# Algorithm Selection for Combinatorial Search Problems: A Survey

Lars Kotthoff(✉)

University of British Columbia, Vancouver, Canada
larsko@cs.ubc.ca

**Abstract.** The Algorithm Selection Problem is concerned with selecting the best algorithm to solve a given problem on a case-by-case basis. It has become especially relevant in the last decade, as researchers are increasingly investigating how to identify the most suitable existing algorithm for solving a problem instead of developing new algorithms. This survey presents an overview of this work focusing on the contributions made in the area of combinatorial search problems, where Algorithm Selection techniques have achieved significant performance improvements. We unify and organise the vast literature according to criteria that determine Algorithm Selection systems in practice. The comprehensive classification of approaches identifies and analyses the different directions from which Algorithm Selection has been approached. This chapter contrasts and compares different methods for solving the problem as well as ways of using these solutions.

## 1 Introduction

For many years, Artificial Intelligence research has been focusing on inventing new algorithms and approaches for solving similar kinds of problems. In some scenarios, a new algorithm is clearly superior to previous approaches. In the majority of cases however, a new approach will improve over the current state of the art only for some problems. This may be because it employs a heuristic that fails for problems of a certain type or because it makes other assumptions about the problem or environment that are not satisfied in some cases. Selecting the most suitable algorithm for a particular problem aims at mitigating these problems and has the potential to significantly increase performance in practice. This is known as the Algorithm Selection Problem.

The Algorithm Selection Problem has, in many forms and with different names, cropped up in many areas of Artificial Intelligence in the last few decades. Today there exists a large amount of literature on it. Most publications are concerned with new ways of tackling this problem and solving it efficiently in practice. Especially for combinatorial search problems, the application of Algorithm Selection techniques has resulted in significant performance improvements that leverage the diversity of systems and techniques developed in recent years. This chapter surveys the available literature and describes how research has progressed.


C. Bessiere et al. (Eds.): Data Mining and Constraint Programming, LNAI 10101, pp. 149–190, 2016.
DOI: 10.1007/978-3-319-50137-6_7

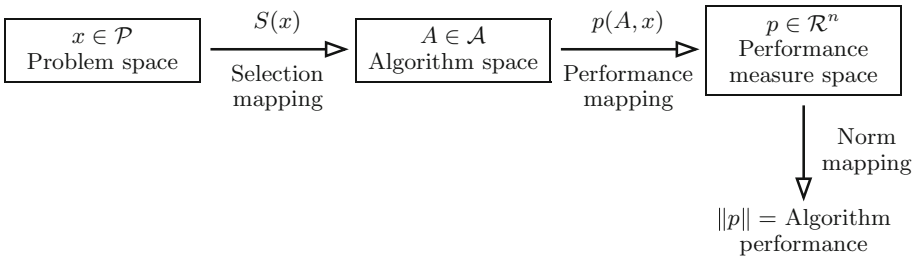


**Fig. 1.** Basic model for the Algorithm Selection Problem as published in [120].

Researchers have long ago recognised that a single algorithm will not give the best performance across all problems one may want to solve and that selecting the most appropriate method is likely to improve the overall performance. Empirical evaluations have provided compelling evidence for this, e.g. [1,154].

The original description of the Algorithm Selection Problem was published in [120]. The basic model described in the paper is very simple – given a space of problems and a space of algorithms, map each problem-algorithm pair to its performance. This mapping can then be used to select the best algorithm for a given problem. The original figure that illustrates the model is reproduced in Fig. 1. As Rice states,

> "The objective is to determine $S(x)$ [the mapping of problems to algorithms] so as to have high algorithm performance."

He identifies the following four criteria for the selection process.

1. Best selection for all mappings $S(x)$ and problems $x$. For every problem, an algorithm is chosen to give maximum performance.
2. Best selection for a subclass of problems. A single algorithm is chosen to apply to each of a subclass of problems such that the performance degradation compared to choosing from all algorithms is minimised.
3. Best selection from a subclass of mappings. Choose the selection mapping from a subset of all mappings from problems to algorithms such that the performance degradation is minimised.
4. Best selection from a subclass of mappings and problems. Choose a single algorithm from a subset of all algorithms to apply to each of a subclass of problems such that the performance degradation is minimised.

The first case is clearly the most desirable one. In practice however, the other cases are more common – we might not have enough data about individual problems or algorithms to select the best mapping for everything.

[120] lists five main steps for solving the problem.

**Formulation.** Determination of the subclasses of problems and mappings to be used.

**Existence.** Does a best selection mapping exist?
**Uniqueness.** Is there a unique best selection mapping?
**Characterization.** What properties characterize the best selection mapping and serve to identify it?
**Computation.** What methods can be used to actually obtain the best selection mapping?

This framework is taken from the theory of approximation of functions. The questions for existence and uniqueness of a best selection mapping are usually irrelevant in practice. As long as a *good* performance mapping is found and improves upon the current state of the art, the question of whether there is a different mapping with the same performance or an even better mapping is secondary. While it is easy to determine the theoretically best selection mapping on a set of given problems, casting this mapping into a *generalisable* form that will give good performance on new problems or even into a form that can be used in practice is hard. Indeed, [62] shows that the Algorithm Selection Problem in general is undecidable. It may be better to choose a mapping that generalises well rather than the one with the best performance. Other considerations can be involved as well. [28,63] compare different Algorithm selection models and select not the one with the best performance, but one with good performance that is also easy to understand, for example. [146] select their method of choice for the same reason. Similarly, [159] choose a model that is cheap to compute instead of the one with the best performance. They note that,

> "All of these techniques are computationally more expensive than ridge regression, and in our previous experiments we found that they did not improve predictive performance enough to justify this additional cost."

Rice continues by giving practical examples of where his model applies. He refines the original model to include features of problems that can be used to identify the selection mapping. The original figure depicting the refined model is given in Fig. 2. This model, or a variant of it, is what is used in most practical approaches. Including problem features is the crucial difference that often makes an approach feasible.

For each problem in a given set, the features are extracted. The aim is to use these features to produce the mapping that selects the algorithm with the best performance for each problem. The actual performance mapping for each problem-algorithm pair is usually of less interest as long as the individual best algorithm can be identified.

Rice poses additional questions about the determination of features.

- What are the best features for predicting the performance of a specific algorithm?
- What are the best features for predicting the performance of a specific class of algorithms?
- What are the best features for predicting the performance of a subclass of selection mappings?

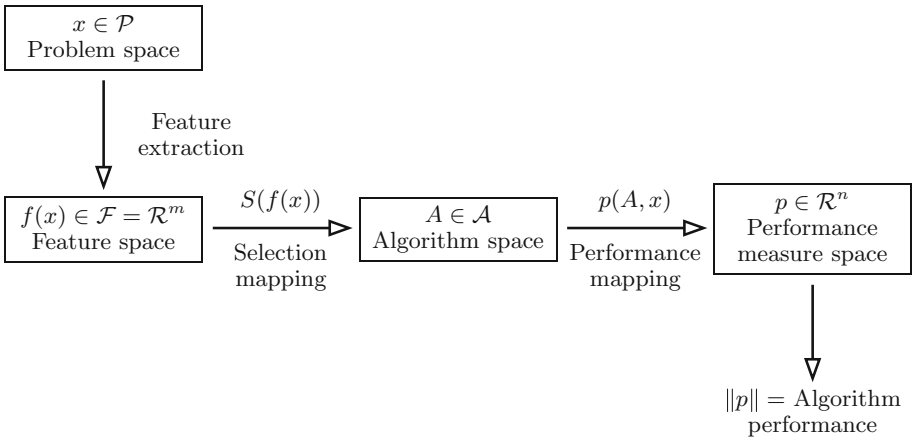


**Fig. 2.** Refined model for the Algorithm Selection Problem with problem features [120].

He also states that,

> "The determination of the best (or even good) features is one of the most important, yet nebulous, aspects of the algorithm selection problem."

He refers to the difficulty of knowing the problem space. Many problem spaces are not well known and often a sample of problems is drawn from them to evaluate empirically the performance of the given set of algorithms. If the sample is not representative, or the features do not facilitate a good separation of the problem classes in the feature space, there is little hope of finding the best or even a good selection mapping.

[145] note that,

> "While it seems that restricting a heuristic to a special case would likely improve its performance, we feel that the ability to partition the problem space of some $\mathcal{NP}$-hard problems by efficient selectors is mildly surprising."

This sentiment was shared by many researchers and part of the great prominence of Algorithm Selection systems especially for combinatorial search problems can probably be attributed to the surprise that it actually works.

Most approaches employ Machine Learning to learn the performance mapping from problems to algorithms using features extracted from the problems. This often involves a *training phase*, where the candidate algorithms are run on a sample of the problem space to experimentally evaluate their performance. This training data is used to create a *performance model* that can be used to predict the performance on new, unseen problems. The term *model* is used only in the loosest sense here; it can be as simple as a representation of the training data without any further analysis.

### 1.1 Practical Motivation

[1] notes that in Machine Learning, researchers often perform experiments on a limited number of data sets to demonstrate the performance improvements achieved and implicitly assume that these improvements generalise to other data. He proposes a framework for better experimental evaluation of such claims and deriving rules that determine the properties a data set must have in order for an algorithm to have superior performance. His objective is

> "... to derive rules of the form 'this algorithm outperforms these other algorithms on these dependent measures for databases with these characteristics'. Such rules summarize *when* [...] rather than *why* the observed performance difference occurred."

[143] make similar observations and show that there is no algorithm that is universally the best when solving constraint problems. They also demonstrate that the best algorithm-heuristic combination is not what one might expect for some of the surveyed problems. This provides an important motivation for research into performing Algorithm Selection automatically. They close by noting that,

> "... research should focus on how to retrieve the most efficient [algorithm-heuristic] combinations for a problem."

The focus of Algorithm Selection is on identifying algorithms with good performance, not on providing explanations for why this is the case. Most publications do not consider the question of "Why?" at all. Rice's framework does not address this question either. The simple reason for this is that explaining the Why? is difficult and for most practical applications not particularly relevant as long as improvements can be achieved. Research into what makes a problem hard, how this affects the behaviour of specific algorithms and how to exploit this knowledge is a fruitful area, but outside the scope of this chapter. However, we present a brief exposition of one of the most important concepts to illustrate its relevance.

The notion of a *phase transition* [26] refers to a sudden change in the hardness of a problem as the value of a single parameter of the problem is changed. Detecting such transitions is an obvious way to facilitate Algorithm Selection. [65] note that,

> "In particular, the location of the phase transition point might provide a systematic basis for selecting the type of algorithm to use on a given problem."

While some approaches make use of this knowledge to generate challenging training problems for their systems, it is hardly used at all to facilitate Algorithm Selection. [109] use a set of features that can be used to characterise a phase transition and note that,

> "It turns out that [...] this group of features alone suffices to construct reasonably good models."

It remains unclear how relevant phase transitions are to Algorithm Selection in practice. On one hand, their theoretical properties seem to make them highly suitable, but on the other hand almost nobody has explored their use in actual Algorithm Selection systems.

**No Free Lunch Theorems.** The question arises of whether, in general, the performance of a system can be improved by always picking the best algorithm. The "No Free Lunch" (NFL) theorems [154] state that no algorithm can be the best across all possible problems and that on average, all algorithms perform the same. This seems to provide a strong motivation for Algorithm Selection – if, on average, different algorithms are the best for different parts of the problem space, selecting them based on the problem to solve has the potential to improve performance.

The theorems would apply to Algorithm Selection systems themselves as well though (in particular the version for supervised learning are relevant, see [153]). This means that although performance improvements can be achieved by selecting the right algorithms on one part of the problem space, wrong decisions will be made on other parts, leading to a loss of performance. On average over all problems, the performance achieved by an Algorithm Selection meta-algorithm will be the same as that of all other algorithms.

The NFL theorems are the source of some controversy however. Among the researchers to doubt their applicability is the first proponent of the Algorithm Selection Problem [121]. Several other publications show that the assumptions underlying the NFL may not be satisfied [31,119]. In particular, the distribution of the best algorithms from the portfolio to problems is not random – it is certainly true that certain algorithms are the best on a much larger number of problems than others.

A detailed assessment of the applicability of the NFL theorems to the Algorithm Selection Problem is outside the scope of this chapter. However, a review of the literature suggests that, if the theorems are applicable, the ramifications in practice may not be significant. Most of the many publications surveyed here do achieve performance improvements across a range of different problems using Algorithm Selection techniques. As a research area, it is very active and thriving despite the potentially negative implications of the NFL.

### 1.2 Scope and Related Work

Algorithm Selection is a very general concept that applies not only in almost all areas of Computer Science, but also other disciplines. However, it is especially relevant in many areas of Artificial Intelligence. This is a large field itself though and surveying all Artificial Intelligence publications that are relevant to Algorithm Selection in a single chapter is infeasible.

In this chapter, we focus on Algorithm Selection for *combinatorial search problems*. This is a large and important subfield of Artificial Intelligence where Algorithm Selection techniques have become particularly prominent in recent

years because of the impressive performance improvements that have been achieved by some approaches. Combinatorial search problems include for example satisfiability (SAT), constraint problems, planning, quantified Boolean formulae (QBF), scheduling and combinatorial optimisation.

A combinatorial search problem is one where an initial state is to be transformed into a goal state by application of a series of operators, such as assignment of values to variables. The space of possible states is typically exponential in the size of the input and finding a solution is $\mathcal{NP}$-hard. A common way of solving such problems is to use *heuristics*. A heuristic is a strategy that determines which operators to apply when. Heuristics are not necessarily complete or deterministic, i.e. they are not guaranteed to find a solution if it exists or to always make the same decision under the same circumstances. The nature of heuristics makes them particularly amenable to Algorithm Selection – choosing a heuristic manually is difficult even for experts, but choosing the correct one can improve performance significantly.

There exists a large body of work that is relevant to Algorithm Selection in the Machine Learning literature. [133] presents a survey of many approaches. Repeating this here is unnecessary and outside the scope of this chapter, which focuses on the application of such techniques. The most relevant area of research is that into *ensembles*, where several models are created instead of one. Such ensembles are either implicitly assumed or explicitly engineered so that they complement each other. Errors made by one model are corrected by another. Ensembles can be engineered by techniques such as *bagging* [18] and *boosting* [128]. [9,111] present studies that compare bagging and boosting empirically. [30] provides explanations for why ensembles can perform better than individual algorithms.

There is increasing interest in the integration of Algorithm Selection techniques with programming language paradigms, e.g. [4,68]. While these issues are sufficiently relevant to be mentioned here, exploring them in detail is outside the scope of the chapter. Similarly, technical issues arising from the computation, storage and application of performance models, the integration of Algorithm Selection techniques into complex systems, the execution of choices and the collection of experimental data to facilitate Algorithm Selection are not surveyed here.

### 1.3 Terminology

Algorithm Selection is a widely applicable concept and as such has cropped up frequently in various lines of research. Often, different terminologies are used.

[15] use the term *algorithm chaining* to mean switching from one algorithm to another while the problem is being solved. [100] call Algorithm Selection *selection by performance prediction*. [145] use the term *hybrid algorithm* for the combination of a set of algorithms and an Algorithm Selection model (which they term *selector*).

In Machine Learning, Algorithm Selection is usually referred to as *meta-learning*. This is because Algorithm Selection models for Machine Learning learn

when to use which method of Machine Learning. The earliest approaches also spoke of *hybrid approaches*, e.g. [144]. [1] proposes rules for selecting a Machine Learning algorithm that take the characteristics of a data set into account. He uses the term *meta-learning*. [20] introduces the notion of *selective superiority*. This concept refers to a particular algorithm being best on some, but not all tasks.

In addition to the many terms used for the process of Algorithm Selection, researchers have also used different terminology for the models of what Rice calls *performance measure space*. [2] call them *runtime performance predictors*. [75,95,96,156] coined the term *Empirical Hardness model*. This stresses the reliance on empirical data to create these models and introduces the notion of *hardness* of a problem. The concept of hardness takes into account all performance considerations and does not restrict itself to, for example, runtime performance. In practice however, the described empirical hardness models only take runtime performance into account. In all cases, the predicted measures are used to select an algorithm.

Throughout this chapter, the term *algorithm* is used to refer to what is selected for solving a problem instance. This is for consistency and to make the connection to Rice's framework. An algorithm may be a system, a programme, a heuristic, a classifier or a configuration. This is not made explicit unless it is relevant in the particular context.

### 1.4 Organisation

An organisation of the Algorithm Selection literature is challenging, as there are many different criteria that can be used to classify it. Each publication can be evaluated from different points of view. The organisation of this chapter follows the main criteria below.

**What to select algorithms from**
Section 2 describes how sets of algorithms, or *portfolios*, can be constructed. A portfolio can be *static*, where the designer decides which algorithms to include, or *dynamic*, where the composition or individual algorithms vary or change for different problems.

**What to select and when**
Section 3 describes how algorithms from portfolios are selected to solve problems. Apart from the obvious approach of picking a single algorithm, time slots can be allocated to all or part of the algorithms or the execution monitored and earlier decisions revised. We also distinguish between selecting before the solving of the actual problem starts and while the problem is being solved.

**How to select**
Section 4 surveys techniques used for making the choices described in Sect. 3. It details how performance models can be built and what kinds of predictions they inform. Example predictions are the best algorithm in the portfolio and the runtime performance of each portfolio algorithm.

**How to facilitate the selection**
Section 5 gives an overview of the types of analysis different approaches perform and what kind of information is gathered to facilitate Algorithm Selection. This includes the past performance of algorithms and structural features of the problems to be solved.

The order of the material follows a top-down approach. Starting with the high-level idea of Algorithm Selection, as proposed by [120] and described in this introduction, more technical details are gradually explored. Earlier concepts provide motivation and context for later technical details. For example, the choice of whether to select a single algorithm or monitor its execution (Sect. 3) determines the types of predictions required and techniques suitable for making them (Sect. 4) as well as the properties that need to be measured (Sect. 5).

The individual sections are largely self-contained. If the reader is more interested in a bottom-up approach that starts with technical details on what can be observed and measured to facilitate Algorithm Selection, Sects. 2 through 5 may be read in reverse order.

Section 6 again illustrates the importance of the field by surveying the many different application domains of Algorithm Selection techniques with a focus on combinatorial search problems. We close by summarising in Sect. 7.

## 2 Algorithm Portfolios

For diverse sets of problems, it is unlikely that a single algorithm will be the most suitable one in all cases. A way of mitigating this restriction is to use a *portfolio* of algorithms. This idea is closely related to the notion of Algorithm Selection itself – instead of making an up-front decision on what algorithm to use, it is decided on a case-by-case basis for each problem individually. In the framework presented by [120], portfolios correspond to the algorithm space $\mathcal{A}$.

Portfolios are a well-established technique in Economics. Portfolios of assets, securities or similar products are used to reduce the risk compared to holding only a single product. The idea is simple – if the value of a single security decreases, the total loss is less severe. The problem of allocating funds to the different parts of the portfolio is similar to allocating resources to algorithms in order to solve a computational problem. There are some important differences though. Most significantly, the past performance of an algorithm can be a good indicator of future performance. There are fewer factors that affect the outcome and in most cases, they can be measured directly. In Machine Learning, *ensembles* [30] are instances of algorithm portfolios. In fact, the only difference between algorithm portfolios and Machine Learning ensembles is the way in which its constituents are used.

The idea of algorithm portfolios was first presented by [73]. They describe a formal framework for the construction and application of algorithm portfolios and evaluate their approach on graph colouring problems. Within the Artificial Intelligence community, algorithm portfolios were popularised by [57,58] and a

subsequent extended investigation [59]. The technique itself however had been described under different names by other authors at about the same time in different contexts.

[143] experimentally show for a selection of constraint satisfaction algorithms and heuristics that none is the best on all evaluated problems. They do not mention portfolios, but propose that future research should focus on identifying when particular algorithms and heuristics deliver the best performance. This implicitly assumes a portfolio to choose algorithms from. [2] perform a similar investigation and come to similar conclusions. They talk about selecting an appropriate algorithm from an *algorithm family*.

Beyond the simple idea of using a set of algorithms instead of a single one, there is a lot of scope for different approaches. One of the first problems faced by researchers is how to construct the portfolio. There are two main types. *Static portfolios* are constructed offline before any problems are solved. While solving a problem, the composition of the portfolio and the algorithms within it do not change. *Dynamic portfolios* change in composition, configuration of the constituent algorithms or both during solving.

### 2.1 Static Portfolios

Static portfolios are the most common type. The number of algorithms or systems in the portfolio is fixed, as well as their parameters. In Rice's notation, the algorithm space $\mathcal{A}$ is constant, finite and known. This approach is used for example in SATzilla [109,158,159], AQME [117,118], CPhydra [110], ArgoSmArT [108], BUS [72] and Proteus [74].

The vast majority of approaches composes static portfolios from different algorithms or different algorithm configurations. [73] however use a portfolio that contains the same randomised algorithm twice. They run the portfolio in parallel and as such essentially use the technique to parallelise an existing sequential algorithm.

Some approaches use a large number of algorithms in the portfolio, such as ArgoSmArT, whose portfolio size is 60. SATzilla uses 19 algorithms, although the authors use portfolios containing only subsets of those for specific applications. BUS uses six algorithms and CPhydra five. [54] select from a portfolio of only two algorithms. AQME has different versions with different portfolio sizes, one with 16 algorithms, one with five and three algorithms of different types and one with two algorithms [118]. The authors compare the different portfolios and conclude that the one with eight algorithms offers the best performance, as it has more variety than the portfolio with two algorithms and it is easier to make a choice for eight than for 16 algorithms. There are also approaches that use portfolios of variable size that is determined by training data [81,157]. [74] combine algorithms and problem encodings in a portfolio – problem instances can be translated into alternative representations, for which other algorithms are available.

As the algorithms in the portfolio do not change, their selection is crucial for its success. Ideally, the algorithms will complement each other such that good

performance can be achieved on a wide range of different problems. [66] report that portfolios composed of a random selection from a large pool of diverse algorithms outperform portfolios composed of the algorithms with the best overall performance. They develop a framework with a mathematical model that theoretically justifies this observation. [126] use a portfolio of heuristics for solving quantified Boolean formulae problems that have specifically been crafted to be orthogonal to each other. [157] automatically engineer a portfolio with algorithms of complementary strengths. In [162], the authors analyse the contributions of the portfolio constituents to the overall performance and conclude that not algorithms with the best overall performance, but with techniques that set them apart from the rest contribute most. [81] use a static portfolio of variable size that adapts itself to the training data. They cluster the training problems and choose the best algorithm for each cluster. They do not emphasise diversity, but suitability for distinct parts of the problem space. [157] also construct a portfolio with algorithms that perform well on different parts of the problem space, but do not use clustering.

In financial theory, constructing portfolios can be seen as a quadratic optimisation problem. The aim is to balance expected performance and risk (the expected variation of performance) such that performance is maximised and risk minimised. [37] solve this problem for algorithm portfolios using genetic algorithms.

Most approaches make the composition of the portfolio less explicit. Many systems use portfolios of solvers that have performed well in solver competitions with the implicit assumption that they have complementing strengths and weaknesses and the resulting portfolio will be able to achieve good performance.

### 2.2 Dynamic Portfolios

Rather than relying on a priori properties of the algorithms in the portfolio, dynamic portfolios adapt the composition of the portfolio or the algorithms depending on the problem to be solved. The algorithm space $\mathcal{A}$ changes with each problem and is a subspace of the potentially infinite super algorithm space $\mathcal{A}'$. This space contains all possible (hypothetical) algorithms that could be used to solve problems from the problem space. In static portfolios, the algorithms in the portfolio are selected from $\mathcal{A}'$ once either manually by the designer of the portfolio or automatically based on empirical results from training data.

One approach is to build a portfolio by combining algorithmic building blocks. An example of this is the Adaptive Constraint Engine (ACE) [35,36]. The building blocks are so-called advisors, which characterise variables of the constraint problem and give recommendations as to which one to process next. ACE combines these advisors into more complex ones. [33,34] use a similar idea to construct search strategies for solving constraint problems. [42,43] proposes CLASS, which combines heuristic building blocks to form composite heuristics for solving SAT problems. In these approaches, there is no strong notion of a portfolio – the algorithm or strategy used to solve a problem is assembled from lower level components.

Closely related is the concept of specialising generic building blocks for the problem to solve. This approach is taken in the SAGE system (Strategy Acquisition Governed by Experimentation) [92,93]. It starts with a set of general operators that can be applied to a search state. These operators are refined by making the preconditions more specific based on their utility for finding a solution. The MULTI-TAC (Multi-tactic Analytic Compiler) system [103–105] specialises a set of generic heuristics for the constraint problem to solve.

There can be complex restrictions on how the building blocks are combined. RT-Syn [131] for example uses a preprocessing step to determine the possible combinations of algorithms and data structures to solve a software specification problem and then selects the most appropriate combination using simulated annealing. [8] model the construction of a constraint solver from components as a constraint problem whose solutions denote valid combinations of components.

Another approach is to modify the parameters of parameterised algorithms in the portfolio. This is usually referred to as automatic tuning and not only applicable in the context of algorithm portfolios, but also for single algorithms. The HAP system [146] automatically tunes the parameters of a planning system depending on the problem to solve. [70] dynamically modify algorithm parameters during search based on statistics collected during the solving process.

**Automatic Tuning.** The area of automatic parameter tuning has attracted a lot of attention in recent years. This is because algorithms have an increasing number of parameters that are difficult to tune even for experts and because of research into dynamic algorithm portfolios that benefits from automatic tuning. A survey of the literature on automatic tuning is outside the scope of this chapter, but some of the approaches that are particularly relevant to this survey are described below.

Automatic tuning and portfolio selection can be treated separately, as done in the Hydra portfolio builder [157]. Hydra uses ParamILS [78,79] to automatically tune algorithms in a SATzilla [159] portfolio. Autofolio [98] uses ParamILS and SMAC [76] to train a claspfolio [67] portfolio. ISAC [81] uses GGA [5] to automatically tune algorithms for clusters of problem instances.

[105] first enumerates all possible rule applications up to a certain time or size bound. Then, the most promising configuration is selected using beam search, a form of parallel hill climbing, that empirically evaluates the performance of each candidate. [8] use hill climbing to similarly identify the most efficient configuration for a constraint solver on a set of problems. [42,141] use genetic algorithms to evolve promising configurations.

The systems described in the previous paragraph are only of limited suitability for dynamic algorithm portfolios. They either take a long time to find good configurations or are restricted in the number or type of parameters. Interactions between parameters are only taken into account in a limited way. More recent approaches have focused on overcoming these limitations.

The ParamILS system [78,79] uses techniques based on local search to identify parameter configurations with good performance. The authors address

over-confidence (overestimating the performance of a parameter configuration on a test set) and over-tuning (determining a parameter configuration that is too specific). SMAC [76] builds a model of the performance response surface in parameter space to predict where the most promising configurations are. [5] use genetic algorithms to discover favourable parameter configurations for the algorithms being tuned. The authors use a racing approach to avoid having to run all generated configurations to completion. They also note that one of the advantages of the genetic algorithm approach is that it is inherently parallel.

Both of these approaches are capable of tuning algorithms with a large number of parameters and possible values as well as taking interactions between parameters into account. They are used in practice in the Algorithm Selection systems Hydra and ISAC, respectively. In both cases, they are only used to construct static portfolios however. More recent approaches focus on exploiting parallelism, e.g. [77,97].

Dynamic portfolios are in general a more fruitful area for Algorithm Selection research because of the large space of possible decisions. Static portfolios are usually relatively small and the decision space is amenable for human exploration. This is not a feasible approach for dynamic portfolios though. [105] notes that

> "MULTI-TAC turned out to have an unexpected advantage in this arena, due to the complexity of the task. Unlike our human subjects, MULTI-TAC experimented with a wide variety of combinations of heuristics. Our human subjects rarely had the inclination or patience to try many alternatives, and on at least one occasion incorrectly evaluated alternatives that they did try."

## 3 Problem Solving with Portfolios

Once an algorithm portfolio has been constructed, the way in which it is to be used has to be decided. There are different considerations to take into account. The two main issues are as follows.

**What to select**
: Given the full set of algorithms in the portfolio, a subset has to be chosen for solving the problem. This subset can consist of only a single algorithm that is used to solve the problem to completion, the entire portfolio with the individual algorithms interleaved or running in parallel or anything in between.

**When to select**
: The selection of the subset of algorithms can be made only once before solving starts or continuously during search. If the latter is the case, selections can be made at well-defined points during search, for example at each node of a search tree, or when the system judges it to be necessary to make a decision.

Rice's model assumes that only a single algorithm $A \in \mathcal{A}$ is selected. It implicitly assumes that this selection occurs only once and before solving the actual problem.

### 3.1 What to Select

A common and the simplest approach is to select a single algorithm from the portfolio and use it to solve the problem completely. This single algorithm has been determined to be the best for the problem at hand. For example SATzilla [109,158,159], ArgoSmArT [108], SALSA [29] and Eureka [28] do this. The disadvantage of this approach is that there is no way of mitigating a wrong selection. If an algorithm is chosen that exhibits bad performance on the problem, the system is "stuck" with it and no adjustments are made, even if all other portfolio algorithms would perform much better.

An alternative approach is to compute schedules for running (a subset of) the algorithms in the portfolio. In some approaches, the terms portfolio and schedule are used synonymously – all algorithms in the portfolio are selected and run according to a schedule that allocates time slices to each of them. The task of Algorithm Selection becomes determining the schedule rather than to select algorithms.

[122] rank the portfolio algorithms in order of expected performance and allocate time according to this ranking. [72] propose a round-robin schedule that contains all algorithms in the portfolio. The order of the algorithms is determined by the expected run time and probability of success. The first algorithm is allocated a time slice that corresponds to the expected time required to solve the problem. If it is unable to solve the problem during that time, it and the remaining algorithms are allocated additional time slices until the problem is solved or a time limit is reached.

[118] determine a schedule according to three strategies. The first strategy is to run all portfolio algorithms for a short time and if the problem has not been solved after this, run the predicted best algorithm exclusively for the remaining time. The second strategy runs all algorithms for the same amount of time, regardless of what the predicted best algorithm is. The third variation allocates exponentially increasing time slices to each algorithm such that the total time is again distributed equally among them. In addition to the three different scheduling strategies, the authors evaluate four different ways of ordering the portfolio algorithms within a schedule that range from ranking based on past performance to random. They conclude that ordering the algorithms based on their past performance and allocating the same amount of time to all algorithms gives the best overall performance.

[110] optimise the computed schedule with respect to the probability that the problem will be solved. They use the past performance data of the portfolio algorithms for this. However, they note that their approach of using a simple complete search procedure to find this optimal schedule relies on small portfolio sizes and that "for a large number of solvers, a more sophisticated approach would be necessary". Later approaches, e.g. the Sunny approach [3], improve on this.

[80] formulate the problem of computing a schedule that solves most problems in a training set in the lowest amount of time as a resource constrained set covering integer programme. They pursue similar aims as [110] but note that

their approach is more efficient and able to scale to larger schedules. However, their evaluation concludes that the approach with the best overall performance is to run the predicted best algorithm for 90 % of the total available time and distribute the remaining 10 % across the other algorithms in the portfolio according to a static schedule.

[113] presents a framework for calculating optimal schedules. The approach is limited by a number of assumptions about the algorithms and the execution environment, but is applicable to a wide range of research in the literature. [16,114] compute an optimal static schedule for allocating fixed time slices to each algorithm. [127] propose an algorithm to efficiently compute an optimal schedule for portfolios of fixed size and show that the problem of generating or even approximating an optimal schedule is computationally intractable. [123] explore different strategies for allocating time slices to algorithms. In a serial execution strategy, each algorithm is run once for an amount of time determined by the average time to find a solution on previous problems or the time that was predicted for finding a solution on the current problem. A round-robin strategy allocates increasing time slices to each algorithm. The length of a time slice is based on the proportion of successfully solved training problems within this time. [56] compute round-robin schedules following a similar approach. Not all of their computed schedules contain all portfolio algorithms. [138] compute a schedule with the aim of improving the average-case performance. In later work, they compute theoretical guarantees for the performance of their schedule [140].

[155] approach scheduling the chosen algorithms in a different way and assume a fixed limit on the amount of resources an algorithm can consume while solving a problem. All algorithms are run sequentially for this fixed amount of time. Similar to [56], they simulate the performance of different allocations and select the best one based on the results of these simulations. [41] estimates the performance of candidate allocations through bootstrap sampling. [57,59] also evaluate the performance of different candidate portfolios, but take into account how many algorithms can be run in parallel. They demonstrate that the optimal schedule (in this case the number of algorithms that are being run) changes as the number of available processors increases. [47] investigate how to allocate resources to algorithms in the presence of multiple CPUs that allow to run more than one algorithm in parallel. [165] craft portfolios with the specific aim of running the algorithms in parallel.

[69] consider computing optimal schedules without selection. They note that their approach can be used in a variety of settings, in particular parallel portfolios.

Related research is concerned with the scheduling of restarts of stochastic algorithms – it also investigates the best way of allocating resources. The chapter that introduced algorithm portfolios [73] uses a portfolio of identical stochastic algorithms that are run with different random seeds. There is a large amount of research on how to determine restart schedules for randomised algorithms and a survey of this is outside the scope of this chapter. A few approaches that are particularly relevant to Algorithm Selection and portfolios are mentioned below.

[70] determine the amount of time to allocate to a stochastic algorithm before restarting it. They use dynamic policies that take performance predictions into account, showing that it can outperform an optimal fixed policy.

[27] investigate a restart model that allocates resources to an algorithm proportional to the number of times it has been successful in the past. In particular, they note that the allocated resources should grow doubly exponentially in the number of successes. Allocation of fewer resources results in over-exploration (too many different things are tried and not enough resources given to each) and allocation of more resources in over-exploitation (something is tried for to too long before moving on to something different).

[139] compute restart schedules that take the runtime distribution of the portfolio algorithms into account. They present an approach that does so statically based on the observed performance on a set of training problems as well as an approach that learns the runtime distributions as new problems are solved without a separate training set.

### 3.2 When to Select

In addition to whether they choose a single algorithm or compute a schedule, existing approaches can also be distinguished by whether they operate before the problem is being solved (offline) or while the problem is being solved (online). The advantage of the latter is that more fine-grained decisions can be made and the effect of a bad choice of algorithm is potentially less severe. The price for this added flexibility is a higher overhead however, as algorithms are selected more frequently.

Examples of approaches that only make offline decisions include [105,110, 131,159]. In addition to having no way of mitigating wrong choices, often these will not even be detected. These approaches do not monitor the execution of the chosen algorithms to confirm that they conform with the expectations that led to them being chosen. Purely offline approaches are inherently vulnerable to bad choices. Their advantage however is that they only need to select an algorithm once and incur no overhead while the problem is being solved.

Moving towards online systems, the next step is to monitor the execution of an algorithm or a schedule to be able to intervene if expectations are not met. [39,40] investigates setting a time bound for the algorithm that has been selected based on the predicted performance. If the time bound is exceeded, the solution attempt is abandoned. More sophisticated systems furthermore adjust their selection if such a bound is exceeded. [15] try to detect behaviour during search that indicates that the algorithm is performing badly, for example visiting nodes in a subtree of the search that clearly do not lead to a solution. If such behaviour is detected, they propose switching the currently running algorithm according to a fixed replacement list.

[125] explore the same basic idea. They switch between two algorithms for solving constraint problems that achieve different levels of consistency. The level of consistency refers to the amount of search space that is ruled out by inference before actually searching it. Their approach achieves the same level of

search space reduction as the more expensive algorithm at a significantly lower cost. This is possible because doing more inference does not necessarily result in a reduction of the search space in all cases. The authors exploit this fact by detecting such cases and doing the cheaper inference. [112,136] also investigate switching propagation methods during solving. [163,164] do not monitor the execution of the selected algorithm, but instead the values of the features used to select it. They re-evaluate the selection function when its inputs change.

Further examples of approaches that monitor the execution of the selected algorithm are [49,118], but also [70] where the offline selection of an algorithm is combined with the online selection of a restart strategy. An interesting feature of [118] is that the authors adapt the model used for the offline algorithm selection if the actual run time is much higher than the predicted runtime. In this way, they are not only able to mitigate bad choices during execution, but also prevent them from happening again.

The approaches that make decisions during search, for example at every node of the search tree, are necessarily online systems. [6] select the best search strategy at checkpoints in the search tree. Similarly, [20] recursively partitions the classification problem to be solved and selects an algorithm for each partition. In this approach, a lower-level decision can lead to changing the decision at the level above. This is usually not possible for combinatorial search problems, as decisions at a higher level cannot be changed easily.

Closely related is the work by [90,91], which partitions the search space into recursive subtrees and selects the best algorithm from the portfolio for every subtree. They specifically consider recursive algorithms. At each recursive call, the Algorithm Selection procedure is invoked. This is a more natural extension of offline systems than monitoring the execution of the selected algorithms, as the same mechanisms can be used. [126] also select algorithms for recursively solving sub-problems.

The PRODIGY system [22] selects the next operator to apply in order to reach the goal state of a planning problem at each node in the search tree. Similarly, [92] learn weights for operators that can be applied at each search state and select from among them accordingly.

Most approaches rely on an offline element that makes a decision before search starts. In the case of recursive calls, this is no different from making a decision during search however. [44,46,49] on the other hand learn the Algorithm Selection model only dynamically while the problem is being solved. Initially, all algorithms in the portfolio are allocated the same (small) time slice. As search progresses, the allocation strategy is updated, giving more resources to algorithms that have exhibited better performance. The expected fastest algorithm receives half of the total time, the next best algorithm half of the remaining time and so on. [7] also rely exclusively on a selection model trained online in a similar fashion. They evaluate different strategies of allocating resources to algorithms according to their progress during search. All of these strategies converge to allocating all resources to the algorithm with the best observed performance.

## 4 Portfolio Selectors

Research on *how* to select from a portfolio in an Algorithm Selection system has generated the largest number of different approaches within the framework of Algorithm Selection. In Rice's framework, it roughly corresponds to the performance mapping $p(A, x)$, although only few approaches use this exact formulation. Rice assumes that the performance of a particular algorithm on a particular problem is of interest. While this is true in general, many approaches only take this into account implicitly. Selecting the single best algorithm for a problem for example has no explicit mapping into Rice's performance measure space $\mathcal{R}^n$ at all. The selection mapping $S(f(x))$ is also related to the problem of how to select.

There are many different ways a mechanism to select from a portfolio can be implemented. Apart from accuracy, one of the main requirements for such a selector is that it is relatively cheap to run – if selecting an algorithm for solving a problem is more expensive than solving the problem, there is no point in doing so. [145] explicitly define the selector as "an efficient (polynomial time) procedure".

There are several challenges associated with making selectors efficient. Algorithm Selection systems that analyse the problem to be solved, such as SATzilla, need to take steps to ensure that the analysis does not become too expensive. Two such measures are the running of a pre-solver and the prediction of the time required to analyse a problem [159]. The idea behind the pre-solver is to choose an algorithm with reasonable general performance from the portfolio and use it to start solving the problem before starting to analyse it. If the problem happens to be very easy, it will be solved even before the results of the analysis are available. After a fixed time, the pre-solver is terminated and the results of the Algorithm Selection system are used. [118] use a similar approach and run all algorithms for a short time in one of their strategies. Only if the problem has not been solved after that, they move on to the algorithm that was actually selected.

Predicting the time required to analyse a problem is a closely related idea. If the predicted required analysis time is too high, a default algorithm with reasonable performance is chosen and run on the problem. This technique is particularly important in cases where the problem is hard to analyse, but easy to solve. As some systems use information that comes from exploring part of the search space (cf. Sect. 5), this is a very relevant concern in practice. On some problems, even probing just a tiny part of the search space may take a very long time.

[54,55] report that using the misclassification penalty as a weight for the individual problems during training improves the quality of the predictions. The misclassification penalty quantifies the "badness" of a wrong prediction; in this case as the additional time required to solve a problem. If an algorithm was chosen that is only slightly worse than the best one, it has less impact than choosing an algorithm that is orders of magnitude worse. Using the penalty

during training is a way of guiding the learned model towards the problems where the potential performance improvement is large.

There are many different approaches to how portfolio selectors operate. The selector is not necessarily an explicit part of the system. [105] compiles the Algorithm Selection system into a Lisp programme for solving the original constraint problem. The selection rules are part of the programme logic. [43,50] evolve selectors and combinators of heuristic building blocks using genetic algorithms. The selector is implicit in the evolved programme.

### 4.1 Performance Models

The way the selector operates is closely linked to the way the performance model of the algorithms in the portfolio is built. In early approaches, the performance model was usually not learned but given in the form of human expert knowledge. [15,125] use hand-crafted rules to determine whether to switch the algorithm during solving. [2] also have hand-crafted rules, but estimate the runtime performance of an algorithm. More recent approaches sometimes use only human knowledge as well. [150] select a local search heuristic for solving SAT problems by a hand-crafted rule that considers the distribution of clause weights. [142] model the performance space manually using statistical methods and use this hand-crafted model to select a heuristic for solving constraint problems. [146] learn rules automatically, but then filter them manually.

A more common approach today is to automatically learn performance models using Machine Learning on training data. The portfolio algorithms are run on a set of representative problems and based on these experimental results, performance models are built. This approach is used by [61,81,110,117,159], to name but a few examples. A drawback of this approach is that the training time is usually large. [45] investigate ways of mitigating this problem by using censored sampling, which introduces an upper bound on the runtime of each experiment in the training phase. [85] also investigate censored sampling where not all algorithms are run on all problems in the training phase. Their results show that censored sampling may not have a significant effect on the performance of the learned model.

Models can also be built without a separate training phase, but while the problem is solved. This approach is used by [7,46] for example. While this significantly reduces the time to build a system, it can mean that the result is less effective and efficient. At the beginning, when no performance models have been built, the decisions of the selector might be poor. Furthermore, creating and updating performance models while the problem is being solved incurs an overhead.

The choice of Machine Learning technique is affected by the way the portfolio selector operates. Some techniques are more amenable to offline approaches (e.g. linear regression models used by [159]), while others lend themselves to online methods (e.g. reinforcement learning used by [7]).

Performance models can be categorised by the type of entity whose performance is modelled – the entire portfolio or individual algorithms within it.

There are publications that use both of those categories however, e.g. [134]. In some cases, no performance models as such are used at all. [8,25,105] run the candidates on a set of test problems and select the one with the best performance that way for example. [56,57,155] simulate the performance of different selections on training data.

**Per-Portfolio Models.** One automated approach is to learn a performance model of the entire portfolio based on training data. Usually, the prediction of such a model is the best algorithm from the portfolio for a particular problem. There is only a weak notion of an individual algorithm's performance. In Rice's notation for the performance mapping $P(A, x)$, $A$ is the (subset of the) portfolio instead of an individual algorithm, i.e. $A \subseteq \mathcal{A}$ instead of Rice's $A \in \mathcal{A}$.

This is used for example by [28,61,108,110,117]. Again there are different ways of doing this. Lazy approaches do not learn an explicit model, but use the set of training examples as a case base. For new problems, the closest problem or the set of $n$ closest problems in the case base is determined and decisions made accordingly. [51,101,108,110,117,151] use nearest-neighbour classifiers to achieve this. Apart from the conceptual simplicity, such an approach is attractive because it does not try to abstract from the examples in the training data. The problems that Algorithm Selection techniques are applied to are usually complex and factors that affect the performance are hard to understand. This makes it hard to assess whether a learned abstract model is appropriate and what its requirements and limitations are.

Explicitly-learned models try to identify the concepts that affect performance for a given problem. This acquired knowledge can be made explicit to improve the understanding of the researchers of the problem domain. There are several Machine Learning techniques that facilitate this, as the learned models are represented in a form that is easy to understand by humans. [20,22,60,146] learn classification rules that guide the selector. [146] note that the decision to use a classification rule leaner was not so much guided by the performance of the approach, but the easy interpretability of the result. [36,92,107] learn weights for decision rules to guide the selector towards the best algorithms. [12,28,54,61,63,122] go one step further and learn decision trees. [63] again note that the reason for choosing decision trees was not primarily the performance, but the understandability of the result. [116] show the set of learned rules in the paper to illustrate its compactness. Similarly, [54] show their final decision tree in the paper.

Some approaches learn probabilistic models that take uncertainty and variability into account. [60] use a probabilistic model to learn control rules. The probabilities for candidate rules being beneficial are evaluated and updated on a training set until a threshold is reached. This methodology is used to avoid having to evaluate candidate rules on larger training sets, which would show their utility more clearly but be more expensive. [29] learn multivariate Bayesian decision rules. [23] learn a Bayesian classifier to predict the best algorithm after a certain amount of time. [137] learn Bayesian models that incorporate collaborative filtering. [32] learn decision rules using naïve Bayes classifiers. [90,113] learn

performance models based on Markov Decision Processes. [85] use statistical relational learning to predict the ranking of the algorithms in the portfolio on a particular problem. None of these approaches make explicit use of the uncertainty attached to a decision though.

Other approaches include support vector machines [6,71], reinforcement learning [7], neural networks [44], decision tree ensembles [71], ensembles of general classification algorithms [87], boosting [12], hybrid approaches that combine regression and classification [83], multinomial logistic regression [126], self-organising maps [134] and clustering [81,102,136]. [127,138] compute schedules for running the algorithms in the portfolio based on a statistical model of the problem instance distribution and performance data for the algorithms. This is not an exhaustive list, but focuses on the most prominent approaches and publications. Within a single family of approaches, such as decision trees, there are further distinctions that are outside the scope of this chapter, such as the type of decision tree inducer.

[6] discuss a technical issue related to the construction of per-portfolio performance models. A particular algorithm often exhibits much better performance in general than other algorithms on a particular instance distribution. Therefore, the training data used to learn the performance model will be skewed towards that algorithm. This can be a problem for Machine Learning, as always predicting this best algorithm might have a very high accuracy already, making it very hard to improve on. The authors mention two means of mitigating this problem. The training set can be *under-sampled*, where examples where the best overall algorithm performs best are deliberately omitted. Alternatively, the set can be *over-sampled* by artificially increasing the number of examples where another algorithm is better.

**Per-Algorithm Models.** A different approach is to learn performance models for the individual algorithms in the portfolio. The predicted performance of an algorithm on a problem can be compared to the predicted performance of the other portfolio algorithms and the selector can proceed based on this. The advantage of this approach is that it is easier to add and remove algorithms from the portfolio – instead of having to retrain the model for the entire portfolio, it suffices to train a model for the new algorithm or remove one of the trained models. Most approaches only rely on the order of predictions being correct. It does not matter if the prediction of the performance itself is wildly inaccurate as long as it is correct relative to the other predictions.

This is the approach that is implicitly assumed in Rice's framework. The prediction is the performance mapping $P(A, x)$ for an algorithm $A \in \mathcal{A}$ on a problem $x \in \mathcal{P}$. Models for each algorithm in the portfolio are used for example by [2,46,72,100,159].

A common way of doing this is to use regression to directly predict the performance of each algorithm. This is used by [64,72,95,123,159]. The performance of the algorithms in the portfolio is evaluated on a set of training problems, and a relationship between the characteristics of a problem and the performance of

an algorithm derived. This relationship usually has the form of a simple formula that is cheap to compute at runtime.

[130] on the other hand learn latent class models of unobserved variables to capture relationships between solvers, problems and run durations. Based on the predictions, the expected utility is computed and used to select an algorithm. [129] surveys sampling methods to estimate the cost of solving constraint problems. [148] models the behaviour of local search algorithms with Markov chains.

Another approach is to build statistical models of an algorithm's performance based on past observations. [149] use Bayesian belief propagation to predict the runtime of a particular algorithm on a particular problem. Bayesian inference is used to determine the class of a problem and the closest case in the knowledge base. A performance profile is extracted from that and used to estimate the runtime. The authors also propose an alternative approach that uses neural nets. [39,40] computes the expected gain for time bounds based on past success times. The computed values are used to choose the algorithm and the time bound for running it. [17] compare algorithm rankings based on different past performance statistics. Similarly, [94] maintain a ranking based on past performance. [27] propose a bandit problem model that governs the allocation of resources to each algorithm in the portfolio. [147] also use a bandit model, but furthermore evaluate a Q-learning approach, where in addition to bandit model rewards, the states of the system are taken into account. [56,57,155] use the past performance of algorithms to simulate the performance of different algorithm schedules and use statistical tests to select one of the schedules.

**Hierarchical Models.** There are some approaches that combine several models into a hierarchical performance model. There are two basic types of hierarchical models. One type predicts additional *properties of the problem* that cannot be measured directly or are not available without solving the problem. The other type makes *intermediate predictions* that do not inform Algorithm Selection directly, but rather the final predictions.

[156] use sparse multinomial logistic regression to predict whether a SAT problem instance is satisfiable and, based on that prediction, use a logistic regression model to predict the runtime of each algorithm in the portfolio. [64] also predict the satisfiability of a SAT instance and then choose an algorithm from a portfolio. Both report that being able to distinguish between satisfiable and unsatisfiable problems enables performance improvements. The satisfiability of a problem is a property that needs to be *predicted* in order to be useful for Algorithm Selection. If the property is *computed* (i.e. the problem is solved), there is no need to perform Algorithm Selection any more.

[55] use classifiers to first decide on the level of consistency a constraint propagator should achieve and then on the actual implementation of the propagator that achieves the selected level of consistency. A different publication that uses the same data set does not make this distinction however [87], suggesting that the performance benefits are not significant in practice.

[74] proposes a hierarchical model that has more than two levels – at the top, the decision is made whether to solve a given constraint problem as a constraint problem or convert it to SAT. At the second level, if the decision to convert to SAT has been made, the encoding for the transformation is chosen. At the third level, the constraint or SAT solver is chosen.

Such hierarchical models are only applicable in a limited number of scenarios, which explains the comparatively small amount of research into them. For many application domains, only a single property needs to be predicted and can be predicted without intermediate steps with sufficient accuracy. [83] proposes a hierarchical approach that is domain-independent. He uses the performance predictions of regression models as input to a classifier that decides which algorithm to choose and demonstrates performance improvements compared to selecting an algorithm directly based on the predicted performance. The idea is very similar to that of *stacking* in Machine Learning [152].

**Selection of Model Learner.** Apart from the different types of performance models, there are different Machine Learning algorithms that can be used to learn a particular kind of model. While most of the approaches mentioned here rely on a single way of doing this, some of the research compares different methods.

[159] mention that, in addition to the chosen ridge regression for predicting the runtime, they explored using lasso regression, support vector machines and Gaussian processes. They chose ridge regression not because it provided the most accurate predictions, but the best trade-off between accuracy and cost to make the prediction. [149] propose an approach that uses neural networks in addition to the Bayesian belief propagation approach they describe initially. [28] compare different decision tree learners, a Bayesian classifier, a nearest neighbour approach and a neural network. They chose the C4.5 decision tree inducer because even though it may be outperformed by a neural network, the learned trees are easily understandable by humans and may provide insight into the problem domain. [95] compare several versions of linear and non-linear regression. [75] report having explored support vector machine regression, multivariate adaptive regression splines (MARS) and lasso regression before deciding to use the linear regression approach of [95]. They also report experimental results with sequential Bayesian linear regression and Gaussian Process regression. [62,63] explore using decision trees, naïve Bayes rules, Bayesian networks and meta-learning techniques. They also chose the C4.5 decision tree inducer because it is one of the top performers and creates models that are easy to understand and quick to execute. [52] compare nearest neighbour classifiers, decision trees and statistical models. They show that a nearest neighbour classifier outperforms all the other approaches on their data sets.

[71] use decision tree ensembles and support vector machines. [12] investigate alternating decision trees and various forms of boosting, while [117] use decision trees, decision rules, logistic regression and nearest neighbour approaches. They do not explicitly choose one of these methods in the paper, but their Algorithm Selection system AQME uses a nearest neighbour classifier by default. [123] use

32 different Machine Learning algorithms to predict the runtime of algorithms and probability of success. They attempt to provide explanations for the performance of the methods they have chosen in [124]. [130] compare the performance of different latent class models. [55] evaluate the performance of 19 different Machine Learning classifiers on an Algorithm Selection problem in constraint programming. The investigation is extended to include more Machine Learning algorithms as well as different performance models and more problem domains in [85]. They identify several Machine Learning algorithms that show particularly good performance across different problem domains, namely linear regression and alternating decision trees. They do not consider issues such as how easy the models are to understand or how efficient they are to compute.

Only [52,55,63,71,85,117,130] quantify the differences in performance of the methods they used. The other comparisons give only qualitative evidence. Not all comparisons choose one of the approaches over the other or provide sufficient detail to enable the reader to do so. In cases where a particular technique is chosen, performance is often not the only selection criterion. In particular, the ability to understand a learned model plays a significant role.

### 4.2 Types of Predictions

The way of creating the performance model of a portfolio or its algorithms is not the only choice researchers face. In addition, there are different predictions the performance model can make to inform the decision of the selector of a subset of the portfolio algorithms. The type of decision is closely related to the learned performance model however. The prediction can be a single categorical value – the algorithm to choose. This type of prediction is usually the output of per-portfolio models and used for example in [28,54,61,108,117]. The advantage of this simple prediction is that it determines the choice of algorithm without the need to compare different predictions or derive further quantities. One of its biggest disadvantages however is that there is no flexibility in the way the system runs or even the ability to monitor the execution for unexpected behaviour.

A different approach is to predict the runtime of the individual algorithms in the portfolio. This requires per-algorithm models. For example [70,113,130] do this. [159] do not predict the runtime itself, but the logarithm of the runtime. They note that,

> "In our experience, we have found this log transformation of runtime to be very important due to the large variation in runtimes for hard combinatorial problems."

[85] also compare predicting the runtime itself and the log thereof, but find no significant difference between the two. [83] however also reports better results with the logarithm.

[2] estimate the runtime by proxy by predicting the number of constraint checks. [100] estimate the runtime by predicting the number of search nodes to explore and the time per node. [90] talk of the *cost* of selecting a particular

algorithm, which is equal to the time it takes to solve the problem. [107] uses the *utility* of a choice to make his decision. The utility is an abstract measure of the "goodness" of an algorithm that is adapted dynamically. [142] use the *value of information* of selecting an algorithm, defined as the amount of time saved by making this choice. [160] predict the *penalized average runtime score*, a measure that combines runtime with possible timeouts. This approach aims to provide more realistic performance predictions when runtimes are capped.

More complex predictions can be made, too. In most cases, these are made by combining simple predictions such as the runtime performance. [17,94,135] produce rankings of the portfolio algorithms. [85] use statistical relational learning to directly predict the ranking instead of deriving it from other predictions. [46,49,72,110,122] predict resource allocations for the algorithms in the portfolios. [14,52,99] consider selecting the most appropriate formulation of a constraint problem. [8,19,131,151] select algorithms and data structures to be used in a software system.

Some types of predictions require online approaches that make decisions during search. [7,15,23,125] predict when to switch the algorithm used to solve a problem. [70] predict whether to restart an algorithm. [90,91] predict the cost to solve a sub-problem. However, most online approaches make predictions that can also be used in offline settings, such as the best algorithm to proceed with.

The primary selection criteria and prediction for [135] and [94] is the quality of the solution an algorithm produces rather than the time it takes the algorithm to find that solution. In addition to the primary selection criteria, a number of approaches predict secondary criteria. [40,72,123] predict the probability of success for each algorithm. [149] predict the quality of a solution.

In Rice's model, the prediction of an Algorithm Selection system is the performance $p \in \mathcal{R}^n$ of an algorithm. This abstract notion does not rely on time and is applicable to many approaches. It does not fit techniques that predict the portfolio algorithm to choose or more complex measures such as a schedule however. As Rice developed his approach long before the advent of algorithm portfolios, it should not be surprising that the notion of the performance of individual algorithms as opposed to sets of algorithms dominates. The model is sufficiently general to be able to accommodate algorithm portfolios with only minor modifications to the overall framework however.

## 5 Features

The different types of performance models described in the previous sections usually use features to inform their predictions. Features are an integral part of systems that do Machine Learning. They characterise the inputs, such as the problem to be solved or the algorithm employed to solve it, and facilitate learning the relationship between the inputs and the outputs, such as the time it will take the algorithm to solve the problem. In Rice's model, features $f(x)$ for a particular problem $x$ are extracted from the feature space $\mathcal{F}$.

The selection of the most suitable features is an important part of the design of Algorithm Selection systems. There are different types of features researchers

can use and different ways of computing these. They can be categorised according to two main criteria.

First, they can be categorised according to how much background knowledge a researcher needs to have to be able to use them. Features that require no or very little knowledge of the application domain are usually very general and can be applied to new Algorithm Selection problems with little or no modification. Features that are specific to a domain on the other hand may require the researcher building the Algorithm Selection system to have a thorough understanding of the domain. These features usually cannot be applied to other domains, as they may be non-existent or uninformative in different contexts.

The second way of distinguishing different classes of features is according to when and how they are computed. Features can be computed *statically*, i.e. before the search process starts, or *dynamically*, i.e. during search. These two categories roughly align with the offline and online approaches to portfolio problem solving described in Sect. 3.

[132] present a survey that focuses on what features can be used for Algorithm Selection. This chapter categorises the features used in the literature.

### 5.1 Low and High-Knowledge Features

In some cases, researchers use a large number of features that are specific to the particular problem domain they are interested in, but there are also publications that only use a single, general feature – the performance of a particular algorithm on past problems. [27,49,113,130,138], to name but a few examples, use this approach to build statistical performance models of the algorithms in their portfolios. The underlying assumption is that all problems are similar with respect to the relative performance of the algorithms in the portfolio – the algorithm that has done best in the past has the highest chance of performing best in the future.

Approaches that build runtime distribution models for the portfolio algorithms usually do not select a single algorithm for solving a problem, but rather use the distributions to compute resource allocations for the individual portfolio algorithms. The time allocated to each algorithm is proportional to its past performance.

Other sources of features that are not specific to a particular problem domain are more fine-grained measures of past performance or measures that characterise the behaviour of an algorithm during search. [93] for example determines whether a search step performed by a particular algorithm is good, i.e. leading towards a solution, or bad, i.e. straying from the path to a solution if the solution is known or revisiting an earlier search state if the solution is not known. [57,59] use the runtime distributions of algorithms over the size of a problem, as measured by the number of backtracks. [40] uses the past success times of an algorithm as candidate time bounds on new problems. [17] do not consider the runtime, but the error rate of algorithms. [56] use both computation time and solution quality.

[11,23,24] evaluate the performance also during search. They explicitly focus on features that do not require a lot of domain knowledge. [11] note that,

> "While existing algorithm selection techniques have shown impressive results, their knowledge-intensive nature means that domain and algorithm expertise is necessary to develop the models. The overall requirement for expertise has not been reduced: it has been shifted from algorithm selection to predictive model building."

They do, like several other approaches, assume *anytime* algorithms – after search has started, the algorithm is able to return the best solution found so far at any time. The features are based on how search progresses and how the quality of solutions is improved by algorithms. While this does not require any knowledge about the application domain, it is not applicable in cases when only a single solution is sought.

Most approaches learn models for the performance on particular problems and do not use past performance as a feature, but to inform the prediction to be made. Considering problem features facilitates a much more nuanced approach than a broad-brush general performance model. This is the classic supervised Machine Learning approach – given the correct prediction derived from the behaviour on a set of training problems, learn a model that enables to make this prediction.

The features that are considered to learn the model are specific to the problem domain or even a subset of the problem domain to varying extents. For combinatorial search problems, the most commonly used basic features include,

- the number of variables,
- properties of the variable domains, i.e. the list of possible assignments,
- the number of clauses in SAT, the number of constraints in constraint problems, the number of goals in planning,
- the number of clauses/constraints/goals of a particular type (for example the number of `alldifferent` constraints, [55]),
- ratios of several of the above features and summary statistics.

Such features are used for example in [72,110,117,149,159].

Other sources of features include the generator that produced the problem to be solved [70], the runtime environment [7], structures derived from the problem such as the primal graph of a constraint problem [51,54,61], specific parts of the problem model such as variables [35], the algorithms in the portfolio themselves [71] or the domain of the problem to be solved [22,56] rely on the problem domain as the only problem-specific feature and select based on past performance data for the particular domain. [10] consider not only the values of properties of a problem, but the changes of those values while the problem is being solved. [131] consider features of abstract representations of the algorithms. [163,164] use features that represent technical details of the behaviour of an algorithm on a problem, such as the type of computations done in a loop. [74] consider features not only of the instance being solved, but also of alternative encodings of the same instance.

Most approaches use features that are applicable to all problems of the application domain they are considering. However, [70] use features that are not only

specific to their application domain, but also to the specific family of problems they are tackling, such as the variance of properties of variables in different columns of Latin squares. They note that,

> "... the inclusion of such domain-specific features was important in learning strongly predictive models."

### 5.2 Static and Dynamic Features

In most cases, the approaches that use a large number of domain-specific features compute them *offline*, i.e. before the solution process starts (cf. Sect. 3.2). Examples of publications that only use such static features are [61,95,117].

An implication of using static features is that the decisions of the Algorithm Selection system are only informed by the performance of the algorithms on past problems. Only dynamic features allow to take the performance on the current problem into account. This has the advantage that remedial actions can be taken if the problem is unlike anything seen previously or the predictions are wildly inaccurate for another reason.

A more flexible approach than to rely purely on static features is to incorporate features that can be determined statically, but try to estimate the performance on the current problem. Such features are computed by probing the search space. This approach relies on the performance probes being sufficiently representative of the entire problem and sufficiently equal across the different evaluated algorithms. If an algorithm is evaluated on a part of the search space that is much easier or harder than the rest, a misleading impression of its true performance may result.

Examples of systems that combine static features of the problem to be solved with features derived from probing the search space are [54,110,159]. There are also approaches that use only probing features. We term this *semi-static* feature computation because it happens before the actual solving of the problem starts, but parts of the search space are explored during feature extraction. Examples include [2,11,100].

The idea of probing the search space is related to *landmarking* [116], where the performance of a set of initial algorithms (the *landmarkers*) is linked to the performance of the set of algorithms to select from. The main consideration when using this technique is to select landmarkers that are computationally cheap. Therefore, they are usually versions of the portfolio algorithms that have either been simplified or are run only on a subset of the data the selected algorithm will run on.

While the work done during probing explores part of the search space and could be used to speed search up subsequently by avoiding to revisit known areas, almost no research has been done into this. [11] run all algorithms in their (small) portfolio on a problem for a fixed time and select the one that has made the best progress. The chosen algorithm resumes its earlier work, but no attempt is made to avoid duplicating work done by the other algorithms.

To the best of our knowledge, there exist no systems that attempt to avoid redoing work performed by a different algorithm during the probing stage.

For successful systems, the main source of performance improvements is the selection of the right algorithm using the features computed through probing. As the time to compute the features is usually small compared to the runtime improvements achieved by Algorithm Selection, using the results of probing during search to avoid duplicating work does not have the potential to achieve large additional performance improvements.

The third way of computing features is to do so *online*, i.e. while search is taking place. These dynamic features are computed by an execution monitor that adapts or changes the algorithm during search based on its performance. Approaches that rely purely on dynamic features are for example [15,107,136].

There are many different features that can be computed during search. [105] determines how closely a generated heuristic approximates a generic target heuristic by checking the heuristic choices at random points during search. He selects the one with the closest match. Similarly, [107] learn how to select heuristics during the search process based on their performance. [7] use an agent-based model that rewards good actions and punishes bad actions based on computation time. [89] follow a very similar approach that also takes success or failure into account.

[23,24] monitor the solution quality during search. They decide whether to switch the current algorithm based on this by changing the allocation of resources. [150] monitor a feature that is specific to their application domain, the distribution of clause weights in SAT, during search and use it to decide whether to switch a heuristic. [136] monitors propagation events in a constraint solver to a similar aim. [25] evaluate the performance of candidate algorithms in terms of number of calls to a specific high-level procedure. They note that in contrast to using the runtime, their approach is machine-independent.

### 5.3 Feature Selection

The features used for learning the Algorithm Selection model are crucial to its success. Uninformative features might prevent the model learner from recognising the real relation between problem and performance or the most important feature might be missing. Many researchers have recognised this problem.

[72] manually select the most important features. They furthermore take the unique approach of learning one model per feature for predicting the probability of success and combine the predictions of the models. [95,159] perform automatic feature selection by greedily adding features to an initially empty set. In addition to the basic features, they also use the pairwise products of the features. [117] also perform automatic greedy feature selection, but do not add the pairwise products. [85] automatically select the most important subset of the original set of features, but conclude that in practice the performance improvement compared to using all features is not significant. [151] use genetic algorithms to determine the importance of the individual features. [115] evaluate subsets of the features they use and learn weights for each of them. [124] consider using a

single feature and automatic selection of a subset of all features. [63,88] also use techniques for automatically determining the most predictive subset of features. [83] compares the performance of ten different sets of features.

It is not only important to use informative features, but also features that are cheap to compute. If the cost of computing the features and making the decision is too high, the performance improvement from selecting the best algorithm might be eroded. [160] predict the feature computation time for a given problem and fall back to a default selection if it is too high to avoid this problem. They also limit the computation time for the most expensive features as well as the total time allowed to compute features. [13] consider the computational complexity of calculating problem features when selecting the features to use. They show that while achieving comparable accuracy to the full set of features, the subset of features selected by their method is significantly cheaper to compute. [54] explicitly exclude features that are expensive to compute.

## 6 Application Domains

The approaches for solving the Algorithm Selection Problem that have been surveyed here are usually not specific to a particular application domain, within combinatorial search problems or otherwise. Nevertheless this survey would not be complete without a brief exposition of the various contexts in which Algorithm Selection techniques have been applied.

Over the years, Algorithm Selection systems have been used in many different application domains. These range from Mathematics, e.g. differential equations [82,149], linear algebra [29] and linear systems [12,89], to the selection of algorithms and data structures in software design [19,21,131,151]. A very common application domain are combinatorial search problems such as SAT [91, 130,159], constraints [36,105,110], Mixed Integer Programming [161], Quantified Boolean Formulae [118,137], planning [22,38,72], scheduling [10,11,27], combinatorial auctions [46,51,95], Answer Set Programming [53,67], the Travelling Salesperson Problem [41,86], graph colouring [106] and general search algorithms [28,93,100].

Other domains include Machine Learning [94,135], the most probable explanation problem [63], parallel reduction algorithms [163,164] and simulation [37,147]. It should be noted that a significant part of Machine Learning research is concerned with developing Algorithm Selection techniques; the publications listed in this paragraph are the most relevant that use the specific techniques and framework surveyed here.

Some publications consider more than one application domain. [137] choose the best algorithm for Quantified Boolean Formulae and combinatorial auctions. [2,88] look at SAT and constraints. [59] consider SAT and Mixed Integer Programming. In addition to these two domains, [81] also investigate set covering problems. [140] apply their approach to SAT, Integer Programming and planning. [48,83,85] compare the performance across Algorithm Selection problems from constraints, Quantified Boolean Formulae and SAT.

In most cases, researchers take some steps to adapt their approaches to the application domain. This is usually done by using domain-specific features, such as the number of constraints and variables in constraint programming. In principle, this is not a limitation of the proposed techniques as those features can be exchanged for ones that are applicable in other application domains. While the overall approach remains valid, the question of whether the performance would be acceptable arises. [85] investigate how specific techniques perform across several domains with the aim of selecting the one with the best overall performance. There are approaches that have been tailored to a specific application domain to such an extent that the technique cannot be used for other applications. This is the case for example in the case of hierarchical models for SAT [64,156].

## 7 Summary

Over the years, there have been many approaches to solving the Algorithm Selection Problem. Especially in Artificial Intelligence and for combinatorial search problems, researchers have recognised that using Algorithm Selection techniques can provide significant performance improvements with relatively little effort. Most of the time, the approaches involve some kind of Machine Learning that attempts to learn the relation between problems and the performance of algorithms automatically. This is not a surprise, as the relationship between an algorithm and its performance is often complex and hard to describe formally. In many cases, even the designer of an algorithm does not have a general model of its performance.

Despite the theoretical difficulty of Algorithm Selection, dozens of systems have demonstrated that it can be done in practice with great success. In some sense, this mirrors achievements in other areas of Artificial Intelligence. Satisfiability is formally a problem that cannot be solved efficiently, yet researchers have come up with ways of solving very large instances of satisfiability problems with very few resources. Similarly, some Algorithm Selection systems have come very close to always choosing the best algorithm.

This survey presented an overview of the Algorithm Selection research that has been done to date with a focus on combinatorial search problems. A categorisation of the different approaches with respect to fundamental criteria that determine Algorithm Selection systems in practice was introduced. This categorisation abstracts from many of the low level details and additional considerations that are presented in most publications to give a clear view of the underlying principles. We furthermore gave details of the many different ways that can be used to tackle Algorithm Selection and the many techniques that have been used to solve it in practice.

On a high level, the approaches surveyed here can be summarised as follows.

– Algorithms are chosen from portfolios, which can be statically constructed or dynamically augmented with newly constructed algorithms as problems are being solved. Portfolios can be engineered such that the algorithms in it complement each other (i.e. are as diverse as possible), by automatically

tuning algorithms on a set of training problems or by using a set of algorithms from the literature or competitions. Dynamic portfolios can be composed of algorithmic building blocks that are combined into complete algorithms by the selection system. Compared to tuning the parameters of algorithms, the added difficulty is that not all combinations of building blocks may be valid.
- A single algorithm can be selected from a portfolio to solve a problem to completion or a set of larger size can be selected that is run in parallel or according to a schedule. Another approach is to select a single algorithm to start with and then decide if and when to switch to another algorithm. Some approaches always select the entire portfolio and vary the resource allocation to the algorithms.
- Algorithm Selection can happen offline, without any interaction with the Algorithm Selection system after solving starts, or online. Some approaches monitor the performance of the selected algorithm and take action if it does not conform to the expectations or some other criteria. Others repeat the selection process at specific points during the search (e.g. every node in the search tree), skew a computed schedule towards the best performers or decide whether to restart stochastic algorithms.
- Performance can be modelled and predicted either for a portfolio as a whole (i.e. the prediction is the best algorithm) or for each algorithm independently (i.e. the prediction is the performance). A few approaches use hierarchical models that make a series of predictions to facilitate selection. Some publications make secondary predictions (e.g. the quality of a solution) that are taken into account when selecting the most suitable algorithm, while others make predictions that the desired output is derived from instead of predicting it directly. The performance models are usually learned automatically using Machine Learning, but a few approaches use hand-crafted models and rules. Models can be learned from separate training data or incrementally while a problem is being solved.
- Learning and using performance models is facilitated by features of the algorithms, problems or runtime environment. Features can be domain-independent or specific to a particular set of problems. Similarly, features can be computed by inspecting the problem before solving or while it is being solved. The use of feature selection techniques that automatically determine the most important and relevant features is quite common.

Given the amount of relevant literature, it is infeasible to discuss every approach in detail. The scope of this survey is necessarily limited to the detailed description of high-level details and a summary overview of low-level traits. Work in related areas that is not immediately relevant to Algorithm Selection for combinatorial search problems has been pointed to, but cannot be explored in more detail.

The proliferation of different approaches, application domains and data sets has stimulated the creation of a common data format and benchmark repository for algorithm selection problems, http://aslib.net. It provides a starting point for researchers wishing to compare their new approach to existing approaches.

A tabular summary of the literature organised according to the criteria introduced here can be found at http://larskotthoff.github.io/assurvey/. This table is updated continuously.

**Acknowledgements.** Ian Miguel and Ian Gent provided valuable feedback that helped shape this chapter. We also thank the anonymous reviewers of a previous version of this chapter whose detailed comments helped to greatly improve it. This work was supported by an EPSRC doctoral prize and EU FP7 FET project ICON. A shorter version of this chapter has appeared in AI Magazine [84].

## References

1. Aha, D.W.: Generalizing from case studies: a case study. In: Proceedings of the 9th International Workshop on Machine Learning, pp. 1–10. Morgan Kaufmann Publishers Inc, San Francisco (1992)
2. Allen, J.A., Minton, S.: Selecting the right heuristic algorithm: runtime performance predictors. In: McCalla, G. (ed.) AI 1996. LNCS, vol. 1081, pp. 41–53. Springer, Heidelberg (1996). doi:10.1007/3-540-61291-2_40
3. Amadini, R., Gabbrielli, M., Mauro, J.: SUNNY: a lazy portfolio approach for constraint solving. TPLP **14**(4–5), 509–524 (2014)
4. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: PetaBricks: a language and compiler for algorithmic choice. SIGPLAN Not. **44**(6), 38–49 (2009)
5. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 142–157. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04244-7_14
6. Arbelaez, A., Hamadi, Y., Sebag, M.: Online heuristic selection in constraint programming. In: Symposium on Combinatorial Search (2009)
7. Armstrong, W., Christen, P., McCreath, E., Rendell, A.P.: Dynamic algorithm selection using reinforcement learning. In: International Workshop on Integrating AI and Data Mining, pp. 18–25, December 2006
8. Balasubramaniam, D., Gent, I.P., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: An automated approach to generating efficient constraint solvers. In: 34th International Conference on Software Engineering, pp. 661–671, June 2012
9. Bauer, E., Kohavi, R.: An empirical comparison of voting classification algorithms: bagging, boosting, and variants. Mach. Learn. **36**(1–2), 105–139 (1999)
10. Beck, J.C., Fox, M.S.: Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. Artif. Intell. **117**(1), 31–81 (2000)
11. Beck, J.C., Freuder, E.C.: Simple rules for low-knowledge algorithm selection. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 50–64. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24664-0_4
12. Bhowmick, S., Eijkhout, V., Freund, Y., Fuentes, E., Keyes, D.: Application of machine learning in selecting sparse linear solvers. Technical report, Columbia University (2006)
13. Bhowmick, S., Toth, B., Raghavan, P.: Towards low-cost, high-accuracy classifiers for linear solver selection. In: Allen, G., Nabrzyski, J., Seidel, E., Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 463–472. Springer, Heidelberg (2009). doi:10.1007/978-3-642-01970-8_45

14. Borrett, J.E., Tsang, E.P.K.: A context for constraint satisfaction problem formulation selection. Constraints **6**(4), 299–327 (2001)
15. Borrett, J.E., Tsang, E.P.K., Walsh, N.R.: Adaptive constraint satisfaction: The quickest first principle. In: ECAI, pp. 160–164 (1996)
16. Bougeret, M., Dutot, P., Goldman, A., Ngoko, Y., Trystram, D.: Combining multiple heuristics on discrete resources. In: IEEE International Symposium on Parallel and Distributed Processing, pp. 1–8. IEEE Computer Society, Washington, DC (2009)
17. Brazdil, P.B., Soares, C.: A comparison of ranking methods for classification algorithm selection. In: López de Mántaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 63–75. Springer, Heidelberg (2000). doi:10.1007/3-540-45164-1_8
18. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
19. Brewer, E.A.: High-level optimization via automated statistical modeling. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP 1995, pp. 80–91. ACM, New York (1995)
20. Brodley, C.E.: Addressing the selective superiority problem: automatic algorithm/model class selection. In: ICML, pp. 17–24 (1993)
21. Cahill, E.: Knowledge-based algorithm construction for real-world engineering PDEs. Math. Comput. Simul. **36**(4–6), 389–400 (1994)
22. Carbonell, J., Etzioni, O., Gil, Y., Joseph, R., Knoblock, C., Minton, S., Veloso, M.: PRODIGY: an integrated architecture for planning and learning. SIGART Bull. **2**, 51–55 (1991)
23. Carchrae, T., Beck, J.C.: Low-knowledge algorithm control. In: AAAI, pp. 49–54 (2004)
24. Carchrae, T., Beck, J.C.: Applying machine learning to Low-knowledge control of optimization algorithms. Comput. Intell. **21**(4), 372–387 (2005)
25. Caseau, Y., Laburthe, F., Silverstein, G.: A meta-heuristic factory for vehicle routing problems. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 144–158. Springer, Heidelberg (1999). doi:10.1007/978-3-540-48085-3_11
26. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: 12th International Joint Conference on Artificial Intelligence, pp. 331–337. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA (1991)
27. Cicirello, V.A., Smith, S.F.: The max k-armed bandit: a new model of exploration applied to search heuristic selection. In: Proceedings of the 20th National Conference on Artificial Intelligence, pp. 1355–1361. AAAI Press (2005)
28. Cook, D.J., Varnell, R.C.: Maximizing the benefits of parallel search using machine learning. In: Proceedings of the 14th National Conference on Artificial Intelligence, pp. 559–564. AAAI Press (1997)
29. Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R.C., Yelick, K.: Self-adapting linear algebra algorithms and software. Proc. IEEE **93**(2), 293–312 (2005)
30. Dietterich, T.G.: Ensemble methods in machine learning. In: Kittler, J., Roli, F. (eds.) MCS 2000. LNCS, vol. 1857, pp. 1–15. Springer, Heidelberg (2000). doi:10.1007/3-540-45014-9_1
31. Domingos, P.: How to get a free lunch: a simple cost model for machine learning applications. In: AAAI98/ICML98 Workshop on the Methodology of Applying Machine Learning, pp. 1–7. AAAI Press (1998)
32. Domshlak, C., Karpas, E., Markovitch, S.: To max or not to max: online learning for speeding up optimal planning. In: AAAI (2010)

33. Elsayed, S.A.M., Michel, L.: Synthesis of search algorithms from high-level CP models. In: Proceedings of the 9th International Workshop on Constraint Modelling and Reformulation, September 2010
34. Elsayed, S.A.M., Michel, L.: Synthesis of search algorithms from high-level CP models. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 256–270. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23786-7_21
35. Epstein, S.L., Freuder, E.C.: Collaborative learning for constraint solving. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 46–60. Springer, Heidelberg (2001). doi:10.1007/3-540-45578-7_4
36. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 525–540. Springer, Heidelberg (2002). doi:10.1007/3-540-46135-3_35
37. Ewald, R., Schulz, R., Uhrmacher, A.M.: Selecting simulation algorithm portfolios by genetic algorithms. In: IEEE Workshop on Principles of Advanced and Distributed Simulation PADS 2010, IEEE Computer Society, Washington, DC (2010)
38. Fawcett, C., Vallati, M., Hutter, F., Hoffmann, J., Hoos, H., Leyton-Brown, K.: Improved features for runtime prediction of domain-independent planners. In: ICAPS (2014)
39. Fink, E.: Statistical selection among problem-solving methods. Technical report CMU-CS-97-101. Carnegie Mellon University (1997)
40. Fink, E.: How to solve it automatically: selection among problem-solving methods. In: Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems, pp. 128–136. AAAI Press (1998)
41. Fukunaga, A.S.: Genetic algorithm portfolios. IEEE Congr. Evol. Comput. **2**, 1304–1311 (2000)
42. Fukunaga, A.S.: Automated discovery of composite SAT variable-selection heuristics. In: 18th National Conference on Artificial Intelligence, pp. 641–648. American Association for Artificial Intelligence, Menlo Park (2002)
43. Fukunaga, A.S.: Automated discovery of local search heuristics for satisfiability testing. Evol. Comput. **16**, 31–61 (2008)
44. Gagliolo, M., Schmidhuber, J.: A neural network model for inter-problem adaptive online time allocation. In: Duch, W., Kacprzyk, J., Oja, E., Zadrożny, S. (eds.) ICANN 2005. LNCS, vol. 3697, pp. 7–12. Springer, Heidelberg (2005). doi:10.1007/11550907_2
45. Gagliolo, M., Schmidhuber, J.: Impact of censored sampling on the performance of restart strategies. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 167–181. Springer, Heidelberg (2006). doi:10.1007/11889205_14
46. Gagliolo, M., Schmidhuber, J.: Learning dynamic algorithm portfolios. Ann. Math. Artif. Intell. **47**(3–4), 295–328 (2006)
47. Gagliolo, M., Schmidhuber, J.: Towards distributed algorithm portfolios. In: Corchado, J.M., Rodríguez, S., Llinas, J., Molina, J.M. (eds.) Advances in Soft Computing. AINSC, vol. 50, pp. 634–643. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85863-8_75
48. Gagliolo, M., Schmidhuber, J.: Algorithm portfolio selection as a bandit problem with unbounded losses. Ann. Math. Artif. Intell. **61**(2), 49–86 (2011)
49. Gagliolo, M., Zhumatiy, V., Schmidhuber, J.: Adaptive online time allocation to search algorithms. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) ECML 2004. LNCS (LNAI), vol. 3201, pp. 134–143. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30115-8_15

50. Garrido, P., Riff, M.: DVRP: a hard dynamic combinatorial optimisation problem tackled by an evolutionary hyper-heuristic. J. Heuristics **16**, 795–834 (2010)
51. Gebruers, C., Guerri, A., Hnich, B., Milano, M.: Making choices using structure at the instance level within a case based reasoning framework. In: CPAIOR, pp. 380–386 (2004)
52. Gebruers, C., Hnich, B., Bridge, D., Freuder, E.: Using CBR to select solution strategies in constraint programming. In: Proceedings of ICCBR 2005, pp. 222–236 (2005)
53. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A portfolio solver for answer set programming: preliminary report. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 352–357. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20895-9_40
54. Gent, I., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N., Nightingale, P., Petrie, K.: Learning when to use lazy learning in constraint solving. In: 19th European Conference on Artificial Intelligence, pp. 873–878, August 2010
55. Gent, I., Kotthoff, L., Miguel, I., Nightingale, P.: Machine learning for constraint solver design - a case study for the alldifferent constraint. In: 3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS), pp. 13–25 (2010)
56. Gerevini, A.E., Saetti, A., Vallati, M.: An automatically configurable portfolio-based planner with macro-actions: PbP. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling, pp. 350–353 (2009)
57. Gomes, C.P., Selman, B.: Algorithm portfolio design: theory vs. practice. In: UAI, pp. 190–197 (1997)
58. Gomes, C.P., Selman, B.: Practical aspects of algorithm portfolio design. In: Proceedings of 3rd ILOG International Users Meeting (1997)
59. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
60. Gratch, J., DeJong, G.: COMPOSER: a probabilistic solution to the utility problem in speed-up learning. In: AAAI, pp. 235–240 (1992)
61. Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In: ECAI, pp. 475–479 (2004)
62. Guo, H.: Algorithm selection for sorting and probabilistic inference: a machine learning-based approach. Ph.D. thesis, Kansas State University (2003)
63. Guo, H., Hsu, W.H.: A learning-based algorithm selection meta-reasoner for the real-time MPE problem. In: Webb, G.I., Yu, X. (eds.) AI 2004. LNCS (LNAI), vol. 3339, pp. 307–318. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30549-1_28
64. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 312–325. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_30
65. Hogg, T., Huberman, B.A., Williams, C.P.: Phase transitions and the search problem. Artif. Intell. **81**(1–2), 1–15 (1996)
66. Hong, L., Page, S.E.: Groups of diverse problem solvers can outperform groups of high-ability problem solvers. Proc. Natl. Acad. Sci. U.S.A. **101**(46), 16385–16389 (2004)
67. Hoos, H., Lindauer, M., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. TPLP **14**(4–5), 569–585 (2014)
68. Hoos, H.H.: Programming by optimization. Commun. ACM **55**(2), 70–80 (2012)

69. Hoos, H.H., Kaminski, R., Lindauer, M., Schaub, T.: aspeed: Solver scheduling via answer set programming. Theory Pract. Logic Program. FirstView **15**, 1–26 (2014)
70. Horvitz, E., Ruan, Y., Gomes, C.P., Kautz, H.A., Selman, B., Chickering, D.M.: A Bayesian approach to tackling hard computational problems. In: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, pp. 235–244. Morgan Kaufmann Publishers Inc., San Francisco (2001)
71. Hough, P.D., Williams, P.J.: Modern machine learning for automatic optimization algorithm selection. In: Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop, November 2006
72. Howe, A.E., Dahlman, E., Hansen, C., Scheetz, M., Mayrhauser, A.: Exploiting competitive planner performance. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS (LNAI), vol. 1809, pp. 62–72. Springer, Heidelberg (2000). doi:10.1007/10720246_5
73. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296), 51–54 (1997)
74. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: a hierarchical portfolio of solvers and transformations. In: CPAIOR, May 2014
75. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 213–228. Springer, Heidelberg (2006). doi:10.1007/11889205_17
76. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25566-3_40
77. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Parallel algorithm configuration. In: Hamadi, Y., Schoenauer, M. (eds.) LION. LNCS, vol. 7219, pp. 55–70. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34413-8_5
78. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Int. Res. **36**(1), 267–306 (2009)
79. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 1152–1157. AAAI Press (2007)
80. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: 17th International Conference on Principles and Practice of Constraint Programming, pp. 454–469 (2011)
81. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC instance-specific algorithm configuration. In: 19th European Conference on Artificial Intelligence, pp. 751–756. IOS Press (2010)
82. Kamel, M.S., Enright, W.H., Ma, K.S.: ODEXPERT: an expert system to select numerical solvers for initial value ODE systems. ACM Trans. Math. Softw. **19**(1), 44–62 (1993)
83. Kotthoff, L.: Hybrid regression-classification models for algorithm selection. In: 20th European Conference on Artificial Intelligence, pp. 480–485, August 2012
84. Kotthoff, L.: Algorithm selection for combinatorial search problems: a survey. AI Mag. **35**(3), 48–60 (2014)
85. Kotthoff, L., Gent, I.P., Miguel, I.: An evaluation of machine learning in algorithm selection for search problems. AI Commun. **25**(3), 257–270 (2012)

86. Kotthoff, L., Kerschke, P., Hoos, H., Trautmann, H.: Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 202–217. Springer, Heidelberg (2015). doi:10.1007/978-3-319-19084-6_18
87. Kotthoff, L., Miguel, I., Nightingale, P.: Ensemble classification for constraint solver configuration. In: 16th International Conference on Principles and Practices of Constraint Programming, pp. 321–329, September 2010
88. Kroer, C., Malitsky, Y.: Feature filtering for Instance-Specific algorithm configuration. In: Proceedings of the 23rd International Conference on Tools with Artificial Intelligence (2011)
89. Kuefler, E., Chen, T.-Y.: On using reinforcement learning to solve sparse linear systems. In: Bubak, M., Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008. LNCS, vol. 5101, pp. 955–964. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69384-0_100
90. Lagoudakis, M.G., Littman, M.L.: Algorithm selection using reinforcement learning. In: Proceedings of the 17th International Conference on Machine Learning, pp. 511–518. Morgan Kaufmann Publishers Inc., San Francisco (2000)
91. Lagoudakis, M.G., Littman, M.L.: Learning to select branching rules in the DPLL procedure for satisfiability. In: LICS/SAT, pp. 344–359 (2001)
92. Langley, P.: Learning effective search heuristics. In: IJCAI, pp. 419–421 (1983)
93. Langley, P.: Learning search strategies through discrimination. Int. J. Man-Mach. Stud. **18**, 513–541 (1983)
94. Leite, R., Brazdil, P., Vanschoren, J., Queiros, F.: Using active testing and meta-level information for selection of classification algorithms. In: 3rd PlanLearn Workshop, August 2010
95. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In: Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 556–572. Springer, Heidelberg (2002). doi:10.1007/3-540-46135-3_37
96. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: methodology and a case study on combinatorial auctions. J. ACM **56**, 1–52 (2009)
97. Lindauer, M., Hoos, H., Hutter, F.: From sequential algorithm selection to parallel portfolio selection. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 1–16. Springer, Heidelberg (2015). doi:10.1007/978-3-319-19084-6_1
98. Lindauer, M., Hoos, H.H., Hutter, F., Schaub, T.: AutoFolio: algorithm configuration for algorithm selection. In: Twenty-Ninth AAAI Workshops on Artificial Intelligence, January 2015
99. Little, J., Gebruers, C., Bridge, D., Freuder, E.: Capturing constraint programming experience: a case-based approach. In: Modref (2002)
100. Lobjois, L., Lemaître, M.: Branch and bound algorithm selection by performance prediction. In: Proceedings of the 15th National/10th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, pp. 353–358. American Association for Artificial Intelligence, Menlo Park (1998)
101. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Non-model-based algorithm portfolios for SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 369–370. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21581-0_33
102. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm portfolios based on cost-sensitive hierarchical clustering. In: IJCAI, August 2013

103. Minton, S.: An analytic learning system for specializing heuristics. In: Proceedings of the 13th International Joint Conference on Artifical Intelligence IJCAI 1993, pp. 922–928. Morgan Kaufmann Publishers Inc., San Francisco (1993)
104. Minton, S.: Integrating heuristics for constraint satisfaction problems: a case study. In: Proceedings of the 11th National Conference on Artificial Intelligence, pp. 120–126. AAAI (1993)
105. Minton, S.: Automatically configuring constraint satisfaction programs: a case study. Constraints **1**, 7–43 (1996)
106. Musliu, N., Schwengerer, M.: Algorithm selection for the graph coloring problem. In: Nicosia, G., Pardalos, P. (eds.) LION 2013. LNCS, vol. 7997, pp. 389–403. Springer, Heidelberg (2013). doi:10.1007/978-3-642-44973-4_42
107. Nareyek, A.: Choosing search heuristics by non-stationary reinforcement learning. In: Nareyek, A. (ed.) Metaheuristics: Computer Decision-Making. Applied Optimization, vol. 86, pp. 523–544. Kluwer Academic Publishers, New York (2001)
108. Nikolić, M., Marić, F., Janičić, P.: Instance-based selection of policies for SAT solvers. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 326–340. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_31
109. Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding random SAT: beyond the clauses-to-variables ratio. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 438–452. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30201-8_33
110. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
111. Opitz, D., Maclin, R.: Popular ensemble methods: an empirical study. J. Artif. Intell. Res. **11**, 169–198 (1999)
112. Paparrizou, A., Stergiou, K.: Evaluating simple fully automated heuristics for adaptive constraint propagation. In: ICTAI (2012)
113. Petrik, M.: Statistically optimal combination of algorithms. In: Local Proceedings of SOFSEM 2005 (2005)
114. Petrik, M., Zilberstein, S.: Learning parallel portfolios of algorithms. Ann. Math. Artif. Intell. **48**(1–2), 85–106 (2006)
115. Petrovic, S., Qu, R.: Case-based reasoning as a heuristic selector in hyper-heuristic for course timetabling problems. In: KES, pp. 336–340 (2002)
116. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.G.: Meta-Learning by landmarking various learning algorithms. In: 17th International Conference on Machine Learning ICML 2000, pp. 743–750, Morgan Kaufmann Publishers Inc., San Francisco (2000)
117. Pulina, L., Tacchella, A.: A multi-engine solver for quantified Boolean formulas. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 574–589. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74970-7_41
118. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified boolean formulas. Constraints **14**(1), 80–116 (2009)
119. Rao, R.B., Gordon, D., Spears, W.: For every generalization action, is there really an equal and opposite reaction? Analysis of the conservation law for generalization performance. In: Proceedings of the 12th International Conference on Machine Learning, pp. 471–479. Morgan Kaufmann (1995)
120. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)
121. Rice, J.R., Ramakrishnan, N.: How to get a free lunch (at no cost). Techical report 99–014, Purdue University, April 1999

122. Roberts, M., Howe, A.E.: Directing a portfolio with learning. In: AAAI 2006 Workshop on Learning for Search (2006)
123. Roberts, M., Howe, A.E.: Learned models of performance for many planners. In: ICAPS 2007 Workshop AI Planning and Learning (2007)
124. Roberts, M., Howe, A.E., Wilson, B., des Jardins, M.: What makes planners predictable? In: ICAPS, pp. 288–295 (2008)
125. Sakkout, H., Wallace, M.G., Richards, E.B.: An instance of adaptive constraint propagation. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 164–178. Springer, Heidelberg (1996). doi:10.1007/3-540-61551-2_73
126. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 255–260. AAAI Press (2007)
127. Sayag, T., Fine, S., Mansour, Y.: Combining multiple heuristics. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 242–253. Springer, Heidelberg (2006). doi:10.1007/11672142_19
128. Schapire, R.E.: The strength of weak learnability. Mach. Learn. **5**(2), 197–227 (1990)
129. Sillito, J.: Improvements to and estimating the cost of solving constraint satisfaction problems. Master's thesis, University of Alberta (2000)
130. Silverthorn, B., Miikkulainen, R.: Latent class models for algorithm portfolio methods. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence (2010)
131. Smith, T.E., Setliff, D.E.: Knowledge-based constraint-driven software synthesis. In: Knowledge-Based Software Engineering Conference, pp. 18–27, September 1992
132. Smith-Miles, K., Lopes, L.: Measuring instance difficulty for combinatorial optimization problems. Comput. Oper. Res. **39**(5), 875–889 (2012)
133. Smith-Miles, K.A.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**, 6: 1–6: 25 (2008)
134. Smith-Miles, K.A.: Towards insightful algorithm selection for optimisation using meta-learning concepts. In: IEEE International Joint Conference on Neural Networks, pp. 4118–4124, June 2008
135. Soares, C., Brazdil, P.B., Kuba, P.: A meta-learning method to select the kernel width in support vector regression. Mach. Learn. **54**(3), 195–209 (2004)
136. Stergiou, K.: Heuristics for dynamically adapting propagation in constraint satisfaction problems. AI Commun. **22**(3), 125–141 (2009)
137. Stern, D.H., Samulowitz, H., Herbrich, R., Graepel, T., Pulina, L., Tacchella, A.: Collaborative expert portfolio management. In: AAAI, pp. 179–184 (2010)
138. Streeter, M.J., Golovin, D., Smith, S.F.: Combining multiple heuristics online. In: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 1197–1203. AAAI Press (2007)
139. Streeter, M.J., Golovin, D., Smith, S.F.: Restart schedules for ensembles of problem instances. In: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 1204–1210. AAAI Press (2007)
140. Streeter, M.J., Smith, S.F.: New techniques for algorithm portfolio design. In: UAI, pp. 519–527 (2008)
141. Terashima-Marín, H., Ross, P., Valenzuela-Rendón, M.: Evolution of constraint satisfaction strategies in examination timetabling. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 635–642. Morgan Kaufmann (1999)

142. Tolpin, D., Shimony, S.E.: Rational deployment of CSP heuristics. In: IJCAI, pp. 680–686 (2011)
143. Tsang, E.P.K., Borrett, J.E., Kwan, A.C.M.: An attempt to map the performance of a range of algorithm and heuristic combinations. In: Proceedings of AISB 1995, pp. 203–216. IOS Press (1995)
144. Utgoff, P.E.: Perceptron trees: a case study in hybrid concept representations. In: National Conference on Artificial Intelligence, pp. 601–606 (1988)
145. Vassilevska, V., Williams, R., Woo, S.L.M.: Confronting hardness using a hybrid approach. In: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms SODA 2006, pp. 1–10. ACM, New York (2006)
146. Vrakas, D., Tsoumakas, G., Bassiliades, N., Vlahavas, I.: Learning rules for adaptive planning. In: Proceedings of the 13th International Conference on Automated Planning and Scheduling, pp. 82–91 (2003)
147. Wang, J., Tropper, C.: Optimizing time warp simulation with reinforcement learning techniques. In: Proceedings of the 39th Conference on Winter simulation WSC 2007, pp. 577–584. IEEE Press, Piscataway (2007)
148. Watson, J.: Empirical modeling and analysis of local search algorithms for the job-shop scheduling problem. Ph.D. thesis, Colorado State University, Fort Collins, CO, USA (2003)
149. Weerawarana, S., Houstis, E.N., Rice, J.R., Joshi, A., Houstis, C.E.: PYTHIA: a knowledge-based system to select scientific algorithms. ACM Trans. Math. Softw. **22**(4), 447–468 (1996)
150. Wei, W., Li, C.M., Zhang, H.: Switching among non-weighting, clause weighting, and variable weighting in local search for SAT. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 313–326. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85958-1_21
151. Wilson, D., Leake, D., Bramley, R.: Case-based recommender components for scientific problem-solving environments. In: Proceedings of the 16th International Association for Mathematics and Computers in Simulation World Congress (2000)
152. Wolpert, D.H.: Stacked generalization. Neural Netw. **5**, 241–259 (1992)
153. Wolpert, D.H.: The supervised learning no-free-lunch theorems. In: Proceedings of the 6th Online World Conference on Soft Computing in Industrial Applications, pp. 25–42 (2001)
154. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Trans. Evol. Comput. **1**(1), 67–82 (1997)
155. Wu, H., van Beek, P.: On portfolios for backtracking search in the presence of deadlines. In: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, pp. 231–238. IEEE Computer Society, Washington, DC (2007)
156. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hierarchical hardness models for SAT. In: CP, pp. 696–711 (2007)
157. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: 24th Conference of the Association for the Advancement of Artificial Intelligence (AAAI 2010), pp. 210–216 (2010)
158. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla-07: the design and analysis of an algorithm portfolio for SAT. In: CP, pp. 712–727 (2007)
159. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) **32**, 565–606 (2008)
160. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla2009: an automatic algorithm portfolio for SAT. In: 2009 SAT Competition (2009)

161. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: 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) (2011)
162. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 228–241. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31612-8_18
163. Yu, H., Rauchwerger, L.: An adaptive algorithm selection framework for reduction parallelization. IEEE Trans. Parallel Distrib. Syst. **17**(10), 1084–1096 (2006)
164. Yu, H., Zhang, D., Rauchwerger, L.: An adaptive algorithm selection framework. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 278–289. IEEE Computer Society, Washington, DC (2004)
165. Yun, X., Epstein, S.L.: Learning algorithm portfolios for parallel execution. In: Hamadi, Y., Schoenauer, M. (eds.) Proceedings of the 6th International Conference Learning and Intelligent Optimisation LION. LNCS, vol. 7219, pp. 323–338. Springer, Heidelberg (2012)