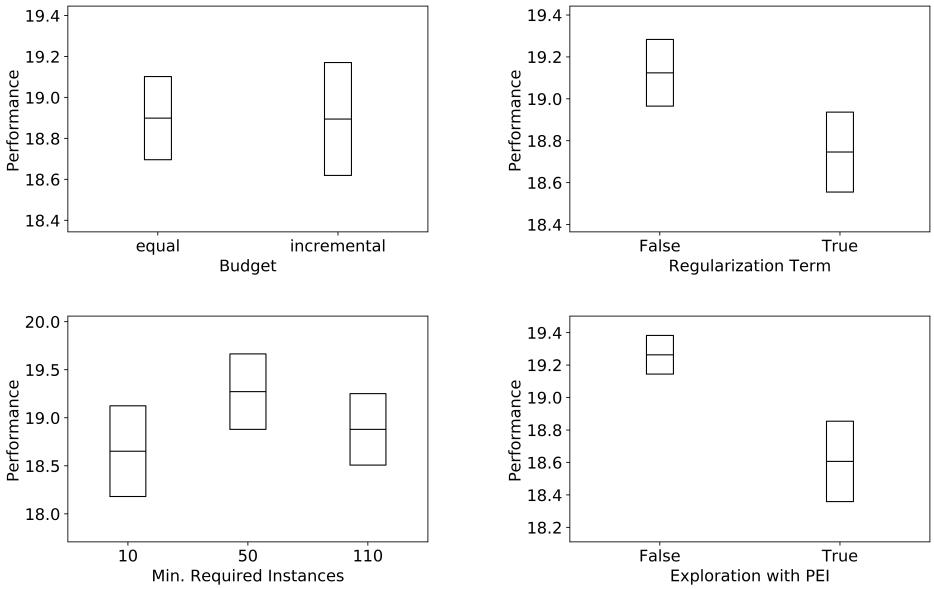


Figure 6.2: *fANOVA results for ISMAC-KC. The upper left shows the result for the different budget strategies. The top right image shows the result for using our regularization method. The bottom left shows the result for our second regularization method and the last image shows the influence of the exploration phase.*

6.2 Parameter Importance

In a pre-evaluation of ISMAC we tested different parameter settings and used fANOVA (Hutter et al., 2014a) to determine promising configurations for ISMAC. We refer to the ISMAC variant using local search during the partitioning as ISMAC-LS and to the variant that only uses configurations that were run on at least one instance as ISMAC-KC. Figures 6.1 and 6.2 show the fANOVA results we obtained for ISMAC-LS and ISMAC-KC respectively. We can see from the graphics that both variants benefit from the exploration phase as well as the regularization that prefers balanced splits. Using an incremental budget benefits ISMAC-LS more than ISMAC-KC. When looking at the number of instances required to form a partition, ISMAC-LS and ISMAC-KC do not seem to favor one clear setting. We used these insights to evaluate ISMAC-LS and ISMAC-KC each for two configurations. Both variants were set to explore the instance sets on 50 instances for one hour, favored balanced splits and used an incremental budget. We evaluated ISMAC-KC with a minimum number of required instances in a partition set to 10 and 110 and ISMAC-LS with a minimum number of instances set to 50 and 110. We report our final results using ISMAC-KC, constrained to find partitions that contain at least 110 instances, as it proved to be the most robust setting in our experiments.

ISMAC configurations on TRAIN					
		LS-50	LS-110	KC-10	KC-110
HAND	PAR10	21.9	23.3	21.9	21.8
	VBS-P	21.8	22.3	20.0	20.5
	Timeouts	164	175	164	164
	VBS-T	156	168	150	154
INDU	PAR10	17.5	18.1	15.9	15.7
	VBS-P	14.6	17.8	5.8	4.6
	Timeouts	83	87	76	75
	VBS-T	68	85	24	18
RAND	PAR10	14.3	13.7	14.1	13.5
	VBS-P	11.3	11.9	12.4	12.1
	Timeouts	251	241	248	238
	VBS-T	198	210	220	213
BM	PAR10	18.3	17.6	19.1	17.9
	VBS-P	14.3	13.5	14.3	13.7
	Timeouts	546	529	574	536
	VBS-T	426	402	424	408

Table 6.2: *Results of different ISMAC configurations on all training sets. PAR10 is the penalized average runtime, with a penalization factor of 10. VBS-P is the virtual best solvers PAR10 and VBS-T the virtual best timeouts. The given values are the median values out of 3 repetitions. LS uses local search to find new promising configurations, whereas KC only considers configurations during partitioning, that were evaluated on at least one instance. The number coming after is the minimal required number of instances per partition.*

Table 6.2 shows the comparison of all four configurations. All values listed in the table are the median values out of 3 runs each training instance set. We observe that on each dataset, there are two well performing configurations. For example on `INDU` both ISMAC-KC configurations outperform both ISMAC-LS settings, whereas on `BM` both settings that require at least 110 instances per partition outperform the other two. However ISMAC-KC with 110 instances per partition is the only configuration that works well on all instance sets.

6.3 Empirical Evaluation

The iterative budget is calculated as $b \cdot \left(2^{s \cdot (i-1)} - (1-s)\right)$, where b is the configuration budget used for equal splits, i is the iteration number and s is the starting factor. In our experiments b is set to one hour and s to 0.25. This gives ISMAC a starting configuration budget of 15 minutes. In the second iteration ISMAC spends $\approx$ 26 minutes on configurations. This setting gives ISMAC a few short initial configuration passes and increasingly shifts the focus from the partitioning procedure to finding good configurations on the found partitions.

All values reported in the result tables, are the median result out of three repetitions. *Def* shows the result for the default configuration of SATenstein, and SMAC the result for the best configuration found by SMAC with a time budget of 10 hours. ISAC lists the result for the portfolio found by ISAC. Hydra shows the result of using Hydra for 10 iterations, where each iteration was given a budget of 1 hour, thus creating a portfolio of 10 configurations. In column ISMAC we list the result obtained with our new method. ALL is the virtual best solver (VBS) using a portfolio of all configurations found by all methods and using a theoretical perfect selector. PAR10 shows the penalized average runtime, where PARX penalizes each timed-out or crashed run as X times the running time cutoff. VBS-P shows the virtual best performance of each methods found portfolio, when using a perfect selector. Timeouts shows the number of timeouts (cutoff after 5 seconds) each learned portfolio-selector pair generated. VBS-T shows the virtual best timeouts the portfolio would result in, if we could use a perfect selector.

The validation results on the handcrafted (`HAND`) dataset are listed in Table 6.3. On this dataset g-means determined the number of partitions to be 3, giving ISAC a portfolio of 3 configurations. ISMAC's CSHC procedure described in the previous section also found 3 partitions. On TRAIN and TEST we can observe that using a standalone configuration procedure already improves quite well over the default. Using only the default configuration would result in 213 timeouts on TRAIN and 100 on TEST. Using an optimized configuration found by SMAC only results in 182 and 86 timeouts on TRAIN and TEST, respectively. However we can also observe that we can further improve by using an instance-aware approach. Comparing the number of timeouts Hydras and ISMAC's portfolio cause on TRAIN, we see that both can further reduce the number of timeouts by nearly 20. Comparing Hydras and ISMAC's virtual best timeouts we can see that Hydra found a better portfolio. When looking at the number of timeouts of ALL, we see that Hydra solves 5 instances less. Looking at TEST, we can see that the difference between instance-aware and a stand-alone configuration procedure is not as large.

A representation of how ISMAC's result evolved over time is presented in Figure 6.3. The dashed lines are a visual aid that show the number of timeouts found

		HAND					
		Def	SMAC	ISAC	Hydra	ISMAC	ALL
TRAIN	PAR10	28.2	24.1	23.9	21.4	21.8	18.6
	VBS-P	-	-	21.0	19.3	20.5	18.6
	Timeouts	213	182	180	161	164	140
	VBS-T	-	-	158	145	154	140
TEST	PAR10	26.5	22.9	22.9	21.0	21.9	18.4
	VBS-P	-	-	19.8	19.4	20.0	18.4
	Timeouts	100	86	86	79	82	69
	VBS-T	-	-	74	73	75	69

Table 6.3: *Results obtained on the Handcrafted dataset. PAR10 is the penalized average runtime, with a penalization factor of 10. VBS-P is the virtual best solvers PAR10 and VBS-T the virtual best timeouts. The given values are the median values out of 3 repetitions. ALL is the portfolio, consisting of all configurations found by the presented methods, using a perfect selector.*

by the other methods after spending the whole budget. Hydras any-time performance is not captured by this graph. After iteration 1, ISMAC already spent one hour of its budget on exploration and 15 minutes for the configuration process. After two more iterations ISMAC's portfolio approaches the performance reached by ISAC and SMAC after spending the whole budget. In the subsequent iteration a bad partition is found, slightly increasing the number of timeouts. After spending the whole budget, ISMAC approaches the same number of timeouts as Hydra.

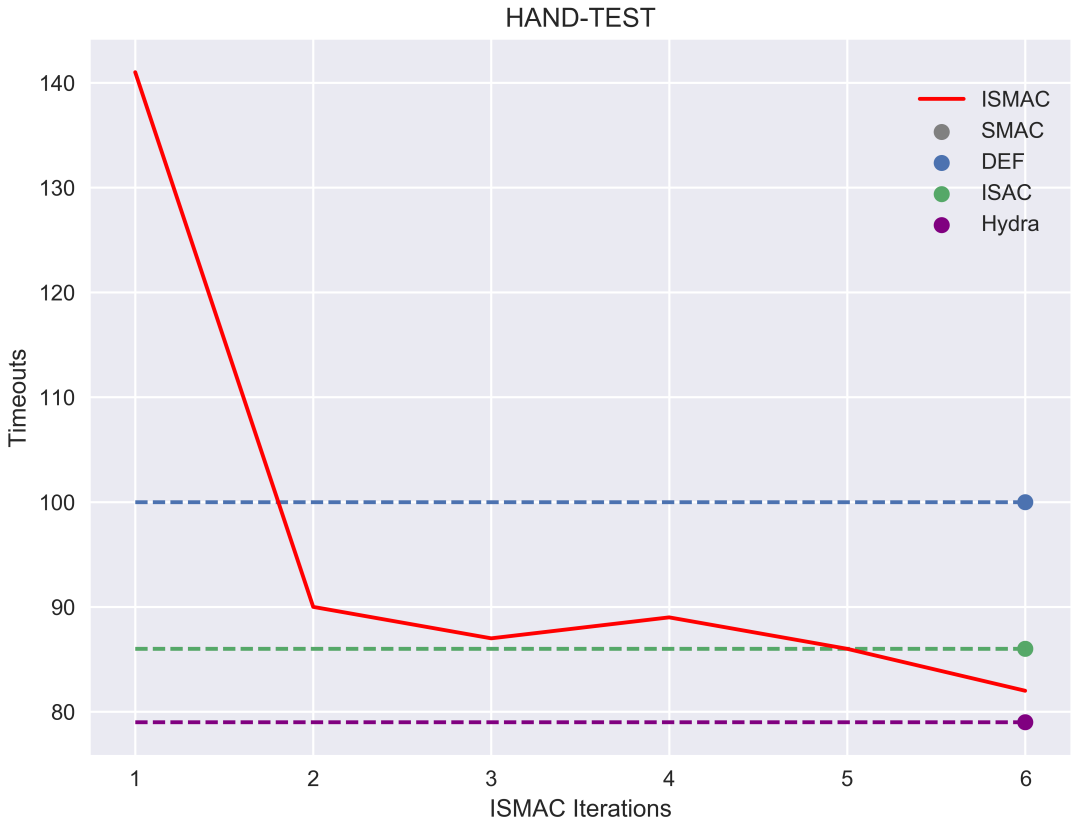


Figure 6.3: *ISMAC validation result on* ***HAND-TEST****. On the x-axis the ISMAC iterations are shown. After iteration 1, ISMAC has already spent one hour of its budget on exploration. The budget spent between iterations is not equal but iteratively increased. The markers on the right side of the figure are the number of timeouts reached by the other systems. The dotted lines leading to the markers on the right are meant as aide to better compare ISMAC iteration values to the other methods. SMAC and ISAC result in the same number of timeouts, thus the SMAC line lies beneath the ISAC line.*

		INDU					
		Def	SMAC	ISAC	Hydra	ISMAC	ALL
TRAIN	PAR10	38.0	20.2	21.7	21.9	15.7	3.1
	VBS-P	-	-	8.2	9.1	4.6	3.1
	Timeouts	187	97	105	106	75	11
	VBS-T	-	-	36	41	18	11
TEST	PAR10	38.8	24.5	24.1	23.0	18.3	2.8
	VBS-P	-	-	11.7	8.9	18.0	2.8
	Timeouts	191	120	118	112	88	9
	VBS-T	-	-	53	39	82	9

Table 6.4: *Results obtained on the Industrial dataset. PAR10 is the penalized average runtime, with a penalization factor of 10. VBS-P is the virtual best solvers PAR10 and VBS-T the virtual best timeouts. The given values are the median values out of 3 repetitions. ALL is the portfolio, consisting of all configurations found by the presented methods, using a perfect selector.*

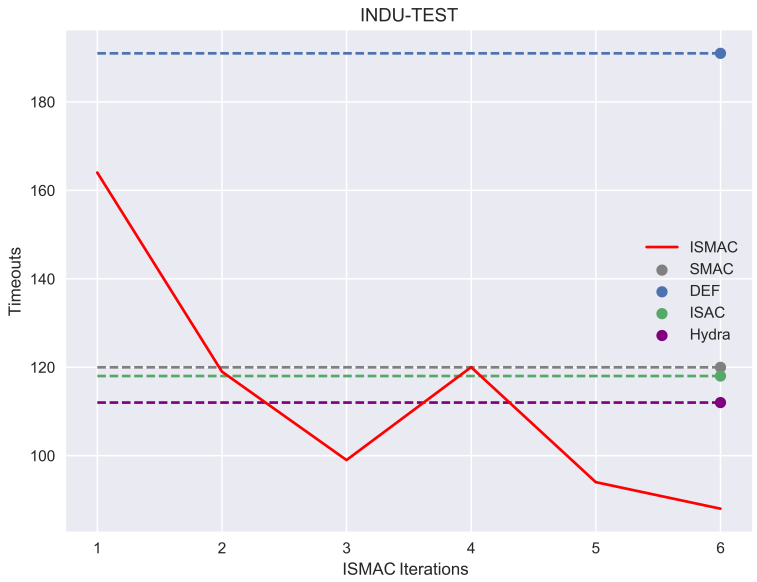


Figure 6.4: *ISMAC validation result on INDU-TEST. On the x-axis the ISMAC iterations are shown. After iteration 1, ISMAC has already spent one hour of its budget on exploration. The budget spent between iterations is not equal but iteratively increased. The markers on the right side of the figure are the number of timeouts reached by the other systems. The dotted lines leading to the markers on the right are meant as aide to better compare ISMAC iteration values to the other methods.*

The validation results on the industrial (INDU) dataset are listed in Table 6.4. On this dataset ISAC found a portfolio consisting of 6 configurations. CSHC found two partitions which are solved best by two different configurations. We observe that ISMAC outperforms all other methods on both TRAIN and TEST. It constructs the best portfolio and the learned selector is capable of selecting well performing configurations. On TEST however, Hydras VBS is more capable of solving instances. The pairwise weighted random forests however are not able to make use of the portfolio and sometimes select a wrong configuration. All methods combined are capable of nearly solving all instances in TRAIN and TEST. This suggests that all methods combined find complementary configurations. Figure 6.4 shows how ISMAC's solution evolved over time on the industrial test set. The partitioning found in iteration 4 reduces the number of timeouts on TRAIN but temporarily increases the timeouts on TEST.

Table 6.5 shows the results obtained on the random dataset. On this instance set, ISAC found 5 partitions and ISMAC 6. Again Hydra and ISMAC have similar PAR10 and timeout values, both when using their learned selectors and when comparing their portfolios using a virtual best solver. The larger number of instances in this dataset required that a bigger chunk of the budget was spent on finding good partitions. This allowed ISMAC to run for only 4 iterations. This problem could be avoided with a CSHC implementation that is capable of partitioning the instance sets in parallel. On this set ISAC is not able to find good partitions and does not improve over the instance-unaware SMAC. Figure 6.5 shows ISMAC's improvement over time on TEST. In iteration 3 a partition is found that is slightly worse on both TRAIN and TEST than the one previously found. However, due to the iterative nature, ISMAC is able to correct the partitioning on TRAIN and further improve over the previous best. This refined partitioning also improves the result on TEST.

Table 6.6 shows the result on a mix of all datasets. On this dataset ISAC found only 3 partitions and ISMAC found 8 partitions. On TRAIN Hydra outperforms ISMAC by 15 timeouts, however on TEST they have similar performance when using their learned selectors. The virtual best solver however shows that Hydra's portfolio is theoretically capable of solving more instances but again the pairwise weighted random forests are not able of perfectly selecting the configurations. The portfolio consisting of all configurations that were found by all methods again suggests that the configurations found by differing PIAC systems are complementary. Figure 6.6 shows ISMAC's performance over time on the test set of this benchmark. ISMAC starts with a poor partition but quickly improves. After just 3 partitions ISMAC outperforms Hydra. In the next iteration however it finds a partitioning that is slightly worse.

RAND							
		Def	SMAC	ISAC	Hydra	ISMAC	ALL
TRAIN	PAR10	17.8	14.7	14.8	13.7	13.5	10.9
	VBS-P	-	-	12.5	11.8	12.1	10.9
	Timeouts	316	260	261	241	238	192
	VBS-T	-	-	221	208	213	192
TEST	PAR10	16.5	13.7	14.1	13.1	13.4	10.6
	VBS-P	-	-	11.9	11.8	11.7	10.6
	Timeouts	146	121	124	116	118	93
	VBS-T	-	-	105	104	103	93

Table 6.5: *Results obtained on the Random dataset. PAR10 is the penalized average runtime, with a penalization factor of 10. VBS-P is the virtual best solvers PAR10 and VBS-T the virtual best timeouts. The given values are the median values out of 3 repetitions. ALL is the portfolio, consisting of all configurations found by the presented methods, using a perfect selector.*

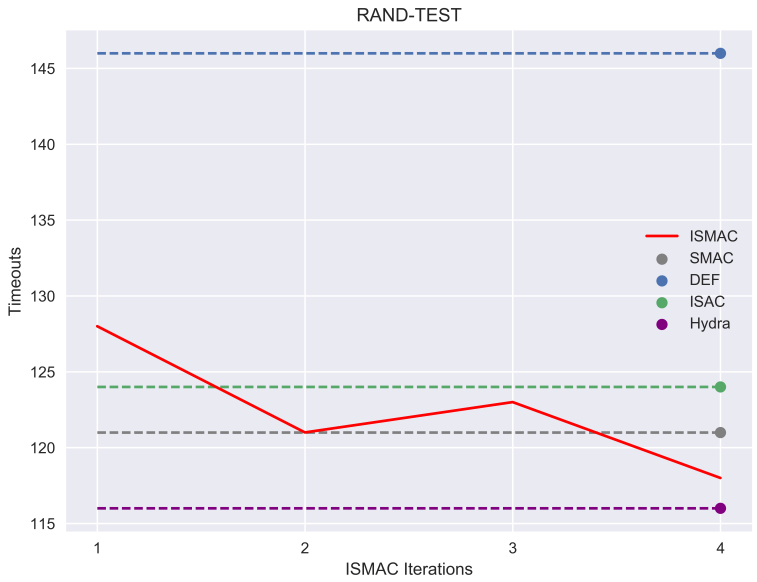


Figure 6.5: *ISMAC validation result on* ***RAND-TEST****. On the x-axis, ISMAC's iterations are shown. After iteration 1, ISMAC has already spent one hour of its budget on exploration. The budget spent between iterations is not equal but iteratively increased. The markers on the right side of the figure are the number of timeouts reached by the other systems. The dotted lines leading to the markers on the right are meant as aide to better compare ISMAC iteration values to the other methods.*

		BM					
		Def	SMAC	ISAC	Hydra	ISMAC	ALL
TRAIN	PAR10	23.7	18.9	18.9	17.3	17.9	12.2
	VBS-P	-	-	16.7	13.5	13.7	12.2
	Timeouts	715	569	569	521	536	363
	VBS-T	-	-	500	404	408	363
TEST	PAR10	24.7	19.9	19.8	18.0	17.9	11.3
	VBS-P	-	-	16.3	13.4	14.2	11.3
	Timeouts	434	348	346	314	312	193
	VBS-T	-	-	283	231	245	193

Table 6.6: *Results obtained on the BigMix dataset. PAR10 is the penalized average runtime, with a penalization factor of 10. VBS-P is the virtual best solvers PAR10 and VBS-T the virtual best timeouts. The given values are the median values out of 3 repetitions. ALL is the portfolio, consisting of all configurations found by the presented methods, using a perfect selector.*

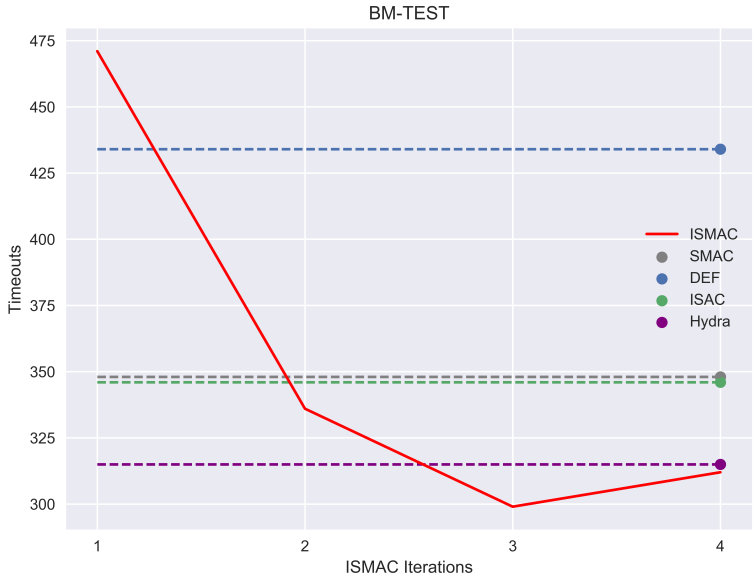


Figure 6.6: *ISMAC validation result on `BM-TEST`. On the x-axis, ISMAC's iterations are shown. After iteration 1, ISMAC has already spent one hour of its budget on exploration. The budget spent between iterations is not equal but iteratively increased. The markers on the right side of the figure are the number of timeouts reached by the other systems. The dotted lines leading to the markers on the right are meant as aide to better compare ISMAC iteration values to the other methods.*

Chapter 7

Conclusion

In this Thesis we first discussed the current state-of-the art of per instance algorithm configuration. We presented three different established systems and discussed parts of these systems that we think might either be beneficial in solving heterogeneous instances set or a hindrance. We then used the insights we gained to build our own PIAC system. We introduced the idea of an exploration phase into the PIAC work-flow. This allows our system to find promising starting points for the subsequent iterative process. In the iterative process we keep track of how well a set of instances can be solved by a given configuration. We use this knowledge to determine if a found partition will improve the overall performance or not. We discussed our reasoning for why our PIAC system splits the instances with respect to possible performance of a configuration. The iterative nature of our system further allows it to refine the found partitions. We finally presented the results of our empirical evaluation, where we compared our new approach to the current state-of-the-art. From the presented results we can see that our approach is a capable PIAC system that can handle different heterogeneous instance sets. Our results further support our claim that a cost-sensitive approach is better than a purely feature based approach, as both Hydra and ISMAC outperform ISAC. Further ISMAC reached similar performance to Hydra on three out of four datasets and outperformed Hydra on the fourth.

We see two possible areas for future work. The first concerns the exploration phase of ISMAC. Instead of uniformly sampling instances on which exploration occurs, we might use g-means to determine instances that have similar features. We could then draw the samples from these partitions. This would avoid the overexploration on some similar instances. If a user then would choose to skip exploration and directly start the iterative process, g-means could be used to determine the initial partitions, instead of using random partitions.

The second improvement concerns our implementation of CSHC. We could parallelize it. This would not only give us a speed improvement, but also allow us to decide which partition should be split. Currently, we look at each partition in sequence and if we find an improving split we keep it. However if we compute

possible splits for each partition in parallel, we can compare their losses and split the partition that results in a better loss value. This might increase the overall quality of the found partitions.

One shortcoming of our approach is that cores that are not needed by ISMAC but were allocated stay idle. We could adjust our implementation such that we use all available resources and for example use the extra cores to run multiple configuration procedures on the same partition in parallel. This would use all resources effectively and give ISMAC more information about the target algorithm.

Bibliography

Kiyan Ahmadizadeh, Bistra Dilkina, Carla Gomes, and Ashish Sabharwal. An empirical study of optimization for maximizing diffusion in networks. *Principles and Practice of Constraint Programming–CP 2010*, pages 514–521, 2010.

Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: A lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

Theodore W. Anderson and Donald A. Darling. A test of goodness of fit. *Journal of the American Statistical Association*, 49(268):765–769, 1954.

Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. *Principles and Practice of Constraint Programming-CP 2009*, pages 142–157, 2009.

Domagoj Babic and Frank Hutter. Spear theorem prover. *Solver description, SAT competition*, 2007, 2007.

André Biedenkapp, Marius Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, and Holger H. Hoos. Efficient parameter importance analysis via ablation with surrogates. 2017.

Jakob Bossek, Bernd Bischl, Tobias Wagner, and Günter Rudolph. Learning feature-parameter mappings for parameter tuning via the profile expected improvement. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO 15*. Association for Computing Machinery (ACM), 2015.

Marco Collautti, Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. SNNAP: Solver-based nearest neighbor for algorithm portfolios. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 435–450. Springer, 2013.

Katharina Eggensperger, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Efficient benchmarking of hyperparameter optimizers via surrogates. In *AAAI*, pages 1114–1120, 2015.

Katharina Eggensperger, Marius Lindauer, and Frank Hutter. Pitfalls and best practices in algorithm configuration. 2017.

Chris Fawcett and Holger H. Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, 2016.

Chris Fawcett, Malte Helmert, Holger H. Hoos, Erez Karpas, Gabriele Röger, and Jendrik Seipp. FD-Autotune: Domain-specific configuration using fast downward. In *ICAPS 2011 Workshop on Planning and Learning*, pages 13–17, 2011.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

David Ginsbourger, Jean Baccou, Clément Chevalier, Frédéric Perales, Nicolas Garland, and Yann Monerie. Bayesian adaptive reconstruction of profile optima and optimizers. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):490–510, 2014.

Greg Hamerly and Charles Elkan. Learning the k in k-means. In *Advances in Neural Information Processing Systems*, pages 281–288, 2004.

Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Automated configuration of mixed integer programming solvers. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 186–202, 2010.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. *LION*, 5:507–523, 2011.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Identifying key algorithm parameters and instance features using forward selection. In *International Conference on Learning and Intelligent Optimization*, pages 364–381. Springer, 2013.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *International Conference on Machine Learning*, pages 754–762, 2014a.

Frank Hutter, Manuel López-Ibánez, Chris Fawcett, Marius Lindauer, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. AClib: A benchmark library for algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 36–40. Springer, 2014b.

Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014c.

Frank Hutter, Marius Lindauer, Adrian Balint, Sam Bayless, Holger H. Hoos, and Kevin Leyton-Brown. The configurable SAT solver challenge (CSSC). *Artificial Intelligence*, 243:1–25, 2017.

Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.

Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *International Conference on Principles and Practice of Constraint Programming*, pages 454–469. Springer, 2011.

Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *IJCAI*, volume 9, pages 517–524, 2009.

Marius Lindauer, Holger H. Hoos, and Frank Hutter. From sequential algorithm selection to parallel portfolio selection. In *International Conference on Learning and Intelligent Optimization*, pages 1–16. Springer, 2015a.

Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015b.

Stuart Lloyd. Least squares quantization in pcm. *IEEE Transactions on Onformation Theory*, 28(2):129–137, 1982.

Manuel López-Ibánez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

Yuri Malitsky and Meinolf Sellmann. Stochastic offline programming. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*. Institute of Electrical and Electronics Engineers (IEEE), 2009.

Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *IJCAI*, 2013.

George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions — I. *Mathematical Programming*, 14(1):265–294, 1978.

Karl Pearson. LIII. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

John R. Rice. The algorithm selection problem. *Advances in Computers*, 15: 65–118, 1976.

Marius Schneider and Holger H. Hoos. Quantifying homogeneity of instance sets for algorithm configuration. In *Lecture Notes in Computer Science*, pages 190–204. Springer Nature, 2012.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM, 2013.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, volume 10, pages 210–216, 2010.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming. In *RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 16–30, 2011.

Lin Xu, Frank Hutter, Jonathan Shen, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.

Xi Yun and Susan Epstein. Learning algorithm portfolios for parallel execution. *Learning and Intelligent Optimization*, pages 323–338, 2012.