Tradeoffs in the Empirical Evaluation of Competing Algorithm Designs

UBC CS Technical Report TR-2009-21

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T1Z4, Canada
{hutter,hoos,kevinlb}@cs.ubc.ca

Abstract. We propose an empirical analysis approach for characterizing tradeoffs between different methods for comparing a set of competing algorithm designs. Our approach can provide insight into performance variation both across candidate algorithms and across instances. It can also identify the best tradeoff between evaluating a larger number of candidate algorithm designs, performing these evaluations on a larger number of problem instances, and allocating more time to each algorithm run. We applied our approach to a study of the rich algorithm design spaces offered by three highly-parameterized, state-of-the-art algorithms for satisfiability and mixed integer programming, considering six different distributions of problem instances. We demonstrate that the resulting algorithm design scenarios differ in many ways, with important consequences for both automatic and manual algorithm design. We expect that both our methods and our findings will lead to tangible improvements in algorithm design methods.

Keywords: Algorithm design · Empirical analysis · Algorithm configuration · Parameter optimization

1 Introduction

There are two main paradigms for the development of heuristic algorithms for solving hard computational problems. The traditional approach is a manual process in which a designer iteratively adds algorithm components or modifies existing mechanisms. More recently, an increasing body of research has proposed the automation of all or part of this process (see, e.g., Gratch and Dejong, 1992; Minton, 1996; Birattari et al., 2002; Birattari, 2005; Adenso-Diaz and Laguna, 2006; Audet and Orban, 2006; Bartz-Beielstein, 2006; Hutter et al., 2007b; Balaprakash et al., 2007; Hoos, 2008; Hutter et al., 2009; KhudaBukhsh et al., 2009, noting that algorithm configuration and parameter optimization can be understood as specific instances of algorithm design). Regardless of whether a manual, a semi-automatic, or an automatic design paradigm is adopted, the designer faces a common problem: deciding how to compare different algorithm designs. In many domains of interest, existing theoretical techniques are not powerful enough to

answer this question. Instead, this comparison is typically based on empirical observations of algorithm performance.

There is a straightforward method for empirically comparing algorithm designs: evaluating each candidate design on every problem instance of interest. Unfortunately, this conceptually simple solution is rarely practical, because the overall algorithm development process is always at least somewhat time constrained. Therefore, it is typically impossible to consider every design from a large space of possible candidates, or to evaluate those designs that *are* considered on every problem instance of interest. Instead, in cases where the evaluation of each candidate algorithm design requires many computationally-expensive algorithm runs, the number of (design, instance) pairs that can be tested is often severely limited. Thus, it is necessary to make careful choices about (1) the number of candidate designs to consider and (2) the number of problem instances to use in evaluating them. Furthermore, it is ideally desirable to avoid prematurely terminating, or "censoring", any run. (This issue arises particularly in the context of designing algorithms with the goal of minimizing the runtime required for solving a given problem instance or reaching a certain solution quality.) Again, however, time constraints can make this impractical, requiring us to choose (3) some *captime* at which runs will be terminated whether or not they have completed. In this paper, we empirically study the tradeoff between these three choices.

Automated methods for performing parameter optimization and algorithm configuration can be understood as sophisticated heuristics for deciding which configurations to consider and for choosing instances on which to evaluate them.[1] Racing algorithms (Maron and Moore, 1994; Birattari et al., 2002; Birattari, 2005; Balaprakash et al., 2007) emphasize using as few problem instances as possible to reliably choose among a fixed set of candidate algorithm designs. More specifically, they incrementally expand the instance set (i.e., perform more runs for all designs) and at each step eliminate designs that are statistically significantly worse than others in terms of observed performance. In contrast, research on sequential search algorithms focusses on the question of *which* algorithm designs to evaluate. Many such procedures, including Multi-TAC (Minton, 1996), Calibra (Adenso-Diaz and Laguna, 2006), the mesh adaptive direct search algorithm (Audet and Orban, 2006) and BasicILS (Hutter et al., 2007b), use a fixed, user-defined instance set. Other search algorithms include mechanisms for adapting the set of instances used for evaluating algorithm designs; examples are Composer (Gratch and Dejong, 1992), SPO (Bartz-Beielstein, 2006) and FocusedILS (Hutter et al., 2007b).

The literature on automatic algorithm configuration places less emphasis on the choice of captime. (The only exception of which we are aware is our own recent extension of the ParamILS framework (Hutter et al., 2009), which dynamically adapts the per-run cutoff time.) However, the issue has been studied in the context of evaluating heuristic algorithms. Segre et al. (1991) demonstrated that small captimes can lead to misleading conclusions when evaluating explanation-based learning algorithms. Etzioni and Etzioni (1994) extended statistical tests to deal with partially-censored runs in an effort to limit the large impact of captimes observed by Segre et al. (1991). Simon

[1] Of course, essentially the same point can be made about manual approaches, except that they are typically less sophisticated and have been discussed less rigorously in the literature.

and Chatalic (2001) demonstrated the relative robustness of comparisons between SAT solvers for three different captimes.

We focus on the algorithm design objective of minimizing runtime for solving a given problem. This objective is important for solving a wide variety of computationally-hard problems in operations research, constraint programming, and artificial intelligence, where the runtime of poor and strong algorithms on the same problem instance often differs by several orders of magnitude. Minimizing runtime is furthermore a very interesting problem, because it implies a strong correlation between the quality of an algorithm design and the amount of time required to evaluate its performance. This property opens up the possibility of terminating many long runs after a comparably small captime, while still obtaining full information on short runs. (Note that by definition, short runs occur more frequently when evaluating good algorithm designs.) Here, we present the first detailed empirical study of the role of this captime in algorithm design. We show that the impact of captime is similar to that of the size of the instance set based upon which design decisions are made. Large captimes lead to unreasonable time requirements for evaluating single candidate algorithm designs (especially poor ones). This typically limits the number of problem instances on which candidate designs are evaluated, which in turn can lead to misleading performance results. On the other hand, evaluations based on overly aggressive captimes favour algorithms with good initial performance, and thus the algorithms chosen on the basis of these evaluations may not perform competitively when allowed longer runs.

We note that it is not our main goal in this work to propose yet another method for making choices about which algorithm designs to explore, which benchmark set to use, or how much time to allocate to each algorithm run. Rather, we propose an empirical analysis approach for investigating the tradeoffs between these choices that must be addressed by *any* (manual or automated) approach to algorithm design. More specifically, we study the rich algorithm design spaces offered by three highly-parameterized, state-of-the-art algorithm frameworks: a tree search and a local search framework for solving propositional satisfiability (SAT) problems, and the commercial solver CPLEX for mixed integer programming (MIP) problems. We study industrial SAT instances from software verification (Babić and Hu, 2007) and bounded model checking (Zarpas, 2005), as well as structured SAT instances from quasi-group completion (Gomes and Selman, 1997) and graph colouring (Gent et al., 1999). Our MIP instances were drawn from winner determination in combinatorial auctions (Leyton-Brown et al., 2000) and a range of real-world problems including capacitated warehouse location and airplane landing scheduling (Beasley, 1990).

Based on the data that we analyze in this paper, we can answer a number of questions about a given design scenario that are important for both manual and automated algorithm design.[2] Here, we focus on the following eight questions:

1. How much does performance vary across candidate algorithm designs?
2. How large is the variability in hardness across benchmark instances?

[2] We note that in most of this work, we do not answer these questions inexpensively; our methods are applied only *post hoc*, not online. However, in Section 6, we describe an approach for performing a computationally-cheap, approximate analysis online.

3. Which benchmark instances are useful for discriminating between candidate designs?
4. Are the same instances "easy" and "hard" for all candidate designs?
5. Given a fixed computational budget and a fixed captime, how should we trade off the number of designs evaluated *vs* the number of instances used in these evaluations?
6. Given a fixed computational budget and a fixed number of instances, how should we trade off the number of designs evaluated *vs* the captime used for each evaluation?
7. Given a budget for identifying the best of a fixed set of candidate designs, how many instances, N, and which captime, κ, should be used for evaluating each algorithm design?
8. Likewise, how should we trade off N and κ if the goal is to *rank* a fixed set of algorithm designs?

Our experimental analysis approach allows us to answer each of these questions for each of our algorithm design scenarios. Throughout, we discuss these answers in detail for two rather different scenarios, summarizing our findings for the others. Overall, our experimental analysis yields the broad conclusion that the algorithm design scenarios we studied are extremely heterogeneous. Specifically, the answers to each of our questions above differ widely across scenarios, suggesting that no single choice of number of designs, number of instances, and set of captimes is likely to yield good performance across the board. Instead, we suggest that current state-of-the-art algorithm design methods may be substantially improved by adapting to the design scenario at hand, driven by observed data.

The remainder of this paper is structured as follows. We start in Section 2 by covering experimental preliminaries and describing in detail the algorithm design scenarios we use throughout. Next, in Section 3, we investigate distributions of instance hardness, quality of candidate algorithms and the interaction between the two, thereby addressing Questions 1–4 above. In Section 4 we consider Questions 5–7, presenting an empirical study of the tradeoffs between instance set size, N, and captime, κ, for the problem of identifying the best of several candidate algorithms. In Section 5, we investigate the same tradeoffs in the context of the problem of *ranking* several candidate algorithms in order to answer Question 8. In Section 6, we study computationally-cheap, approximate answers to Questions 1–8. These are obtained online, based on a predictive model of runtime. Finally, in Section 7, we discuss the high-level implications of our findings and identify some promising avenues for future work.

2 Experimental Preliminaries

In this paper, we study the SATENSTEIN, SPEAR and CPLEX solver frameworks, described in detail in Table 1. SATENSTEIN is a very recent, highly parameterized framework for SAT solvers based on stochastic local search (SLS) (KhudaBukhsh et al., 2009). It is able to instantiate nearly all state-of-the-art SLS solvers for SAT. SATENSTEIN is designed to be used in conjunction with a configuration procedure to automatically construct new SLS solvers for problem domains of interest. It has 41 parameters

Algorithm	Parameter type	# params of type	# values considered	Total # configs, $\lvert\Theta\rvert$
CPLEX	Categorical	50	2–7	$1.38 \cdot 10^{37}$
	Integer	8	5–7	
	Continuous	5	3–5	
SPEAR	Categorical	10	2–20	$8.34 \cdot 10^{17}$
	Integer	4	5–8	
	Continuous	12	3–6	
SATENSTEIN	Categorical	16	2–13	$4.82 \cdot 10^{12}$
	Integer	5	4–9	
	Continuous	20	3–10	

Table 1. Parameter overview for the algorithms we consider. High-level information on the algorithms' parameters is given in the text; a detailed list of all parameters and the values we considered can be found in an online appendix at `http://www.cs.ubc.ca/labs/beta/Projects/AlgoDesign`.

and includes components and mechanisms from algorithms based on WalkSAT, dynamic local search and G2WSAT variants (see KhudaBukhsh et al., 2009, for further references). In total, this gives rise to 4.82×10^{12} different possible algorithm designs.

SPEAR is a recent state-of-the-art tree search SAT solver targeting industrial instances (Hutter et al., 2007a). Many of its categorical parameters control heuristics for variable and value selection, clause sorting and resolution ordering, while others enable or disable optimizations. The continuous and integer parameters mainly deal with activity, decay and elimination of variables and clauses, as well as randomized restarts. In total, there are 26 parameters, giving rise to 8.34×10^{17} possible algorithm designs with widely varying characteristics.

Finally, CPLEX (version 10.1.1) is a commercial optimization tool for solving mixed integer programming (MIP) problems (ILOG Inc., 2008). It is a massively-parameterized branch-and-cut algorithm, with categorical parameters governing variable and branching heuristics, types of cuts to be used, probing, dive type and subalgorithms, as well as amount and type of preprocessing to be performed. Out of 159 user-definable parameters, we identified 63 that affect the search trajectory. We were careful to exclude all parameters that change the problem formulation (e.g., by specifying the numerical accuracy required of a solution). The total number of possible algorithm designs is 1.38×10^{38}.

We constructed six algorithm design scenarios by combining those three algorithms with two benchmark distributions each. We selected these scenarios to span a wide range of algorithms, problem classes and instance hardness. For the study of SATENSTEIN, we employed two sets of SAT-encoded "crafted" instances for which we believe it to be the best-performing algorithm (KhudaBukhsh et al., 2009): satisfiable quasi-group completion problems (`QCP` (Gomes and Selman, 1997)) and satisfiable graph colouring problems based on small world graphs (`SWGCP` (Gent et al., 1999)). For the study of SPEAR, we employed two sets of SAT-encoded industrial verification problem instances, for which it has been demonstrated to be the best available algorithm (Hutter et al., 2007a): IBM bounded model checking instances (Zarpas, 2005) and software verification instances generated by the CALYSTO static checker (Babić and Hu, 2007). Finally, for the study of CPLEX, we employed two very different sets

of MIP-encoded problems, for both of which, to our best knowledge, CPLEX is the best available algorithm: a homogeneous set of combinatorial winner determination instances (Leyton-Brown et al., 2000), `Regions100`, and a very heterogenous mix of 140 instances from ORLIB, including set covering and capacitated p-median problems, as well as problems from capacitated warehouse location and airplane landing scheduling (see Beasley, 1990, noting that we obtained the instances from `http://www.andrew.cmu.edu/user/anureets/mpsInstances.htm`).

In order to complete the specification of these algorithm design scenarios, we need to define an objective function to be optimized. In this paper, we use the objective of minimizing the mean runtime across a set of instances. However, when runs are prematurely terminated, we only know a lower bound on mean runtime. In order to penalize timeouts, we define the *penalized average runtime (PAR)* of a set of runs with cutoff time κ as the mean runtime over those runs, where unsuccessful runs are counted as $a \cdot \kappa$ with penalization constant $a \geq 1$. There is clearly more than one way of sensibly aggregating runtimes in the presence of capping. One reason we chose PAR is that it generalizes two other natural schemes;

1. "lexicographic": the number of instances solved, breaking ties by total runtime (used in the 2009 SAT competition[3]); and
2. total (or average) runtime across instances, treating timeouts as completed runs.

Scheme 1 is PAR with $a = \infty$, while Scheme 2 is PAR with $a = 1$. Here, we use $a = 10$ to emphasize the importance of timeouts more than in the second scheme, but to yield a more robust measure than the first scheme. KhudaBukhsh et al. (2009) found algorithm rankings to be robust with respect to different choices of a. Here, we also experimented with $a = 1$ and $a = 100$ instead of $a = 10$, and obtained qualitatively very similar results. All methods we study in this paper are well defined and meaningful under other design objectives, such as median runtime and more complicated measures, such as pre-2009 SAT competition scoring functions.

When estimating the performance of a candidate design based on N runs, we only perform a single run for each of N problem instances. This is justified, because for the minimization of mean runtime (and indeed other common optimization objectives), performing a single run on each of N different instances yields an estimator with minimal variance for a given sample size N (see, e.g., Birattari, 2005).

The empirical analysis approach that we propose in this paper takes as its input data a $M \times P$ matrix describing the performance of M candidate algorithm designs on a set of P problem instances. In order to analyze the enormous algorithm design spaces of SATENSTEIN, SPEAR and CPLEX in an unbiased way, we obtained candidate algorithm designs by sampling feasible parameter configurations uniformly at random. Thus, in this paper we chose the rows of our matrix to be the algorithm default plus $M - 1 = 999$ random designs. However, our methods are not limited to random designs. Candidate designs may also be obtained from trajectories of automated algorithm design approaches or manually; indeed, in Section 5 we use ten solvers from a recent SAT competition as candidates.

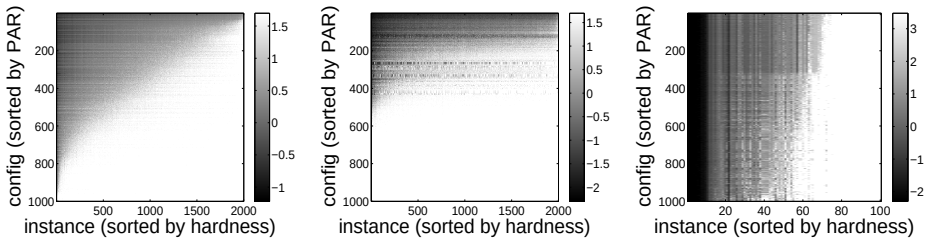
[3] `http://www.satcompetition.org/2009/spec2009.html`

Design Scenario	Captime [s]	Default	Best known	Best sampled
SATENSTEIN-SWGCP	5	21.02	0.035 (from KhudaBukhsh et al., 2009)	0.043
SATENSTEIN-QCP	5	10.19	0.17 (from KhudaBukhsh et al., 2009)	0.21
SPEAR-IBM	300	1393	795 (from Hutter et al., 2007a)	823
SPEAR-SWV	300	466	1.37 (from Hutter et al., 2007a)	1.90
CPLEX-REGIONS100	5	1.76	0.32 (from Hutter et al., 2009)	0.86
CPLEX-ORLIB	300	74.4	74.42 (CPLEX default)	54.1

Table 2. Comparison of penalized average runtime (PAR) for our algorithm design scenarios. We show PAR of the default, of the best known domain-specific algorithm design (including its source), and of the best of our 999 randomly-sampled algorithm designs.

One may wonder how well randomly-sampled designs can actually perform for hard problems like those we study here. Indeed, in many other optimization problems, sampling candidate solutions at random could yield very poor solutions. As it turns out, this is not the case in the algorithm design scenarios we study. In Table 2, for each scenario we compare the performance of the algorithm default, the best-known candidate design for the scenario, and the best out of 999 randomly-sampled designs. In all scenarios, the best randomly-sampled algorithm design performed better than the default, typically by a substantial margin. For the SATENSTEIN and the SPEAR scenarios, this performance was close to that of the best-known candidate design, while for CPLEX-REGIONS100 the difference between the two was somewhat larger. (For ORLIB, we are not aware of any published parameter setting of CPLEX and thus only compare to the CPLEX default.)

We generated the data for these runtime matrices as follows. For each design scenario, we ran $M = 1000$ designs on the available set of instances, with a captime κ_{max}. We chose κ_{max} such that either the best design could easily solve all instances within that time, or, for design scenarios with harder problem instances, at a maximum of 300 seconds per run. Specifically, for the comparably easy instances in design scenarios SATENSTEIN-QCP, SATENSTEIN-SWGCP and CPLEX-REGIONS100, we evaluated each candidate design on $P = 2000$ instances, terminating unsuccessful runs after a captime of $\kappa_{max} = 5s$. Scenarios SPEAR-IBM, SPEAR-SWV and CPLEX-ORLIB contain much harder instances, and we thus used a captime of $\kappa_{max} = 300s$ seconds; the benchmark instance sets contained $P = 140$, 100, and 100 instances, respectively. We carried out all experiments using a cluster of 55 dual 3.2GHz Intel Xeon CPUs.[4]



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 1. Raw data: matrix of runtime of each of the $M = 1000$ sampled candidate algorithm designs on each of the P instances. Each dot in the matrix represents the runtime of a candidate design on a single instance. Darker dots represent shorter runtimes; the grayscale is logarithmic with base 10. Designs are sorted by their PAR score across all P instances. Instances are sorted by hardness (mean runtime of the M candidate designs, analogous to PAR counting runs that timed out at captime κ as $10 \cdot \kappa$).

3 Analysis of Runtime Variability across Designs and Instances

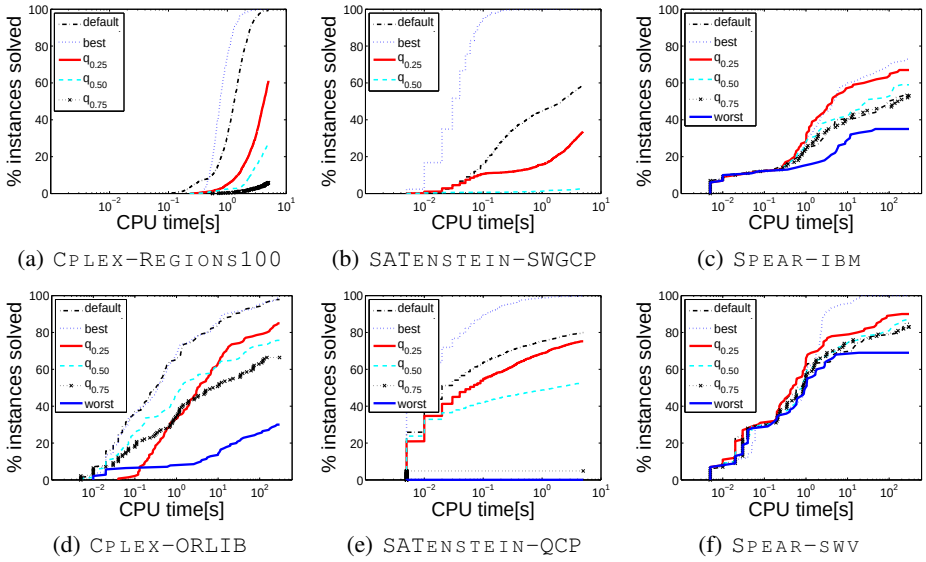
In this section, we provide an overview of the interaction between our three choices: which candidate algorithm designs are evaluated, which instances are used, and how much time is allocated to each run.

Figures 1 and 2 together give an overall description of this space. In Figure 1, we plot the raw data: the runtime for all combinations of instances and candidate algorithm designs. In Figure 2, we give more detailed information about the precise runtime values for six candidate designs (the default, the best, the worst, and three quantiles), plotting a cumulative distribution of the percentage of benchmark instances solved by $\boldsymbol{\theta}$ as a function of time. Based on these plots, we can make four key observations about our algorithm design scenarios, providing answers to Questions 1–4 posed in the introduction.

Q1: How much does performance vary across candidate designs?

The variability of quality across candidate algorithm designs differed substantially be-

[4] These cluster nodes had 2GB RAM each and 2MB cache per CPU, and ran OpenSuSE Linux 10.1. We measured runtimes as CPU time on these reference machines. Gathering the data for the input matrix took around 1.5 CPU months for each of the three scenarios with $\kappa_{max} = 5s$, 1 CPU month for SPEAR-SWV, and 2.5 CPU months for each of SPEAR-IBM and CPLEX-ORLIB.



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 2. Hardness variation across all P instances in different algorithm design scenarios. For each scenario, the plot shows the percentage of benchmark instances solved by a number of candidate algorithm designs (default, best and worst sampled design, and designs at the $q_{0.25}$, $q_{0.50}$, and $q_{0.75}$ quantiles of quality across the sampled designs) as a function of allowed time. In cases where a design did not solve *any* of the P instances, we do not show it in the figure.

tween scenarios. For example, in CPLEX-REGIONS100, the worst candidate design did not solve a single problem instance, while the best one solved all instances in less than five seconds (see Figures 1(a) and 2(a)). In contrast, the difference between worst and best candidate design was much smaller for scenario SPEAR-IBM (see Figures 1(c) and 2(c)). The SATENSTEIN scenarios showed even larger variation across candidate designs than CPLEX-REGIONS100, whereas CPLEX-ORLIB and SPEAR-SWV showed less variation (comparable to SPEAR-IBM). We expect scenarios with large performance variation across candidate designs to require automated design procedures that emphasize an effective search through the design space. In other scenarios, an effective mechanism for selecting the best number of instances and cutoff time to use can be more important.

Q2: How large is the variability in hardness across benchmark instances?

The variability of hardness across benchmark instances also differed substantially. For example, in scenario CPLEX-REGIONS100, there was "only" about an order of magnitude difference between the runtime of a candidate algorithm on the easiest and the hardest instances (see Figure 2(a)). In contrast, this difference was at least five orders of magnitude for scenario SPEAR-IBM (see Figure 2(c)). Scenario SATENSTEIN-SWGCP was similar to scenario CPLEX-REGIONS100 in having small variability of instance hardness, while the other scenarios were more similar to scenario SPEAR-IBM in this respect. In some scenarios (e.g., SATENSTEIN-SWGCP; see Figure 2(b)), the difference

in hardness between the easiest and the hardest instances depended on the candidate design, with good algorithms showing more robust performance across all instances. Variability in instance hardness can substantially affect the performance of algorithm design procedures. When hardness varies substantially across instances, these observations support the strategy of performing many runs on easy instances and only using harder ones sparingly in order to assess scaling behaviour. On the other hand, if the objective is, for example, the minimization of mean runtime across all instances in the set, then special care needs to be taken to ensure good performance on the hardest instances, which often dominate the mean.

Q3: Which benchmark instances are useful for discriminating between candidate designs?

In some—but not all—scenarios only a subset of instances were useful for discriminating between the candidate designs. While essentially all instances were useful in this sense for scenario CPLEX-REGIONS100, this was not true for scenario SPEAR-IBM. In that scenario, within the cutoff time ($\kappa_{max} = 300s$), over 35% of the instances were infeasible for *all* considered candidate designs (see Figure 1(c)). Similarly, about 10% of the instances in that scenario were trivially solvable for all candidate designs, resulting in runtimes smaller than the resolution of the CPU timer. Next to SPEAR-IBM, only SATENSTEIN-QCP and SPEAR-SWV had substantial percentages of such trivial instances. While such uniformly easy instances do not pose a problem in principle (since they can always be solved quickly) they can pose a problem for automated algorithm configuration procedures that often evaluate candidate designs based on a few instances. For example, the performance of SPO (Bartz-Beielstein, 2006) and FocusedILS (Hutter et al., 2007b) could be expected to degrade if many trivial instances were added. On the other hand, uniformly *infeasible* instances pose a serious problem, both for manual and automated algorithm design. Every algorithm run on such an instance costs valuable time without yielding any information.

Q4: Are the same instances "easy" and "hard" for all candidate designs?

In some scenarios, the per-design runtime rankings were fairly stable across candidate designs. In other words, the runtime of a candidate design $\boldsymbol{\theta}$ on an instance π could be well-modeled as depending on the overall quality of $\boldsymbol{\theta}$ (averaged across instances) and the overall hardness of π (averaged across candidate algorithms). This was approximately the case for scenario CPLEX-REGIONS100 (see Figure 1(a)), where better-performing candidate algorithms solved the same instances as weaker algorithms, plus some additional ones. The ranking was also comparably stable in scenario SPEAR-IBM and SATENSTEIN-QCP. In contrast, in some scenarios we observed instability in the ranking of candidate designs from one instance to another: whether or not a design $\boldsymbol{\theta}$ performed better on an instance π than another design $\boldsymbol{\theta}'$ depended more strongly on the instance π. One way this is evidenced in the matrix is as a "checkerboard pattern"; see, e.g., rows 400–900 and the two instance sets in columns 76–83 and 84–87 in Figure 1(f). In these cases, designs that did well on the first set of instances tended to do poorly on the second set and vice versa. Overall, the most pronounced examples of this behaviour were observed in scenarios CPLEX-ORLIB and SPEAR-SWV (see Figures 1(d) and 1(f), and the crossings of cumulative distributions in Figures 2(d) and 2(f).) Scenarios in which algorithm rankings are unstable across instances are problematic to

address with both manual and automated methods for offline, algorithm configuration and parameter tuning, because different instances often require very different mechanisms to be solved effectively. Approaches likely to hold promise in such cases are (1) splitting such heterogeneous sets of instances into more homogeneous subsets, (2) applying portfolio-based techniques (Gomes and Selman, 2001; Horvitz et al., 2001; Xu et al., 2008), or (3) applying per-instance algorithm configuration (Hutter et al., 2006). However, note that in some cases (e.g., scenario SPEAR-SWV) the instability between relative rankings is local to poor algorithm designs. In such cases it is possible to find a single good candidate design that performs very well on all instances, limiting the potential improvement by more complicated per-instance approaches.

4 Tradeoffs in Identifying the Best Candidate Algorithm

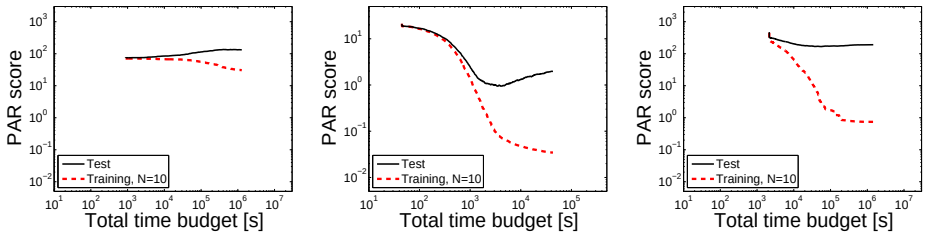
In this section, we consider the problem of identifying the best algorithm design—that is, the design yielding the lowest penalized average runtime (PAR).

4.1 Overconfidence and Overtuning: Basics

One might imagine that without resource constraints but given a fixed set of instances from some distribution of interest, it would be easy to identify the best candidate algorithm. Specifically, we could just evaluate every algorithm design on every instance, and then pick the best. This is indeed a good way of identifying the design with the best performance on the exact instances used for evaluation. However, when the set of instances is too small, the observed performance of the design selected may not be reflective of—and, indeed, may be overly optimistic about—performance on *other* instances from the same underlying distribution. We call this phenomenon *overconfidence*. This effect is notorious in machine learning, where it is well known that models optimized to perform well on small datasets often generalize poorly (Hastie et al., 2001). Furthermore, generalization performance can actually *degrade* when the number of algorithm designs considered (in machine learning the *hypothesis space*) grows too large. This effect is called *overfitting* in machine learning (Hastie et al., 2001) and *overtuning* in optimization (Birattari et al., 2002; Birattari, 2005; Hutter et al., 2007b). In this section we will examine the extent to which overconfidence and overtuning can arise in our scenarios. Before doing so, we need to explain how we evaluate the generalization performance of a candidate design.

Up to this point, we have based our analysis on the full input matrix; from now on, we partition instances into a *training set* Π and a *test set* Π'. The training set is used for choosing the best candidate design $\hat{\boldsymbol{\theta}}$, while the test set is used only for assessing $\hat{\boldsymbol{\theta}}$'s performance. A separate test set is required because the performance of $\hat{\boldsymbol{\theta}}$ on Π is a *biased* estimator of its performance on Π' (see, e.g., Birattari, 2005); indeed, this bias increases as $|\Pi|$ decreases. When algorithm designs are selected based on a relatively small benchmark set, they can be adapted specifically to the instances in that set and fail to achieve good performance on other instances.

We compute training and test performance for a given set of candidate designs $\boldsymbol{\Theta}$ based on training and test sets Π and Π' and captime κ as follows. Let $\boldsymbol{\Theta}_{input}$ denote



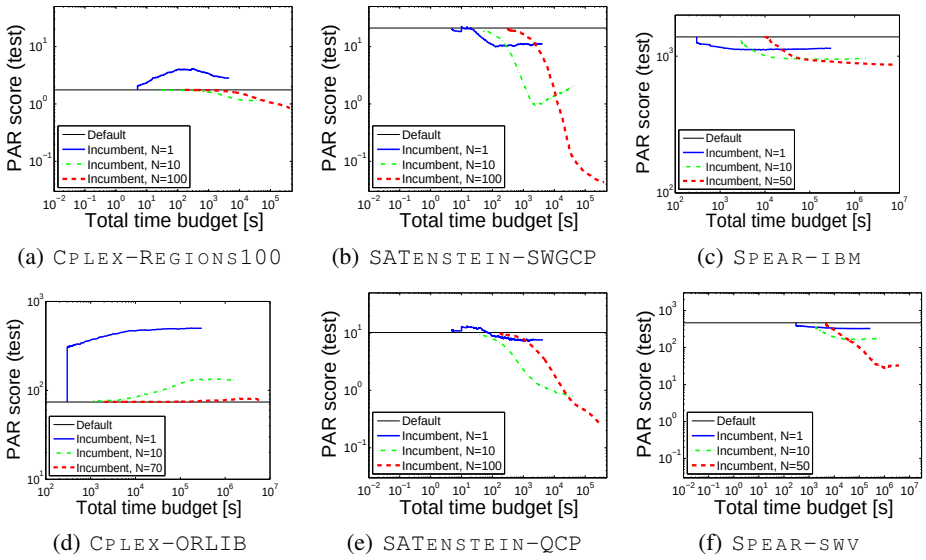
(a) CPLEX-ORLIB (b) SATENSTEIN-SWGCP (c) SPEAR-SWV

Fig. 3. Overconfidence and overtuning for training sets of size $N = 10$. We plot training and test performance (penalized average runtime, PAR, for $N = 10$ training instances and $P/2$ test instances) of *IS-Best*$(N = 10, \kappa, t)$, where κ is the "full" captime of the design scenario ($\kappa = 300s$ for CPLEX-ORLIB and SPEAR-SWV, and $\kappa = 5s$ for SATENSTEIN-SWGCP).

the set of $M = 1000$ candidate designs and let Π_{input} denote the set of P instances on which the input matrix is based. We evaluate candidate designs $\boldsymbol{\theta} \in \boldsymbol{\Theta}_{input}$ on subsets of instances $\Pi \subseteq \Pi_{input}$ with captime κ by using their known runtimes from the input matrix. We count runtimes $\geq \kappa$ as $10 \cdot \kappa$, according to the PAR criterion. Given a set of candidate designs $\boldsymbol{\Theta} \subseteq \boldsymbol{\Theta}_{input}$, an instance set $\Pi \subseteq \Pi_{input}$ and a captime κ, the training score is the PAR score of the $\hat{\boldsymbol{\theta}} \in \boldsymbol{\Theta}$ with minimal PAR on Π using captime κ. The test score is the PAR score of the same $\hat{\boldsymbol{\theta}}$ on Π' based on the "full" captime, κ_{max}, used to construct the input matrix.

We use an iterative sampling approach to estimate the *expected training and test performance* given a computational budget t, N training instances and captime κ. In each iteration, we draw a training set of N disjoint instances $\Pi \subseteq \Pi_{input}$ and start with an empty set of designs $\boldsymbol{\Theta}$. We then expand $\boldsymbol{\Theta}$ by randomly adding elements of $\boldsymbol{\Theta}_{input} \setminus \boldsymbol{\Theta}$, until either $\boldsymbol{\Theta} = \boldsymbol{\Theta}_{input}$ or the time for evaluating all $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ on all instances $\pi \in \Pi$ with captime κ exceeds t. Finally, we evaluate training and test performance of $\boldsymbol{\Theta}$ based on Π and κ. In what follows, we always work with expected training and test performance, sometimes dropping the term "expected" for brevity. We calculate these quantities based on $K = 1000$ iterations, each of them using independently-sampled, disjoint training and test sets. We always use test sets of cardinality $|\Pi'| = P/2$, but vary the size of the training set Π from 1 to $P/2$. We refer to the resulting expected performance using a training set of N instances, captime κ, and as many designs as can be evaluated in time t as *IS-Best*(N, κ, t), short for *IterativeSampling-Best*.

Using this iterative sampling approach, we investigated the difference between training and test performance. Figure 3 shows the three that gave rise to the most pronounced training/test performance gaps. Based on training sets of size $N = 10$, we saw clear evidence for overconfidence (divergence between training and test performance) in all three scenarios, and evidence for overtuning (test performance that degrades as we increase the number of designs considered) for CPLEX-ORLIB and SATENSTEIN-SWGCP. We believe that CPLEX-ORLIB, SATENSTEIN-SWGCP, and SPEAR-SWV exhibited the most pronounced training/test performance gaps because of their likewise-unstable relative algorithm rankings across the respective instance sets (which we observed in Fig-



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 4. Test performance (PAR for $P/2$ test instances) of *IS-Best*(N, κ, t), where κ is the "full" captime of the design scenario ($\kappa = 300s$ for CPLEX-ORLIB and the SPEAR scenarios, and $\kappa = 5s$ for the rest). We plot graphs for $N = 1$, $N = 10$, and $N = \min(100, P/2)$. For reference, we plot test performance of the default. The best N for a given time budget x can be observed as the line with minimal PAR score at x.

ures 1(b), 1(d), and 1(f)). Intuitively, in such scenarios it is not surprising that the best algorithm design for a small subset of instances can be very different from the best design for a much larger set.

4.2 Trading off number of designs, number of instances, and captime

Now we address Questions 5–7 from the introduction, investigating tradeoffs between the number of algorithm designs evaluated, the size of the benchmark set N, and the captime κ, given a time budget.

Q5: Given a fixed computational budget and a fixed captime, how should we trade off the number of designs evaluated* vs *the number of instances used in these evaluations?

To answer this question, we studied how the performance of *IS-Best*(N, κ, t) progressed as we increased the time t for three different values of N and fixed κ_{max}. In Figure 4, for each total amount of CPU time t available for algorithm design, we plot the performance that arose from using each of these three values of N.[5] The optimal number of training instances, N, clearly depended on the overall CPU time available,

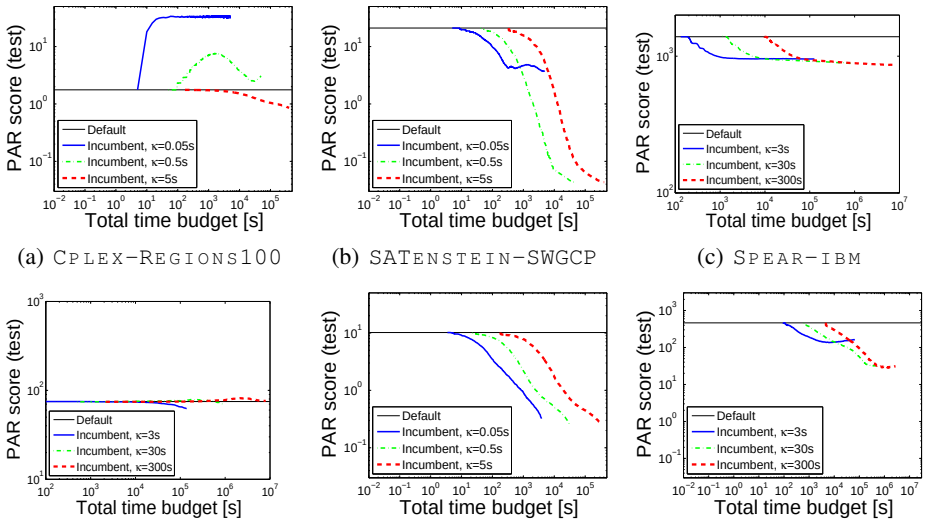
[5] The plots for $N = 1$ and $N = 10$ end at the point where all of the M given candidate algorithms have been evaluated. It would be appealing to extend the curves corresponding to lower values of N by considering more candidate designs $\boldsymbol{\Theta}_{additional}$. Unfortunately, this is impossible without filling in our whole input matrix for the new designs. This is because *IS-*

and the impact of using different values of N differed widely across the scenarios. For scenario CPLEX-REGIONS100 (see Figure 4(a)), using a single training instance ($N = 1$) yielded very poor performance throughout. For total time budgets t below about 50 000 seconds, the best tradeoff was achieved using $N = 10$. After that time, all $M = 1\,000$ designs had been evaluated, but using $N = 100$ yielded better performance. In contrast, for scenario SPEAR-IBM (see Figure 4(c)), using fewer instances led to rather good performance. For total time budgets below 3 000 seconds, using a single training instance ($N = 1$) actually performed best. For time budgets between 3 000 and about 70 000 seconds, $N = 10$ yielded the best performance, and only for larger budgets did $N = 100$ yield better results. For brevity, we do not discuss the remaining scenarios in depth, but rather summarize some highlights and general trends. Overall, $N = 1$ typically led to poor test performance, particularly for scenario CPLEX-ORLIB, which showed very pronounced overtuning. $N = 10$ yielded good performance for scenarios that showed quite stable relative rankings of algorithms across instances, such as CPLEX-REGIONS100 and SATENSTEIN-QCP (see Figures 4(a) and 4(e)). In contrast, for scenarios where the relative ranking of algorithms depended on the particular subset of instances used, such as SATENSTEIN-SWGCP, CPLEX-ORLIB and SPEAR-SWV, $N = 10$ led to overconfidence or even overtuning (see Figures 4(b), 4(d), and 4(f)). For the very heterogeneous instance set in scenario CPLEX-ORLIB, even using $P/2 = 70$ instances led to slight overtuning, yielding a design worse than the default. This illustrates the interesting point that even though the best sampled candidate algorithm design outperformed the CPLEX default for the full set (containing P instances, see Table 2), it was not possible to *identify* this design based on a training set of only $P/2$ instances.

Q6: Given a fixed computational budget and a fixed number of instances, how should we trade off the number of designs evaluated* vs *the captime used for each evaluation?

To answer this question, we studied how the performance of *IS-Best*(N, κ, t) progressed as we increased the time t for fixed $N = 100$ and three different values of κ (each scenario's "full" captime κ_{max}, as well as $\kappa_{max}/10$ and $\kappa_{max}/100$). To our best knowledge, this constitutes the first detailed empirical investigation of captime's impact on the outcome of empirical comparisons between algorithms. In Figure 5, for each total amount of CPU time t available for algorithm design, we plot the performance that arose from using each of these three values of κ. We observe that captime's impact depended on the overall CPU time available for algorithm design, and that this impact differed widely across the different scenarios. Scenario CPLEX-REGIONS100 was the only one for which low captimes led to very poor performance. In that scenario (see Figure 5(a)), captimes $\kappa_{max}/100$ and $\kappa_{max}/10$ led to overtuning and resulted in the selection of algorithm designs worse than the default, leaving κ_{max} as the preferred choice for any available time budget. In contrast, most other scenarios favoured smaller captimes. For example, in scenario SPEAR-IBM (see Figure 5(c)), the lowest captime, $\kappa_{max}/100$, led to extremely good performance and was the optimal choice for time budgets below $t = 10\,000$ seconds. For time budgets between 10 000 seconds and 800 000 seconds, the

Best(N, κ, t) averages across many different sets of N instances, and we would thus require the results of $\boldsymbol{\Theta}_{additional}$ for all instances. Furthermore, all curves are based on the same population of algorithm designs.



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

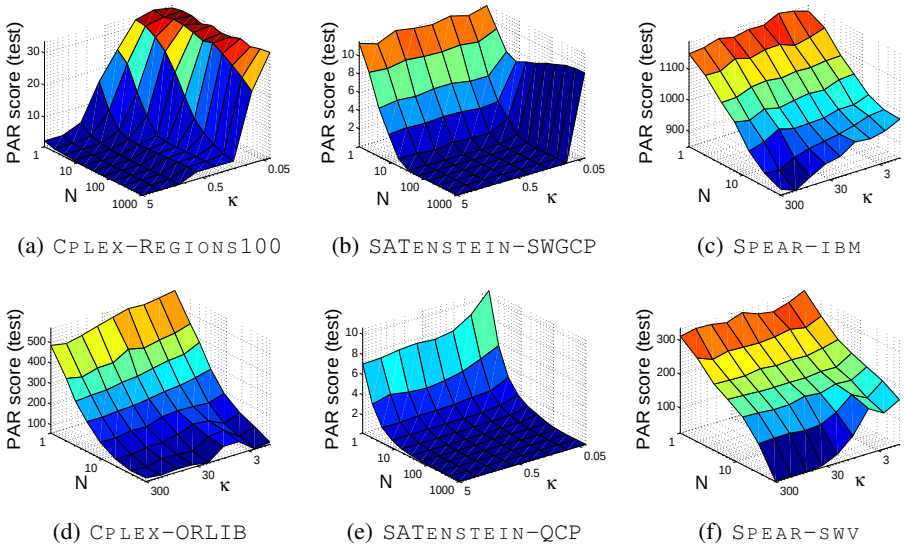
(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 5. Test performance (PAR for $P/2$ test instances) of *IS-Best*$(\min(P/2, 100), \kappa, t)$ for various values of κ. We plot graphs for each scenario's "full" captime κ_{max}, a tenth of it, and a hundredth of it. For reference, we plot test performance of the algorithm default.

optimal choice of captime was $\kappa_{max}/10$. Above that, all $M = 1\,000$ designs had been evaluated, and using a larger captime of κ_{max} yielded better performance. We summarize highlights of the other scenarios. SATENSTEIN-QCP is an extreme case of good performance with low captimes: using $\kappa_{max}/100$ yielded results very similar to those obtained with κ_{max}, at one-hundredth of the time budget. We observed a similar effect for SATENSTEIN-SWGCP for captime $\kappa_{max}/10$. For scenario CPLEX-ORLIB, captime $\kappa_{max}/100$ actually seemed to result in *better* performance than larger captimes; we believe that this was caused by a noise effect related to the small number of instances and the instability in the relative rankings of the algorithms with respect to different instances. It is remarkable that for the SPEAR-IBM scenario, which emphasizes very hard instances (Zarpas, 2005), captimes as low as $\kappa_{max}/100 = 3s$ actually yielded good results. We attribute this to the relative stability in the relative rankings of algorithms across different instances in that scenario (see Figures 1(c) and 2(c)): candidate designs that solved many instances quickly also tended to perform well when allowed longer runtimes.

Q7: Given a budget t for identifying the best of a fixed set of candidate designs, how many instances, N, and which captime, κ, should be used for evaluating each algorithm design?

To answer this question, we studied the performance of *IS-Best*(N, κ, ∞) for various combinations of N and κ. Figure 6 shows the test performance of these selected designs. Unsurprisingly, given unlimited resources, the best results were achieved with the maximal number of training instances ($N = P/2$) and the maximal captime ($\kappa = 5s$),

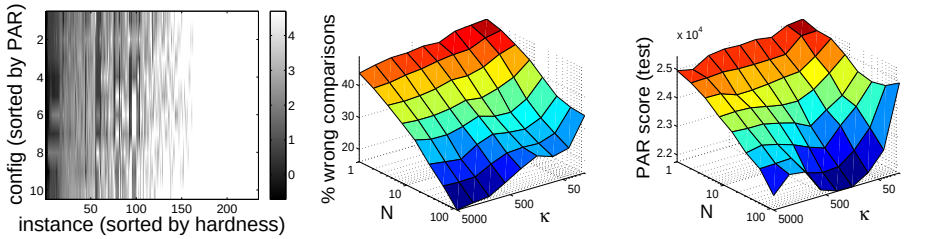


(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 6. Test performance (PAR for $P/2$ test instances) of *IS-Best*(N, κ, ∞) as depending on the number of training instances, N, and the captime, κ, used. Note that each combination of N and κ was allowed to evaluate the same number of $M = 1000$ candidate designs, and that the time required for each combination was thus roughly proportionally to $N \cdot \kappa$.

and performance degraded when either N or κ decreased. However, the *extent* to which performance degraded with lower N and κ—and therefore the optimal tradeoff between them—differed widely across our design scenarios. For example, in scenario CPLEX-REGIONS100, with a budget of 100 seconds for evaluating each candidate design, better performance could be achieved by using the full captime $\kappa_{max} = 5s$ and only $N = 20$ instances than with other combinations, such as, for example, $\kappa = 0.5s$ and $N = 200$. (To see this, inspect the diagonal of Figure 6(a) where $N \cdot \kappa = 100s$). In contrast, in most other scenarios using sufficiently many instances was much more important. For example, in scenario SPEAR-IBM, with a budget of 150 seconds for evaluating each candidate design, the best performance was achieved by using all $N = 50$ training instances and $\kappa = 3s$ (inspect the diagonal of Figure 6(c) where $N \cdot \kappa = 150s$). This effect was even more pronounced for the two SATENSTEIN scenarios. There, reductions of κ to a certain point seemed to have no negative effect on performance at all. We note that it is quite standard in the literature for researchers to set captimes high enough to ensure that they will rarely be reached, and to evaluate fewer instances as a result. Our findings suggest that more reliable comparisons can often be achieved by inverting this pattern, evaluating a larger set of instances with a more aggressive cutoff.



(a) Raw data: result of 10 solvers on $P = 234$ instances

(b) % wrong pairwise comparisons based on N training instances and captime κ

(c) performance of "best" solver; "best" is determined based on N training instances and captime κ

Fig. 7. Reliability of comparisons between solvers for results from 2007 SAT competition (scoring metric: total runtime across instances). The set of 234 instances was split equally and at random into training and test instances.

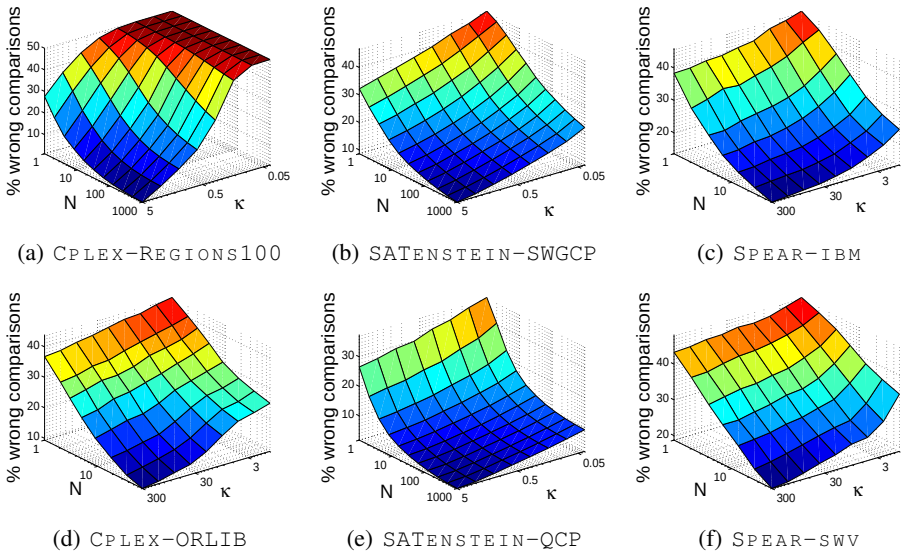
5 Tradeoffs in Ranking Candidate Algorithms

To answer Question 8 from the introduction, we now investigate the same tradeoff between number of training instances and captime studied in the previous section, but with the new objective of *ranking* candidate algorithms rather than to simply choosing one as the best. This problem arises in any comparative study of several algorithms in which our interest is not restricted to the question of which algorithm performs best.

One prominent example of such a comparative study is the quasi-annual SAT competition, one purpose of which is to compare new SAT solvers with state of-the-art solvers.[6] To demonstrate the versatility of our methods, we obtained the runtimes of the ten finalists for the second phase of the 2007 SAT competition on the 234 industrial instances from `http://www.cril.univ-artois.fr/SAT07/`. These runtimes constitute a matrix just like the input matrices we have used for our algorithm design scenarios throughout. Thus, we can employ our methods "out-of-the-box" to visualize the data. For example, Figure 7(a) shows the raw data; we observe a strong checkerboard patterns indicating a lack of correlation between the solvers.

As before, we split the instances into training and test instances, and used the test instances solely to obtain an unbiased performance estimate. Beyond the characteristics evaluated before, we computed the percentage of "wrong" pairwise comparisons, that is, those with an outcome opposite than the one by a pairwise comparison based on the test set and the "full" captime of $\kappa = 5000$s per run. Figure 7(b) gives this percentage for various combinations of N and κ. We can see that large cutoff times were indeed necessary in the SAT competition to provide an accurate *ranking* of solvers. However, Figure 7(c) shows that much lower captimes would have been sufficient to identify a *single* solver with very good performance. (In fact, in this case, the test performance of algorithms selected based on captimes of around 300 seconds was *better* than that based on the full captime of 5000 seconds. We hypothesize that this is due to the instability in the relative rankings of algorithms with respect to different instances, and

[6] `www.satcompetition.org`



(a) CPLEX-REGIONS100 (b) SATENSTEIN-SWGCP (c) SPEAR-IBM

(d) CPLEX-ORLIB (e) SATENSTEIN-QCP (f) SPEAR-SWV

Fig. 8. Ratio of pairwise comparisons based on N training instances and captime κ whose outcome is different than the outcome based on $P/2$ test instances and the "full" captime of the design scenario ($\kappa = 300s$ for CPLEX-ORLIB and the SPEAR scenarios, and $\kappa = 5s$ for the rest).

to the overall small number of instances. We also observed multiple crossings in the cumulative distributions of the number of instances solved for the ten algorithms considered here.) This analysis demonstrates that if one wanted to run the SAT competition within one-tenth of the CPU budget, it would be better to impose a tighter captime than to drop instances.

Q8: Given a time budget, how should we trade off N and κ for ranking a fixed set of algorithm designs?

We applied the same method to the six algorithm design scenarios studied throughout this paper, leading to the percentages of wrong comparisons shown in Figure 8. For a given domain and an overall time budget T, the optimal tradeoff between N and κ for ranking a set of M algorithm designs can be found by inspecting the diagonal of the corresponding figure characterized by $N \cdot \kappa = T/M$. Comparing Figure 8 to Figure 6, we see that in the context of ranking, performance tends to degrade more quickly for lower captimes than when the objective is to only identify the best algorithm.

6 Empirical Analysis of a *Predicted* Matrix of Runtimes

So far, we have discussed the application of our empirical analysis approach to a matrix of runtime data that is gathered offline, often at significant computational expense. We now demonstrate that the same methods can fruitfully be applied to matrices of *predicted* runtimes. In previous work we have shown how to construct predictive models that can yield surprisingly accurate predictions of runtime (Leyton-Brown et al.,

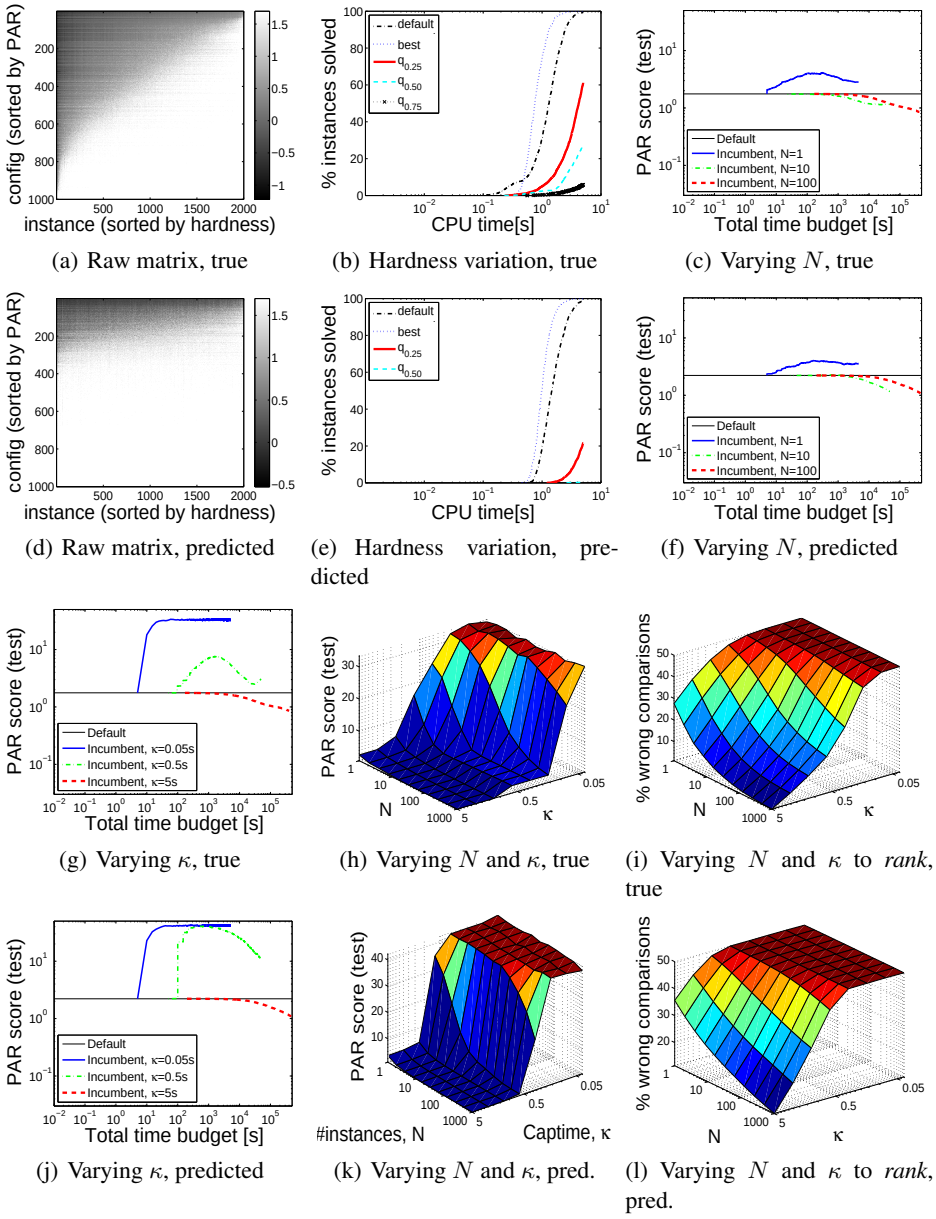
2002; Nudelman et al., 2004; Hutter et al., 2006; Xu et al., 2008; Leyton-Brown et al., 2009; Hutter, 2009). Here we consider the application of our newest family of models to the construction of a matrix of predicted runtimes, based on a much smaller training set of runtime information. This can be understood as an approximate version of the computationally-expensive offline analysis method we have discussed so far. Notably, this approximate method can be used *during the execution of an algorithm design procedure*, without the need for any additional algorithm runs to fill in missing entries of the runtime matrix. In this section we present a qualitative study of these approximations. Specifically, we compare our diagnostic plots derived from the true runtime matrix with ones derived from predicted runtime matrices. For brevity, we restrict our analysis to the scenarios that we discussed in the greatest depth above, CPLEX-REGIONS100 and SPEAR-IBM.

We begin by very briefly describing the prediction problem and the "random forest" models we used. (For more technical details and a complete discussion, see Hutter, 2009). For each SAT and MIP problem instance π, we had available a set of features for π (Leyton-Brown et al., 2002; Nudelman et al., 2004; Xu et al., 2008; Hutter, 2009). We augmented these with features of the algorithm design, namely the values of the (partially categorical) parameters in parameter vector $\boldsymbol{\theta}$. For each design scenario, we selected 10 000 training data points (combinations of π, $\boldsymbol{\theta}$, and the according runtime $r_{\pi,\boldsymbol{\theta}}$) uniformly at random from the $N \cdot P$ entries of the matrix and fitted a regression model that, given as input a $\langle \pi, \boldsymbol{\theta} \rangle$ pair, approximates runtime $r_{\pi,\boldsymbol{\theta}}$. For this task, we employed a random forest model with 10 regression trees. Each tree was built on a so-called *bootstrap sample*, a set of 10 000 data points sampled uniformly at random with repetitions from the original 10 000 training data points. For each split within each regression tree, we allowed a randomly-chosen fraction of $5/6$ of the input features (instance features and algorithm parameters), and did not further split nodes containing less than 10 data points. For each entry of the matrix, we used the mean prediction of the 10 regression trees.

Gathering the data to train our random forest models was substantially cheaper than gathering the data for the full matrix. In particular, for scenario SPEAR-IBM, the 10 000 data points amounted to 10% of the full matrix, and for CPLEX-REGIONS100 only to 0.5%. Note that our random forest models also work well when the instances are chosen in a biased way, as for example by an automated configuration procedure (Hutter, 2009). In design scenario CPLEX-REGIONS100, the construction of the random forest model took 20.3 CPU seconds, and in scenario SPEAR-IBM it took 19.6 CPU seconds. In either case, this is comparable to the time required for only a handful of algorithm runs.

In what follows we resist the temptation to quantitatively analyze the accuracy of our predictive models. This topic is explored at length in our past work, but is beyond the scope of the current paper. Our aim in the rest of this section is to show that our existing predictive runtime models can be combined with our empirical analysis methods, dramatically reducing their computational cost without substantially degrading their accuracy. Therefore, we restrict ourselves to qualitative analysis.

Overall, the plots based on the predictive matrix qualitatitvely resemble the ones based on the true matrix. For scenario CPLEX-REGIONS100, Figure 9 shows the diagnostic plots based on the true and the predicted runtime matrix (for convenience, the



(a) Raw matrix, true (b) Hardness variation, true (c) Varying N, true

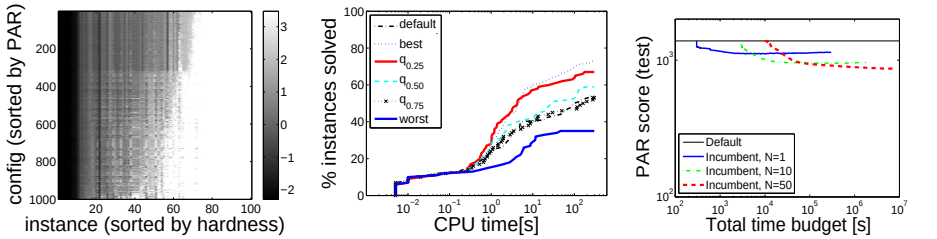
(d) Raw matrix, predicted (e) Hardness variation, predicted (f) Varying N, predicted

(g) Varying κ, true (h) Varying N and κ, true (i) Varying N and κ to *rank*, true

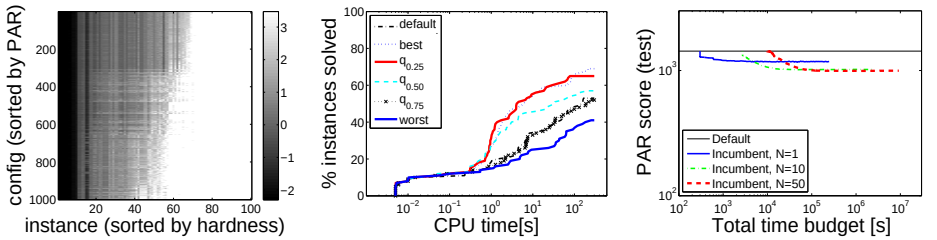
(j) Varying κ, predicted (k) Varying N and κ, pred. (l) Varying N and κ to *rank*, pred.

Fig. 9. Empirical analysis for algorithm design scenario CPLEX-REGIONS100, based on true and predicted matrices.

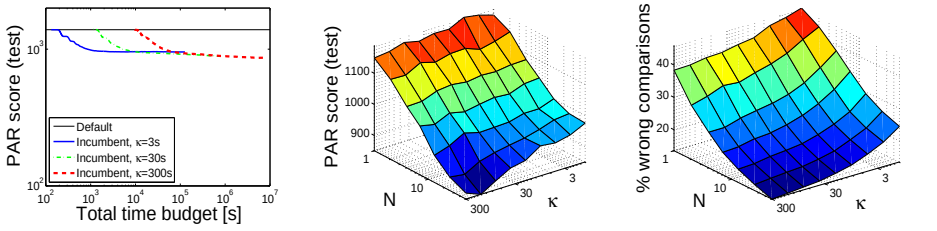
plots based on the true matrix are repeated from the various figures found elsewhere in this paper). The predicted matrix in Figure 9(d) was only roughly similar to the true runtime matrix in Figure 9(a); in particular, the prediction missed that even poor candidate designs solved some instances. Nevertheless, the remaining plots are qualitatively



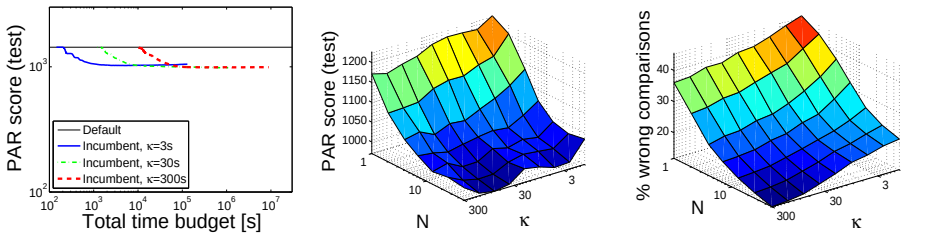
(a) Raw matrix, true (b) Hardness variation, true (c) Varying N, true



(d) Raw matrix, predicted (e) Hardness variation, predicted (f) Varying N, predicted



(g) Varying κ, true (h) Varying N and κ, true (i) Varying N and κ to *rank*, true



(j) Varying κ, predicted (k) Varying N and κ, pred. (l) Varying N and κ to *rank*, pred.

Fig. 10. Empirical analysis for algorithm design scenario SPEAR-IBM, based on true and predicted matrices.

quite similar. For scenario SPEAR-IBM, Figure 10 shows the equivalent plots based on the true and predicted matrix. The predicted matrix in Figure 10(d) closely resembles the true matrix in Figure 10(a). Note, however, that the predictive model slightly underestimated the percentage of instances the best-performing algorithm design could solve

when given a sufficiently high captime. (Compare Figure 10(b), where the best design solves around 73% of the instances with a captime of 300 seconds and Figure 10(e), where that number is below 70%.) This led to the incorrect prediction that higher captimes do not lead to the identification of better designs (compare Figure 10(k) to Figure 10(h)).

Results for the other CPLEX and SPEAR design scenarios were qualitatively similar; we omit the corresponding figures for brevity.[7] The only qualitative difference we observed is that for scenario CPLEX-ORLIB low captimes were predicted to perform worse than they actually did; this is the reverse situation than in scenario SPEAR-IBM, where *large* captimes were predicted to perform worse than they actually did.

To evaluate the effectiveness of our approximate empirical analysis, we summarize the answers it would have given to our eight questions for scenarios CPLEX-REGIONS100 and SPEAR-IBM and assess each answer's correctness.

1. *How much does performance vary across candidate algorithm designs?*
   CPLEX-REGIONS100: performance varies strongly: the best design solves all instances in 5 seconds, while the worst 60% do not solve any instance.
   SPEAR-IBM: performance varies less: the best design solves about 65% of the instances, whereas the worst design solves about 40%.
   Correctness: CPLEX-REGIONS100: performance indeed varied strongly, and the estimate for the best design is correct. Most designs, however, did solve some instances; only the very worst did not solve any. SPEAR-IBM: correct.
2. *How large is the variability in hardness across benchmark instances?*
   CPLEX-REGIONS100: for the best design, this variability is "only" about one order of magnitude.
   SPEAR-IBM: variability is much larger, above five orders of magnitude.
   Correctness: both correct.
3. *Which benchmark instances are useful for discriminating between candidate designs?*
   CPLEX-REGIONS100: all instances are useful.
   SPEAR-IBM: roughly 35% of the instances are infeasible for all candidate algorithm designs. About 10% are trivially solved by all designs.
   Correctness: both correct.
4. *Are the same instances "easy" and "hard" for all candidate designs?*
   CPLEX-REGIONS100: yes.
   SPEAR-IBM: largely, yes (some minor checkerboard patterns for poor designs).
   Correctness: both correct.
5. *Given a fixed computational budget and a fixed captime, how should we trade off the number of designs evaluated* vs *the number of instances used in these evaluations?*
   CPLEX-REGIONS100: $N = 1$ always performs worst, $N = 10$ is the optimal choice for the first $5\,000$ seconds, but for larger time budgets $N = 100$ yields better results.
   SPEAR-IBM: For total time budgets less than about $3\,000s$, $N = 1$ performs best, for about $3\,000s < t < 70\,000s$, $N = 10$ is best, above that $N = 100$ is best.
   Correctness: both correct.

[7] We have not yet constructed predictive models of algorithm runtime for SATENSTEIN since its many conditional parameters complicate model construction (see Hutter et al., 2009).

6. *Given a fixed computational budget, t, and a fixed number of instances, how should we trade off the number of designs evaluated* vs *the captime used for each evaluation?*
   CPLEX-REGIONS100: κ_{max} always performs best, regardless of the time budget.
   SPEAR-IBM: below about $t = 10\,000$ seconds, $\kappa_{max}/100$ performs best, for about $10\,000s < t < 800\,000$ seconds, $\kappa_{max}/10$ is best (for the last part basically the same as κ_{max}), above that κ_{max} is best.
   Correctness: both correct.
7. *Given a budget for identifying the best of a fixed set of candidate designs, how many instances, N, and which captime, κ, should be used for evaluating each algorithm design?*
   CPLEX-REGIONS100: within the bounds studied here (N from 1 to 1 000 and κ from $0.05s$ to $5s$), for any time budget t above about 50 seconds per evaluation, it is always preferable to use a captime as large as possible. For very low time budgets, it is better to use captimes around $\kappa = 1.25$ seconds to enable the evaluation on a larger number of instances.
   SPEAR-IBM: within the bounds studied here (N from 1 to 50 and κ from $3s$ to $300s$), it is almost always preferable to use an N as large as possible. The exception is a ridge around $\kappa = 19s$, which causes $\kappa = 38s$ in combination with a halved N to be preferred.
   Correctness: CPLEX-REGIONS100: correct. SPEAR-IBM: largely correct; only the predicted exception around $\kappa = 19s$ is incorrect. We found it remarkable that these predictions are as good as they are, given that the predicted plots look rather different from the true plots.
8. *Given a time budget, how should we trade off N and κ for ranking a fixed set of algorithm designs?*
   CPLEX-REGIONS100: for the bounds studied here, it is always preferable to use a larger κ and a correspondingly smaller N.
   SPEAR-IBM: for the bounds studied here, it is always preferable to use a larger N and a correspondingly smaller κ.
   Correctness: both correct.

Overall, we note that the answers given by our approximate empirical analysis approach were surprisingly similar to those given by the computationally-expensive offline approach. Thus, the techniques presented in this article can often be applied using dramatically smaller amounts of computationally-expensive runtime data.

7 Conclusions and Future Work

We have proposed an empirical analysis approach for studying the tradeoffs faced in evaluating the relative performance of a set of candidate algorithms. We applied this approach to six rich algorithm design scenarios based on highly-parameterized, state-of-the-art algorithms for satisfiability and mixed integer programming. Our analysis answers a wide variety of questions pertaining to the performance variation across both candidate algorithms and instances. For each design scenario, we showed how to best

trade off the number of designs considered, the number of instances used for the evaluation, and the computation time allowed for each run. We showed that the six design scenarios we studied have very different characteristics, suggesting that different algorithm design procedures would work well for them.

Our analytic tools can be used "out of the box" for new domains of interest; the only input required is a simple matrix of runtimes for each combination of candidate algorithm and benchmark instance of interest. We provide source code at `http://www.cs.ubc.ca/labs/beta/Projects/AlgoDesign/`. While the gathering of this runtime matrix is typically computationally expensive, we demonstrated that our approach can also be applied to a matrix of *predicted* runtimes, which enables a computationally much cheaper, approximate analysis.

We believe that the qualitative differences that we observed across the different algorithm design scenarios considered here can be exploited by automated design procedures, and plan to investigate this in future work. We plan to use the analytic approach introduced here to characterize the type of scenarios for which a given algorithm design method works well. We expect that such a characterization would be useful for making online decisions about which strategy to use in a given algorithm design scenario, and whether instance-based algorithm selection techniques should be applied. Thus, we hope to substantially improve current state-of-the-art algorithm design methods.

8 Acknowledgements

We thank Thomas Stützle for valuable comments on an early version of this work. Thanks also to Ashiqur KhudaBukhsh for help with SATenstein.

Bibliography

Adenso-Diaz, B. and Laguna, M. (2006). Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114.

Audet, C. and Orban, D. (2006). Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 17(3):642–664.

Babić, D. and Hu, A. J. (2007). Structural Abstraction of Software Verification Conditions. In W. Damm, H. H., editor, *Computer Aided Verification: 19th International Conference, CAV 2007*, volume 4590 of *LNCS*, pages 366–378. Springer Verlag, Berlin, Germany.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In Bartz-Beielstein, T., Aguilera, M. J. B., Blum, C., Naujoks, B., Roli, A., Rudolph, G., and Sampels, M., editors, *4th International Workshop on Hybrid Metaheuristics (MH'07)*, pages 108–122.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, Germany.

Beasley, J. (1990). OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072.

Birattari, M. (2005). *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. DISKI 292, Infix/Aka, Berlin, Germany.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In Langdon, W. B., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *Proc. of GECCO-02*, pages 11–18. Morgan Kaufmann, SF, CA, USA.

Etzioni, O. and Etzioni, R. (1994). Statistical methods for analyzing speedup learning experiments. *Journal of Machine Learning*, 14(3):333–347.

Gent, I. P., Hoos, H. H., Prosser, P., and Walsh, T. (1999). Morphing: Combining structure and randomness. In Hendler, J. and Subramanian, D., editors, *Proc. of AAAI-99*, pages 654–660, Orlando, Florida. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gomes, C. P. and Selman, B. (1997). Problem structure in the presence of perturbations. In Kuipers, B. and Webber, B., editors, *Proc. of AAAI-97*, pages 221–226. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Gomes, C. P. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62.

Gratch, J. and Dejong, G. (1992). Composer: A probabilistic solution to the utility problem in speed-up learning. In Rosenbloom, P. and Szolovits, P., editors, *Proc. of AAAI-92*, pages 235–240. AAAI Press / The MIT Press, Menlo Park, CA, USA.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer Verlag.

Hoos, H. H. (2008). Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Department of Computer Science.

Horvitz, E., Ruan, Y., Gomes, C. P., Kautz, H., Selman, B., and Chickering, D. M. (2001). A Bayesian approach to tackling hard computational problems. In Breese, J. S. and Koller, D., editors, *Proc. of UAI-01*, pages 235–244. Morgan Kaufmann, SF, CA, USA.

Hutter, F. (2009). *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, Department of Computer Science, Vancouver, Canada.

Hutter, F., Babić, D., Hoos, H. H., and Hu, A. J. (2007a). Boosting Verification by Automatic Tuning of Decision Procedures. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34, Washington, DC, USA. IEEE Computer Society.

Hutter, F., Hamadi, Y., Hoos, H. H., and Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. In Benhamou, F., editor, *Proc. of CP-06*, volume 4204 of *LNCS*, pages 213–228. Springer Verlag, Berlin, Germany.

Hutter, F., Hoos, H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *JAIR*, 36:267–306.

Hutter, F., Hoos, H. H., and Stützle, T. (2007b). Automatic algorithm configuration based on local search. In Howe, A. and Holte, R. C., editors, *Proc. of AAAI-07*, pages 1152–1157. AAAI Press / The MIT Press, Menlo Park, CA, USA.

ILOG Inc. (2008). ILOG CPLEX, The World's Leading Mathematical Programming Optimizers. `http://www.ilog.com/products/cplex/`. Version last visited on April 29, 2009.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2009). Satenstein: Automatically building local search SAT solvers from components. In *Proc. of IJCAI-09*, pages 517–524.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2002). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In Hentenryck, P. V., editor, *Proc. of CP-02*, volume 2470 of *LNCS*, pages 556–572. Springer Verlag, Berlin, Germany.

Leyton-Brown, K., Nudelman, E., and Shoham, Y. (2009). Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52.

Leyton-Brown, K., Pearson, M., and Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In Jhingran, A., Mason, J. M., and Tygar, D., editors, *Proc. of EC-00*, pages 66–76, New York, NY, USA. ACM.

Maron, O. and Moore, A. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Proc. of NIPS-94*, volume 6, pages 59–66. Morgan Kaufmann, SF, CA, USA.

Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1):1–40.

Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A., and Shoham, Y. (2004). Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of CP-04*, LNCS. Springer Verlag, Berlin, Germany.

Segre, A., Elkan, C., and Russell, A. (1991). A critical look at experimental evaluations of EBL. *Journal of Machine Learning*, 6(2):183–195.

Simon, L. and Chatalic, P. (2001). SATEx: a web-based framework for SAT experimentation. In Kautz, H. and Selman, B., editors, *Proc. of SAT-01*.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606.

Zarpas, E. (2005). Benchmarking SAT Solvers for Bounded Model Checking. In Bacchus, F. and Walsh, T., editors, *Proc. of SAT-05*, volume 3569 of *LNCS*, pages 340–354. Springer Verlag.