Ann Math Artif Intell (2006) 47:295–328
DOI 10.1007/s10472-006-9036-z

Learning dynamic algorithm portfolios

Matteo Gagliolo · Jürgen Schmidhuber

Published online: 26 January 2007

Abstract Algorithm selection can be performed using a model of runtime distribution, learned during a preliminary training phase. There is a trade-off between the performance of model-based algorithm selection, and the cost of learning the model. In this paper, we treat this trade-off in the context of bandit problems. We propose a fully dynamic and online algorithm selection technique, with no separate training phase: all candidate algorithms are run in parallel, while a model incrementally learns their runtime distributions. A redundant set of *time allocators* uses the partially trained model to propose machine time shares for the algorithms. A bandit problem solver mixes the model-based shares with a uniform share, gradually increasing the impact of the best time allocators as the model improves. We present experiments with a set of SAT solvers on a mixed SAT-UNSAT benchmark; and with a set of solvers for the Auction Winner Determination problem.

Keywords algorithm selection · algorithm portfolios · online learning · life-long learning · bandit problem · expert advice · survival analysis · satisfiability · constraint programming

Mathematics Subject Classifications (2000) 68T05 · 68T20 · 62N99 · 62G99

This work was supported by SNF grant 200020-107590/1.

M. Gagliolo (✉) · J. Schmidhuber
IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
e-mail: matteo@idsia.ch

M. Gagliolo · J. Schmidhuber
Faculty of Informatics, University of Lugano, Via Buffi 13, 6904 Lugano, Switzerland

J. Schmidhuber
TU Munich, Boltzmannstr. 3, 85748 Garching, München, Germany
e-mail: juergen@idsia.ch

1 Motivation

Most problems in AI can be solved by more than one algorithm. Most algorithms feature a number of parameters that have to be set. Both choices can dramatically affect the quality of the solution, and the time spent obtaining it. Algorithm Selection [63], or *Meta-learning* [75] techniques, address these questions in a machine learning setting. Based on a training set of performance data for a large number of problem instances, a model is learned that maps (*problem*, *algorithm*) pairs to expected performance. The model is later used to select and run, for each new problem, only the algorithm that is expected to give the best results.

A generalization of algorithm selection, inspired by the *Algorithm Portfolio* paradigm [33], is to use the model to select a *subset* of the available algorithms, and run them in parallel until the fastest one solves the problem. For some classes of algorithms, with a "heavy-tailed" runtime distribution, the execution of multiple parallel runs differing only for the random seed, can actually have an advantage over a single run [25]. In any case, only a fraction of the computation time will be spent on the fastest solver.

These approaches, though preferable to the far more popular "trial and error", pose a number of problems:

1. *Training set representativeness.* Problem instances encountered during the training phase are assumed to be statistically representative of successive ones. This hypothesis is practically unavoidable for any model-based selection technique, if referred to a single instance.
2. *Static selection.* The actual algorithm performance on a given problem is assumed to be predictable with sufficient precision before even *starting* the algorithm. This assumption is often violated by stochastic algorithms, whose performance can exhibit large fluctuations across different runs (see, e. g., Section 7.1, or [25]).
3. *Training cost.* Generating the training data obviously requires solving each training problem repeatedly, at least once for each of the algorithms. The computational cost of this initial training phase is neglected, even though it can be high enough to make algorithm selection impractical.

One common trait of the problems listed above, is that they can be related to the lack of feedback information from the actual execution of the chosen algorithms. Such a *dynamic* feedback can be used to update the model's predictions, and adapt the computational resource allocation accordingly, allowing for a finer distinction among problem instances (problem 2). It can also be used to guide the training phase itself, avoiding exceedingly long runs of inefficient algorithm/problem combinations (problem 3).

A step in this direction can be taken using a *Dynamic Algorithm Portfolio* (DAP) [17, 18, 21, 59]. Instead of *first* choosing a portfolio and *then* running it, a DAP iteratively allocates a time slice that is shared among all the available algorithms, and updates the relative algorithm priorities, based on their current state, in order to favor the most promising ones. To this aim, a model is needed to map (*problem*, *algorithm*, *current algorithm state*) triples to the *expected time* to solve the problem.

To reduce training cost, the artificial boundary between training and usage should be dropped, adopting an *online* learning technique: after the first problem is solved, the model is updated, and used to guide the solution of the next problem.

In previous work, we termed this approach *Adaptive Online Time Allocation* (AOTA). In [21], we presented an *oblivious* time allocator, with no knowledge transfer across problem instances. Runtime predictions, evaluated by extrapolating recent performance improvements, were mapped to time allocation for the next time slice, based on a simple "ranking" heuristic. In [17] we proposed a method for learning a probabilistic model *online*, while solving a problem sequence. The model was conditioned on features of both the problems and the algorithms (parameter values, current state). The downside of introducing knowledge transfer across problem instances was that the model would obviously be unreliable during the initial portion of the problem sequence. Time was then allocated according to a modification of the ranking heuristic: the first problem was solved with a uniform share, and the impact of the model on the time allocation was gradually increased through the sequence of tasks, according to a fixed schedule, independent of model performance.

In this work we keep the same *dynamic online* philosophy, but we separate the two problems of allocating time based on runtime predictions, and grading the impact of model-based allocation, giving a sound solution for both. In the following we briefly present some related work (Section 2), distinguishing between *static* techniques, in which the selection is performed before runtime, and *dynamic* ones, where the selection process is somehow adapted during the actual execution of the algorithms. We then introduce some simple concepts from survival analysis, which are relevant to our method, and to algorithm performance modeling in general. Section 4 describes an ideal implementation of a static portfolio, based on exact knowledge of the runtime distributions of the algorithms, illustrating different optimality criteria to share machine time among the algorithms. Section 5 introduces the *dynamic* extension, and the online learning scheme, discussing the exploration-exploitation trade-off determined by the online setting. In Section 6, we address this trade-off in the context of bandit problems [1], and present our new time allocator (TA) GambleTA. Section 7 analyzes experimental results on two challenging algorithm selection problems. In the first set of experiments, a local search and a complete SAT solver are controlled during the solution of a sequence of random satisfiable and unsatisfiable problems. In the second, we compare with results of a static algorithm selection approach [45], on a set of combinatorial Auction Winner Determination problems. Section 8 discusses originality, limitations, and viable improvements of GambleTA.

2 Related work

Many algorithm selection, or parameter tuning, techniques, are tailored to a specific algorithm, and often present similar interesting solutions across different fields of research. We will give some examples of these, but we will keep our focus on "black box" techniques, that can be applied in more general settings.

We will first introduce some naming conventions. A first distinction needs to be made among *decision* problems, in which a binary criterion for recognizing a solution is available; and *optimisation* problems, in which different levels of solution quality can be attained, measured by an *objective* function [29]. A decision problem can be viewed as an optimisation problem with a binary objective function; an optimisation problem can be turned into a decision problem, if a reachable target

value of performance can be set in advance. Literature on algorithm selection is often focused on one of these two classes of problems. The selection is normally aimed at maximizing performance quality for optimisation problems; and at minimizing solution time for decision problems.

The selection among different algorithms can be performed once for an entire set of problem instances (*per set* selection, following the terminology of [34]); or repeated for each instance (*per instance* selection). A further independent distinction [59] can be made among *static* algorithm selection, in which any decision on the allocation of resources precedes algorithm execution; and *dynamic*, or *reactive*, algorithm selection, in which the allocation can be adapted during algorithm execution.

Another orthogonal feature is related to learning. Here we borrow from the machine learning terminology, distinguishing between *offline* or *batch* learning techniques, in which there is a separate training phase, after which the selection criteria are kept fixed; and *online*[1] or *life-long* learning [62] techniques, in which the criteria are updated at every instance solution. *Oblivious* algorithm selection techniques do not transfer any knowledge across different problem instances.

2.1 Static algorithm selection

A seminal paper in this field is [63], in which offline, per instance algorithm selection is first advocated, both for decision and optimisation problems. More recently, similar concepts have been proposed, with different terminology (algorithm *recommendation*, *ranking*, *model selection*), by the *Meta-Learning* community [15, 23, 42, 75]. For example, in [70], different values for the kernel parameter of a Support Vector Machine [74] are evaluated on different training data sets. Each data set is described through a set of features. For an unseen data set, the features are first evaluated, and a *ranking* of the kernel parameter values is induced, using a k-nearest-neighbor estimate of performance, based on distance in feature space between the new data set, and the ones used for training.

Usually, meta-learning research deals with optimisation problems, and is focused on maximizing solution quality, without taking into account the computational aspect. An interesting exception is offered by *landmarking* techniques [61] in which the performances of fast base-learners, not included in the algorithm set, are used as task features, in order to obtain a better discrimination of task difficulty.

Works on *Empirical Hardness Models* [45, 46, 56, 57] are instead applied to decision problems, and focus on obtaining accurate models of runtime performance, conditioned on numerous features of the problem instances, as well as on parameters of the solvers [34, 35]. The models are used to analyze this performance, or to generate harder benchmarks, but also to perform algorithm selection on a per instance basis. Online selection is advocated in [34].

Literature on algorithm portfolios [24, 33, 60] is usually focused on choice criteria for building the set of candidate solvers, such that their areas of good performance

[1] In previous works [17, 21], the terms "offline" and "online" were used to distinguish among static and dynamic approaches, but we found this nomenclature to be misleading, especially for the machine learning community.

don't overlap; and optimal static allocation of computational resources among elements of the portfolio.[2]

Other interesting research areas, in which both solution quality and computational aspects are taken into account, include *anytime algorithm scheduling* [8], and *time limited planning* [14, 32, 65, 66]), in which time is allocated sequentially to a set of planning primitives (e. g., finding the path to a goal) and the subsequent actions exploiting the decisions taken (e. g., following the chosen path), in order to obtain a good compromise between solution quality and time spent computing it.

Bandit problem solvers (BPS) [1, 6], can in principle be applied to static per set algorithm selection, considering each available algorithm as an arm and runtime as a loss, to be minimized (see also Section 2.2, Section 6, [16, 20]). As an alternative, one can consider the use of a BPS to solve selection problems on a per instance basis, in an oblivious setting, as in [11, 12, 73], where the *Max K-armed bandit problem* is presented, and solvers for this game are used to maximize performance quality.

In [20], we presented an online method for learning a per set estimate of an optimal restart strategy (GambleR). The method consists in alternating the universal strategy of [51], and an estimated optimal strategy, again based on [51]. The estimate is performed according to a model of runtime distribution on the set of instances, updated at every solution. Here the bandit problem solver is used at an upper level, to allocate runs of the two strategies: a similar approach will be taken in this work, to weight the decisions of different time allocators (Section 6).

The classification of Racing Algorithms [7, 53] as static or dynamic depends on the definition of a problem instance. In these works, the algorithm set contains different parametrizations of a given supervised algorithm. Each is repeatedly run on a sequence of increasingly large leave-one-out training sets, which can be seen as a sequence of related problems; after a problem is solved, badly performing algorithms are discarded if statistically sufficient evidence is gathered against them, such that machine time is shared among fewer algorithms on next problem.

Search in program space can also be formalized as an algorithm selection problem. For example, the algorithm set of the Optimal Ordered Problem Solver [68] may include all programs of a universal programming language. Time is allocated to these programs proportionally to a probability distribution that is updated when a problem is solved. Other interesting program search techniques include Genetic Programming [13] and Probabilistic Incremental Program Evolution [67].

2.2 Dynamic algorithm selection

A number of interesting dynamic exceptions to the static selection paradigm have been proposed recently. In [31], algorithm performance modeling is based on the behavior of the candidate algorithms during a predefined amount of time, called the *observational horizon*. Each algorithm is run on each training problem, with a high enough cutoff time, and features are extracted from the dynamic data recorded during this initial period. Runs are distinguished as belonging to two classes of "short" and "long" experiments, using the median of runtimes as a decision threshold. A

[2] With the term algorithm portfolio, we always refer to the parallel execution of (a subset of) the members of the portfolio. In other works (e. g., [45]), the term is also referred to the algorithm set from which single algorithm selection is performed.

mapping is learned from the static and dynamic features to the correct classification labels. The same approach is used in [40] to implement dynamic context-sensitive restart policies for SAT solvers: the authors assume that the runtime distribution of their algorithm is not known in advance, but belongs to a known finite set of distributions, from which the correct one can be discriminated based on dynamic features.

Algorithmic chaining [9] executes a predetermined sequence of Constraint Programming solvers, using an ad-hoc mechanism to decide when to switch to next algorithm, according to a prediction of "thrashing" behavior, given the current state. This can be viewed as a dynamic portfolio, but all its components are fixed, designed based on a priori expertise.

In *anytime algorithm monitoring* [26], the *dynamic performance profile* of a planning technique is updated according to its performance, in order to stop the planning phase when further improvements in the planned action sequence are not worth the time spent evaluating them. Also in this case, both the quality of a solution and its computational cost are taken into account.

In [71], the author presents a collection of ideas for solving sequences of time-limited optimisation problems by searching in a space of problem solving techniques, allocating time to them according to their probabilities, and updating the probabilities according to positive and negative results.

In a Reinforcement Learning [38] setting, algorithm selection can be formulated as a Markov Decision Process: in [44], the algorithm set includes sequences of recursive algorithms, formed dynamically at run-time solving a sequential decision problem, and a variation of Q-learning is used to find a dynamic algorithm selection policy; in [58, 59], from which we borrow some terminology, a set of deterministic algorithms is considered, and, under some limitations, static and dynamic schedules are obtained, based on dynamic programming. Success Story algorithms [69] can undo policy modifications that did not improve the reward rate. A simple reinforcement learning feedback mechanism is used at runtime in [3] to adapt the size of the prohibition list of a tabu-search algorithm.

Some dynamic selection methods are *oblivious*, i. e., are characterized by the absence of any knowledge transfer across problem instances.

The "parameterless GA" [27] may be viewed as a specialized heuristic for dynamic selection. It consists of a sequence of simple generational Genetic Algorithms [28], with exponentially spaced population sizes, generated and executed according to a fixed interleaving schedule that assigns more runtime to smaller populations. Once a small population converges, or a larger one achieves a higher average fitness, the small one is discarded.

"Low-knowledge" approaches can be found in [4, 10], in which various simple indicators of current solution improvement are used for algorithm selection, in order to achieve the best solution quality within a given time contract. In [4], all available algorithms are run for a fraction of the contract, and a performance predictor is then used to select a single one for the remaining time. In [10], the selection process is iterated: machine time shares are based on a recency-weighted average of performance improvements. This latter oblivious technique is actually a simple solver for time-varying bandit problems, here applied on a per instance basis.

In [21] we adopted a similar approach. We considered algorithms with a scalar state, that had to reach a target value. The time to solution was estimated based

on a shifting-window linear extrapolation of the learning curves: a recency-weighted average was tried at first, but its results were not competitive with the comparison term [27].

3 Algorithm survival analysis

This paper is focused on *decision* problems, in which a binary criterion for recognizing a solution is available. In this case, performance modeling aims at predicting the *runtime*, i. e., the time to solve a problem. More precisely, consider a randomized algorithm solving a given problem instance, or, equivalently, a randomized or deterministic algorithm solving a randomly selected problem instance. In both cases, the runtime spent before finding a solution can be treated as a random variable T, described by its *cumulative distribution* function (CDF), $F(t) = \Pr\{T \leq t\}$, $F : [0, \infty) \to [0, 1]$, representing the probability that a solution is found within a time t. This function is referred to as the *runtime distribution* (RTD) in literature about algorithm performance modeling (see, e. g., [29]).

A large corpus of research, known under the name of *survival analysis*[3] [37, 54], is devoted to the modeling of events in time. In this section, we briefly review the basic concepts and terminology in these fields, and discuss their application to algorithm performance modeling.

We start by noting a difference between the events of interest in survival analysis, typically death, or failure, and problem solution: the latter does not necessarily have to happen. This can be described by a RTD with $F(\infty) < 1$. The resulting *probability density* function (pdf), defined as $f(t) = dF(t)/dt$, is *improper*, i. e., its integral over $[0, \infty)$ does not sum to 1. In this situation, the expected runtime is ∞, and the usual formulation

$$E\{T\} = \int_0^\infty t f(t) dt \tag{1}$$

cannot be applied. A *quantile* t_α of a the RTD, defined as the time at which F intercepts the value α, can still be evaluated, solving the equation

$$t_\alpha = F^{-1}(\alpha), \quad \alpha \in [0, F(\infty)]. \tag{2}$$

Lifetime distributions are often described in terms of the *survival* function

$$S(t) = 1 - F(t), \tag{3}$$

representing, in our case, the probability that the algorithm is still "alive" and running at time t.

[3] This is the most widely used term, in medicine, biostatistics, biology, but different application fields use other terms. Engineers modeling the duration of a device speak of *failure analysis*, or *reliability theory*. Actuaries setting premiums for insurance companies use the term *actuarial science*.

Another ubiquitous concept in survival analysis is the *hazard* function $h(t)$, quantifying the instantaneous probability of occurrence of the event of interest at time t, given that it was not observed earlier:

$$h(t) = \lim_{\Delta t \to 0} \Pr\{T \leq t + \Delta t | T > t\} = \frac{f(t)}{1 - F(t)} = \frac{f(t)}{S(t)}, \tag{4}$$

where $f(t)/S(t) = f(t|T > t)$ is the pdf conditioned on observed survival until time t.

The integral of (4) is termed *cumulative* hazard, and can be shown to have the following relationship with the survival function:

$$H(t) = \int_0^t h(\tau) d\tau = \int_0^t \frac{dF(\tau)}{S(\tau)} = -\ln S(t), \tag{5}$$

or $S(t) = \exp(-H(t))$.

3.1 Censored sampling

A typical problem that survival analysts have to face is the *incompleteness* of the data. For example, in biostatistics and medicine, patients might "drop-out" a group of study: in this case, only a lower bound on their lifetime would be known. A sample containing incomplete data is referred to as a *censored* sample. In failure analysis [54], censoring is normally the result of experimenter's decisions, aimed at reducing the duration of an experiment. For example, in estimating a duration model of a newly produced light bulb, an engineer could leave a large number of prototypes turned on for a predetermined period of time (*type I* censored sampling): in this case the number of observed failures is a random variable, related to the lifetime distribution of the bulbs. As an alternative, the experiment could end as soon as a predetermined number of bulbs has gone off (*type II* censored sampling). In this case, the duration of the experiment is a random variable. In both cases, only a lower bound on failure time would be available for the surviving bulbs. Unless the engineer is willing to wait for years, or the new product is quite cheap, this incomplete data will constitute a large portion of the collected sample. The precision of the model would clearly be affected. In other words, there is a *trade-off* between the duration of the experiment, and the precision of the obtained model: in any case, discarding incomplete data can result in an extremely biased model.

In algorithm performance modeling, type I censoring is typically performed, imposing a threshold on runtime. Also in this case there is a trade-off between training time and model precision. In the context of algorithm selection techniques, this trade-off should rather be measured between training time and the gain in performance resulting from the use of the learned model: in this sense, the required precision can be much lower than expected. We give an example in [19] where this trade-off is analyzed in the context of restart strategies, reporting the training times, and resulting performance, of model-based restart strategies, learned with different levels of censoring.

The treatment of censored data differs in the *parametric* and *non-parametric* settings. When fitting a parametric model $f(t|\theta)$, a censored runtime t_c can be taken

into account by expressing the likelihood of the parameter θ, given this piece of data, as the survival probability at time t_c.

$$\mathfrak{L}_c(t_c|\theta) = \int_{t_c}^{\infty} f(\tau|\theta)d\tau = [1 - F(t_c|\theta)] = S(t_c|\theta). \tag{6}$$

In *nonparametric* methods [41, 72], estimates are based solely on the data observed so far. The simplest nonparametric method is the empirical CDF,

$$\hat{F}(t) = \sum_{t_i<t} \frac{1}{n}. \tag{7}$$

In this setting, censored samples can be taken into account by distinguishing between the number of events recorded, and the number of individuals observed "at risk" (in our case: still running), at a time t. This is the essence of the Kaplan–Meier estimator of the hazard function [39]:

$$\hat{h}(t) = \frac{\sum_{t_i=t,\nu_i=1} 1}{\sum_{t_i\geq t} 1}, \tag{8}$$

where ν_i is the *event indicator*, and is 1 for uncensored observations, and 0 for censored ones.

In these and other nonparametric methods, $F(t)$, $S(t)$, $H(t)$ are "stepwise" functions, that change only at uncensored observations $\{t_i|\nu_i = 1\}$, and are defined until the largest one; while $f(t)$, $h(t)$ are pulse trains, i. e., are 0 everywhere, but with a positive integral across the observation values t_i. For example, a non-parametric hazard function can be represented as $h(t) = \sum_i h_i\delta(t - t_i)$, where h_i is the hazard (8) at t_i, and the corresponding cumulative hazard function is $H(t) = \sum_{t_i<t} h_i$. In order to obtain meaningful predictions also for $t \notin \{t_i\}$, hazard or density estimates can be *smoothed* [76].

3.2 Conditional models

Conditional estimates [5] take into account *covariate* or *feature* values x for each individual. If *dynamic* information about the algorithm is also available, this can be treated as a *time-varying* covariate, or *longitudinal* data [43, 49, 55], to update an estimated RTD. The simplest time-varying covariate is time itself: if an algorithm is still running at a time y, the RTD for the rest of the run can be evaluated by simply shifting and scaling the original F

$$F(t|T > y) = \frac{F(t) - F(y)}{1 - F(y)} = \frac{F(t) - F(y)}{S(y)}, \tag{9}$$

defined only for $t > y$. Given the definition of the hazard function (4), its formula does not change, while the cumulative hazard becomes:

$$H(t|T > y) = \int_y^t h(\tau)d\tau. \tag{10}$$

Both cases can be represented in the non-parametric setting, simply discarding hazard values h_i with $t_i \leq y$.

In the next section, we will apply the simple notions described here, to propose different optimisation criteria for a static algorithm portfolio. Literature on survival analysis is obviously much richer than this. Recent research is facing challenging applications, and developing advanced estimation techniques, with Bayesian methods playing a major role [36]. For example, biostatisticians working on gene expression data [50] have to deal with *thousands* of time-varying covariates, and often very small and censored samples. Both algorithm performance modeling, and model-based algorithm selection, can profit from this field of research: for selection, also the computational complexity of modeling should be taken into account.

4 Static algorithm portfolios

Consider now a portfolio of K algorithms $\mathcal{A} = \{a_1, a_2, ..., a_K\}$, solving the same problem instance in parallel, and sharing the computational resources of a single machine according to a share $\mathbf{s} = \{s_1, .., s_K\}$, $s_k \geq 0$, $\sum_{i=1}^{K} s_k = 1$, i. e., for any amount t of machine time, a portion $t_k = s_k t$ will be allocated[4] to a_k. An a_k that can solve the problem in a time t_k if run alone, will spend a time $t = t_k/s_k$ if run with a share s_k. If the runtime distribution $F_k(t_k)$ of a_k on the current problem is available, one can obtain the distribution $F_{k,s_k}(t)$ of the event "a_k solved the problem after a time t, using a share s_k", by simply substituting $t_k = s_k t$ in F_k:

$$F_{k,s_k}(t) = F_k(s_k t). \tag{11}$$

If the execution of all the algorithms is stopped as soon as one of them solves the problem, as in Type II censored sampling (Section 3), the resulting duration of the solution process is a random variable, representing the runtime of the parallel portfolio. Its distribution $F_{\mathcal{A},\mathbf{s}}(t)$ can be evaluated based on the share $\mathbf{s}$, and the $\{F_k\}$. The evaluation is more intuitive if we reason in terms of the survival distribution: at a given time t, the probability $S_{\mathcal{A},\mathbf{s}}(t)$ of *not* having obtained a solution is equal to the joint probability that no single algorithm a_k has obtained a solution within its time share $s_k t$. Assuming that the solution events are independent for each a_i, this joint probability can be evaluated as the product of the individual survival functions $S_k(s_k t)$

$$S_{\mathcal{A},\mathbf{s}}(t) = \prod_{k=1}^{K} S_k(s_k t), \tag{12}$$

or, in CDF form:

$$F_{\mathcal{A},\mathbf{s}}(t) = 1 - \prod_{k=1}^{K} [1 - F_k(s_k t)]. \tag{13}$$

[4] Here and in the following we assume an "ideal" machine, with no task switching overhead.

Given (5), (12) has an elegant representation in terms of the cumulative hazard function[5]

$$H_{\mathcal{A},\mathbf{s}}(t) = -\ln(S_{\mathcal{A},\mathbf{s}}(t)) = \sum_{i=1}^{K} -\ln(S_k(s_k t)) = \sum_{i=1}^{K} H_k(s_k t). \tag{14}$$

Algorithm selection can be represented in this framework by setting a single s_k value to 1, while a uniform algorithm portfolio would have $\mathbf{s} = \mathbf{s}_{\mathcal{U}} = (1/K, ..., 1/K)$. If the distributions F_k are available, other alternatives can be implemented. One naive approach could consist in evaluating, for each a_k, the probability that it will be the fastest, and using this value as the corresponding $s_k = \Pr\{T_k < T_{j\neq k}\}$. This would only have a good performance if there is one algorithm in the set that greatly dominates the others. Otherwise, this method would share resources among similarly performing algorithms, resulting in a poor performance. In [17, 21], we mapped runtime predictions to $\mathbf{s}$ values based on an heuristic "ranking" approach, in which the rth expected fastest solver would get a share 2^{-r}. Here we propose three different analytic approaches, based on function optimisation.

1. *Expected time.* The expected runtime value $E_{\mathcal{A},\mathbf{s}}(t) = \int_0^\infty t f_{\mathcal{A},\mathbf{s}}(t)dt$ can be obtained, and minimized with respect to $\mathbf{s}$:

$$\mathbf{s} = \arg\min_{\mathbf{s}} E_{\mathcal{A},\mathbf{s}}(t). \tag{15}$$

2. *Contract.* If an upper bound, or *contract*, t_u on execution time is imposed, one can instead use (13) to pick the $\mathbf{s}$ that maximizes the probability of solution within the contract $F_{\mathcal{A},\mathbf{s}}(t_u) = \Pr\{T_{\mathcal{A},\mathbf{s}} \leq t_u\}$ (or, equivalently, maximizes $H_{\mathcal{A},\mathbf{s}}(t_u)$, or minimizes $S_{\mathcal{A},\mathbf{s}}(t_u)$):

$$\mathbf{s} = \arg\min_{\mathbf{s}} S_{\mathcal{A},\mathbf{s}}(t_u). \tag{16}$$

3. *Quantile.* In other applications, one could want to solve the problem with probability at least α, and minimize execution time. In this case, a quantile $t_{\mathcal{A},\mathbf{s}}(\alpha) = F^{-1}_{\mathcal{A},\mathbf{s}}(\alpha)$ should be minimized:

$$\mathbf{s} = \arg\min_{\mathbf{s}} F^{-1}_{\mathcal{A},\mathbf{s}}(\alpha). \tag{17}$$

If the F_k are parametric, a gradient of the above quantities could be computed analytically, depending on the particular parametric form: otherwise, the optimisation can be performed numerically. Note that the shares $\mathbf{s}$ resulting from these three optimisation processes could differ: in the last two cases, they could also depend on the chosen values for t_u and α respectively. In no case is there a guarantee of unimodality, and it might be advisable to repeat the optimisation process multiple times, with different random initial values for $\mathbf{s}$, in case of extreme multimodality.

[5] Apart form the terms s_k, (14) is the method used by engineers to evaluate the failure distribution of a *series* system, which stops working as soon as one of the components fail, based on the failure distribution for each single component.

A choice among the three alternatives, as well as the choice of the relative parameters, might be imposed by the particular application, or left open as a design decision. We will postpone its discussion, and conclude this section remarking that the methods described here all rely on the assumption of independence of the runtime values among the different algorithms, which allows to express the joint probability (12) as a product. This assumption is met only if the F_k represent the runtime distributions of the a_k on the particular problem *instance* being solved. If instead the only F_k available capture the behavior of the algorithms on a *set* of instances, which includes the current one, independence cannot be assumed: in this case, the methods presented should be viewed as approximations. In a less pessimistic scenario, one could have access to models $\mathcal{M}$ of the F_k conditioned on features, or *covariates*, $\mathbf{x}$ of the current problem. In such a case the *conditional* independence of the runtime values would be sufficient, and the resulting joint survival probability could still be evaluated as a product

$$S_{\mathcal{A},\mathbf{s}}(t|\mathbf{x}) = \prod_{i=1}^{K} S_k(s_k t|\mathbf{x}). \tag{18}$$

In practice, such a model is usually not available, and has to be estimated. The degree of approximation implied by assumption (18) will depend on the fit of the model.

5 A continually learning dynamic portfolio

Let us now focus on the second of the issues mentioned in the introduction, namely, the difficulty of *static* runtime predictions. It is intuitive that re-evaluating $\mathbf{s}$ periodically could improve the performance, especially if the runtime values are spread on a large range. To be effective, this evaluation has to be based on a model $\mathcal{M}$ of the RTD conditioned also on the current state $\mathbf{x}_k$ of each algorithm: in the simplest setting, one can always consider the time spent y_k as the current state information, updating each F_k as in (9).

A dynamic algorithm portfolio (Algorithm 1) can be implemented by re-evaluating $\mathbf{s}$ periodically, each time based on F_k conditioned on the current state information, and time already spent. Any of the three methods presented in Section 4 could be used as a *time allocator* TA to update $\mathbf{s}$. An additional design decision would be required to set the sequence of time intervals Δt. Note also that in (Alg. 1) it is assumed that, for each incoming problem instance, there is at least one a_k that can solve it.

Algorithm 1 Dynamic algorithm portfolio

```
Algorithm set A = {a_1, ..., a_K}
Model M
while problem not solved do
  update F_k(t_k) := M(t_k|x_k, y_k) for k = 1, ..., K
  update Δt
  update s := TA({F_k})
  run A with share s for a maximum time Δt
end while
```

The conditional model $\mathcal{M}$ is usually not available, and would have to be estimated from experimental data. A straightforward application of the machine learning paradigm would require solving, with each algorithm, the same sequence of "training" problem instances, in order to collect a sufficient amount of runtime data. This approach would share the third issue mentioned in the introduction with other algorithm selection techniques: a huge amount of time would be spent solving the same training problems over and over again, in order to gather a sufficiently large amount of data.

A first idea for reducing training time is inspired by censored sampling techniques. As the engineers do with the light bulbs, we could run our portfolio with a uniform share $\mathbf{s}_{\mathcal{U}} = (1/K, 1/K, ..., 1/K)$ on each training problem instance, and instead of waiting for all the algorithms to end, we could stop after the first few solve the problem, and switch to the next. As said in Section 3, this would have an impact on the accuracy of the model, but the uniform share would at least assure that the fastest algorithm(s) would not be censored. In this way the model would be less accurate for less efficient algorithm/problem combinations. The downside of the uniform share, is that it would still have a huge overhead on performance.[6]

Another speed-up could be obtained using a partially trained model to guide further training. There might be good algorithm/problem combinations that are easy to learn, and bad ones that are easy to avoid. Instead of keeping a uniform $\mathbf{s}_{\mathcal{U}}$ throughout the training sequence, we could periodically train the model $\mathcal{M}$ during the sequence, and run our static or dynamic portfolio of choice on the remaining training problems "mixing" the output of the chosen time allocator $\mathbf{s}_{\mathcal{M}} = \mathrm{TA}(\mathcal{M})$, with the uniform $\mathbf{s}_{\mathcal{U}}$, as $\mathbf{s} = p_{\mathcal{M}}\mathbf{s}_{\mathcal{M}} + (1 - p_{\mathcal{M}})\mathbf{s}_{\mathcal{U}}$; the mixing coefficient $p_{\mathcal{M}} \in [0, 1]$ could be increased each time the model is updated. This would be more dangerous, as we would loose the positive effect of $\mathbf{s}_{\mathcal{U}}$, and risk of censoring the fastest algorithm. It is intuitive that, if $p_{\mathcal{M}}$ is increased too quickly, and the initial portion of the training sequence is somehow deceptive, an initially imprecise model could cause more time to be allocated to less efficient algorithms, and the execution of the fastest algorithms to be censored, thus reinforcing its own mistakes.

We are facing a trade-off between *exploration* of the performance of the various a_k, and *exploitation* of the model obtained so far. In [17], we addressed this trade-off heuristically, updating the model after each task solution, and gradually shifting through the problem sequence, from a uniform initial share to a model-based share, again heuristically evaluated. In the following section, we will treat this trade-off in the context of bandit problems with expert advice.

6 Time allocation as a bandit problem

In its most basic form [64], the *multi-armed bandit* problem is faced by a gambler, playing a sequence of trials against a K-armed slot machine. At each trial, the gambler chooses one of the available arms, whose rewards are randomly generated from different stationary distributions. The gambler can then receive the corresponding

[6] If we wait for just one algorithm to terminate, and t_I is the performance of the fastest, the resulting training cost will be Kt_I: another uncensored sample t_{II} would cost an additional $(K - 1)(t_{II} - t_I)$, and so on.

reward r_k, and, in the *full information* game, observe the rewards that he would have gained pulling any of the other arms. The aim of the game is to minimize the regret R, defined as the difference between the cumulative reward of the best arm, and the one earned by the gambler G

$$R = \max_k \sum_j x_k(j) - G. \tag{19}$$

A bandit problem solver (BPS) can be described as a mapping from the history of the observed rewards $r_k \in [0, 1]$ for each arm k, to a probability distribution $\mathbf{p} = (p_1, ..., p_K)$, from which the choice for the successive trial will be picked.

In recent works, the original restricting assumptions have been progressively relaxed, allowing for non-stationary reward distributions, *partial* information (only the reward for the pulled arm is observed), and *adversarial* bandits, that can set their rewards in order to deceive the player. In [1], no statistical assumptions are made about the process generating the rewards, which are allowed to be an arbitrary function of the entire history of the game (*non-oblivious* adversarial setting). Based on these pessimistic hypotheses, the authors describe probabilistic gambling strategies for the full and the partial information games, proving interesting bounds on the expected value of the regret.

Assuming that all a_k can solve all problem instances, it is straightforward to describe static algorithm selection in a K-armed bandit setting, where "pick arm k" means "run algorithm a_k on next problem instance." The reward for this game could be set based on the runtime of the chosen algorithm, for example as $r_k := 1/t_k$; alternatively, runtime t_k could represent a *loss*, to be minimized. The information would be partial: the runtime for other algorithms would not be available. The rewards would be generated by a rather complex mechanism, i.e., the algorithms a_k themselves, so the bandit problem would fall into the adversarial setting. As BPS typically minimize the regret with respect to a single arm, this approach would only allow to implement *per set* selection, of the overall best algorithm.

To avoid excessively long t_k, machine time could be subdivided into arbitrarily small intervals δt: "pick arm k" would mean "resume algorithm a_k on current problem instance, for a time δt, then pause it." Reward could be attributed $r_k := 1/t_k$ as before, t_k being the *total* runtime of the winning algorithm. Information would again be partial: more precisely, in this case it would be *censored*, as a lower bound on performance, and a corresponding upper bound on reward, would be available for the other algorithms. The bandit would be a non-oblivious adversary, as the result of each arm pull would depend on previous pulls of the same arm.

On a large number of arm pulls, the expected value of time spent executing a_k would be proportional to p_k. And, typically, bounds on regret for a BPS are proved based on expected values. The game described above is then equivalent to a static portfolio, using the $\mathbf{p}$ of the BPS as the share value $\mathbf{s}$, and updating it after a problem instance is solved. Again, the resulting selection technique is *static*, *per set*,[7] only profitable if one of the algorithms dominates the others on all problem instances.

[7] *Oblivious* per instance techniques could be based on different reward attributions, as in [10].

A less restrictive, and more interesting hypothesis, is that there is one of a set of *time allocators*, whose performance dominates the others. At this higher level, one could use a BPS to select among different static *time allocators*, $\text{TA}^{(1)}$, $\text{TA}^{(2)}$,..., working on a same algorithm set $\mathcal{A}$. In this case, "pick arm n" would mean "use time allocator $\text{TA}^{(n)}$ on $\mathcal{A}$ to solve next problem instance." In the long term, the BPS would allow to select, on a *per set* basis, the $\text{TA}^{(n)}$ that is best at allocating time to algorithms in $\mathcal{A}$ on a *per instance* basis. If the BPS allows for time-varying reward distributions, it could also deal with time allocators that are *learning* to allocate time.

A more refined alternative is suggested by the bandit problem with expert advice, as described in [1, 2]. Two games are going on on parallel: at a lower level, a partial information game is played, based on the probability distribution obtained *mixing* the advice of different *experts*, represented as probability distributions on the K available arms. The experts can be arbitrary functions, and give a different advice for each trial. At a higher level, a *full information* game is played, with the N experts playing the roles of the different arms. The probability distribution $\mathbf{p}$ at this level is not used to pick a single expert, but to mix their advices, in order to generate the distribution for the lower level game. In [1], Auer et al. propose an algorithm called Exp4 (Alg. 2) to play this two-level game. Exp4 is a combination of the algorithms for the full and the partial information setting. It features a fixed lower bound γ on the exploration probability, which can be set, based on the total number of trials M, in order to obtain a bound on the *expected* regret relative to the performance of the best *expert*:

$$E(R) \leq 2.63\sqrt{MK \ln N}. \tag{20}$$

Algorithm 2 Exp4(K, N, M) by Auer et al. [1]

K arms, N experts, M trials
set $\gamma := \min\left\{1, \sqrt{\frac{K \ln N}{(e-1)M}}\right\}$
initialize $w_n := 1$ for $n = 1, ..., K$;
for each trial **do**
  get advice vectors $\mathbf{s}^{(n)} \in [0, 1]^K$ from experts $n = 1, ..., N$
  set $p_n := w_n / \sum_{i=1}^{N} w_i$ for $n = 1, ..., N$
  pick arm k with probability $s_k := (1 - \gamma) \sum_{n=1}^{N} p_n s_k^{(n)} + \gamma / K$
  observe reward $r_k \in [0, 1]$
  set $\hat{r}_k := r_k / p_k$
  update $w_n := w_n \exp(\gamma s_k^{(n)} \hat{r}_k / K)$ for $n = 1, ..., N$
end for

The original formulation is based on a finite upper bound on the cumulative reward of the best expert, which is at most M if each reward is in $[0, 1]$. A variant of the algorithm is proposed if M is unknown, or if the rewards are much smaller than 1. Bound (20) requires that the *uniform* expert $\mathbf{s} = (1/K, ..., 1/K)$ is included in the set.

In our case, the time allocators play the role of the experts, each suggesting a different $\mathbf{s}$, on a per instance basis; and the arms of the lower level game are the

K algorithms, to be run in parallel with the mixture share. The *partial* information on the reward at the lower level (based on the runtime of the a_k first to solution) is translated into *full* information at the upper level, based on the $\mathbf{s}^{(n)}$ proposed by each $\mathrm{TA}^{(n)}$.

Before prosecuting, we need to decide how to attribute the rewards. Ideally, we would like Exp4 to select the time allocator that is better at giving more time to the fastest algorithms. As we cannot know the real fastest algorithm, one good idea could be to reward minimization of solution time, setting $r_k \propto 1/t_k$. One possible side effect of this choice could be that, for problem sequences on which runtimes vary of different order of magnitudes, the rewards for the harder problems would be much lower than the ones for the easy ones. We will then adopt a logarithmic reward attribution, as in [20]. As Exp4 requires normalized rewards, we can set lower and upper bounds t_{min}, t_{max} on runtime, and set the reward for the winning algorithm a_k as

$$r_k = \frac{\ln t_{max} - \ln t_k}{\ln t_{max} - \ln t_{min}}. \tag{21}$$

This reward will be then distributed by Exp4 to the time allocators, based on how much time they allocated to a_k. The extension to *dynamic* time allocators (Alg. 1) is straightforward: in this case the $\mathbf{s}^{(n)}$ would depend, for each allocator, on the sequence of intervals $\Delta t(0)$, $\Delta t(1)$, ..., and the corresponding $\mathbf{s}$ proposed during each interval, and the normalized value of $\sum_j \mathbf{s}^{(n)}(j)\Delta t(j)$ would be used in place of $\mathbf{s}^{(n)}$ at line 10 of Alg. 2.

We can then use Exp4 to address the exploration-exploitation trade-off that we left open in the last section. We can solve each problem in the training sequence mixing the uniform $\mathbf{s}_{\mathcal{U}}$, and the $\mathbf{s}_{\mathcal{M}}$ evaluated by the model-based allocator, using the current output $\mathbf{p}$ of Exp4 as a mixing coefficient. In this way Exp4 would detect when the model is ready to use, and starts gaining a better performance than the uniform allocator. After each instance is solved, we can also update Exp4.

The regret rate (20) is particularly interesting, as it depends on the *logarithm* of the number of experts N. We can exploit this fact to take the design decision that we left open in Section 4, namely, which allocator function to use: we can leave this decision to Exp4, picking a redundant set of time allocators. We can also try different values for the respective parameters. Note that all these allocators can share a common model $\mathcal{M}$, so the *computational* overhead would depend on the cost of the time allocators alone. The resulting "gambling" time allocator (GambleTA) is described in Alg. 3.

Using a non-uniform share, there is no guarantee that the winner algorithm will be the actual fastest, so our reward scheme could be deceptive. The sequence of tasks can also be deceptive, and again cause the model to reinforce its own mistakes. All this is allowed in the pessimistic settings of Exp4, which will still guarantee that the expected regret, compared to the gain of the best time allocator, is bounded by (20).

This optimal regret is defined with respect to the best *allocator*. Nothing can be said about the performance w.r.t. the best *algorithm*. In a worst-case setting, if none of the time allocator learns anything, Exp4 will give most credit to the uniform share, which gains a reward $\hat{r}_k/K$ at every trial. We will now see two example applications on which the performance of GambleTA is quite far from this pessimistic scenario.

Algorithm 3 GAMBLETA Gambling time allocator

Algorithm set $\mathcal{A}$ with K algorithms
N time allocators, including $\mathbf{s}_{\mathcal{U}} = (1/K, ..., 1/K)$
M problem instances
initialize EXP4(K, N, M)
let EXP4 initialize $\mathbf{p} \in [0, 1]^N$
initialized model $\mathcal{M}$
for each problem $b_1, b_2, ..., b_M$ **do**
  while b_m not solved **do**
    update Δt
    for each time allocator $\text{TA}^{(1)}, ..., \text{TA}^{(N)}$ **do**
      update $\mathbf{s}^{(n)} = \text{TA}^{(n)}(\mathcal{M})$, $\mathbf{s}^{(n)} \in [0, 1]^K$
    end for
    evaluate mix $\mathbf{s} = \sum_{n=1}^{N} p_n \mathbf{s}^{(n)}$
    run $\mathcal{A}$ with share $\mathbf{s}$, for a maximum time Δt
  end while
  observe reward r_k for winner a_k
  update EXP4
  let EXP4 update $\mathbf{p}$
  update $\mathcal{M}$ based on collected runtime data
end for

7 Experiments

We present two experiments, both with very small algorithm sets ($K = 2$), but long, and challenging, problem sequences. The first experiment features a complete and a local search SAT solver, dealing with a mixed set of CNF3 SAT instances at the sat-unsat threshold. The second experiment features solvers for a published Auction Winner Determination Problem (WDP) benchmark [45].

Before proceeding, we will describe the remaining details of our time allocation algorithm. As said, we use EXP4 [1] at the top level, to mix the share decisions $\mathbf{s}^{(n)}$ of different time allocators $\text{TA}^{(n)}$ (Alg. 3). No care was put in selecting the set of time allocators, as EXP4 is better at this game. The set included (see Section 4 for a description):

- The uniform time allocator, with share $\mathbf{s} = (1/K, ..., 1/K)$, required by EXP4.
- A set of nine quantile minimizers , $\mathbf{s} = \arg\min_{\mathbf{s}} F^{-1}_{\mathcal{A},\mathbf{s}}(\alpha)$, with equally spaced values for the parameter α (0.1, 0.2, ..., 0.9).
- A "greedy" contract allocator, using the next time limit as a contract: $\mathbf{s} = \arg\min_{\mathbf{s}} S_{\mathcal{A},\mathbf{s}}(\Delta t + \sum_{k=1}^{K} y_k)$, y_k being the time spent so far by a_k.

Each experiment was repeated using each one of the allocators, always accompanied by the uniform, but none of them could improve on the performance of the ensemble. EXP4 preferred different time allocators on the two benchmarks, but always discarded the quantile allocators with $\alpha \geq 0.5$.

The sequence of time intervals Δt employed by the dynamic portfolio was exponential, with base two. (Δt_0,$2\Delta t_0$,$4\Delta t_0$,...). We set the initial Δt_0 to two different

values for the two experiments. Also t_{min} was different for the two benchmarks, while t_{max} was kept fixed at 10^{10}.

As a model, we used the conditional non-parametric hazard estimator ($\hat{h}(t|x)$) by Wichert and Wilke (WW in the following) [77]. This model is conceptually simple, and computationally efficient. As most non-parametric methods, it stores all the training data (x_i, t_i): the time values t_i of censored and uncensored events, and the covariates x_i, evaluating an empirical CDF (7) $F_x(x)$ of the covariate value x. In order to predict the hazard function for an unseen value x of the covariate, it first estimates its CDF value $F_x(x)$, by simply evaluating its rank in the sorted list of covariates. The probability $F_x(x)$ is then compared to the $F_x(x_i)$ of each sample (again obtained from the rank), through a kernel function K, with bandwidth parameter b_n, and the value of $K((F_x(x) - F_x(x_i))/b_n)$ is used to weight the event t_i. The weight values are used in place of "1" in (8), to evaluate a Kaplan–Meier estimate of the hazard for the covariate x:

$$\hat{h}(t|x) = \frac{\sum_{t_i=t, v_i=1} K\left(\frac{F(x)-F(x_i)}{b_n}\right)}{\sum_{t_i \geq t} K\left(\frac{F(x)-F(x_i)}{b_n}\right)}. \tag{22}$$

If the covariates are multidimensional, the process is repeated for each dimension, and the products of the resulting kernel distances are used as weights. In short, (22) performs a nearest neighbor estimate of the hazard: the kernel distance is measured on the *distribution* of covariate values, and is not sensitive to scaling. The kernel function K is required to be symmetric around 0, and integrate to 1. We used a uniform kernel (0.5 on $[-1, 1]$, and 0 elsewhere), which is a common choice in non-parametric statistics. The convergence proof for the estimator requires the bandwidth parameter b_n to be set based on the size n of the stored sample, as $b_n \in [n^{-1/2}, n^{-1/4}]$. We present results for $b_n = n^{-1/4}$, which provides the widest allowed kernel.

A separate model was learned for each algorithm, using a small set of problem specific features as covariates.[8] The only dynamic feature taken into account was the time spent y_k, as in (9, 10), which, in the non-parametric setting, simply consists in discarding hazard values h_j with $t_j \leq y_k$. The RTD of the portfolio was evaluated based on the cumulative hazard[9] form (14). The time allocators described in Section 4

[8] As the two algorithms are in both cases not related. For different parametrizations of the same algorithm, a single model can be used, conditioned also on parameter values.

[9] The model (WW) outputs, for a given covariate x, two vectors, one of event times $\{t_i\}$, one of the corresponding hazard estimates $\{h_i\}$. Based on this data, a vector of hazard values for the algorithm running with share s_k has first to be evaluated. Note that the derivative of $H(s_k t)$ would be $s_k h(s_k t)$, but in the nonparametric setting the h_i are pulses, not point values: scaling them by s_k would not be correct. To see why, consider that the cumulative hazard at $H(\infty)$ should not vary by scaling time, so the integral across the scaled time values must remain the same. Only time has to be dilated, dividing the time values t_i by the s_k chosen by GambleTA. Hazard values relative to different algorithms are then merged, sorting the resulting list according to time values. The cumulative hazard (14) can finally be evaluated, as the cumulative sum of the resulting hazard values. This value is used by two different functions, evaluating the quantile (3) and survival probability at the next contract (2), based on the survival function obtained from (5). These last two functions are passed as arguments to the Matlab function `fminbnd`, to be minimized.

were evaluated numerically, using a line search routine (see note 9), careless of multimodality: on runs that were monitored, we observed multimodality only for high levels of the parameters α or t_u, for which the performance of the time allocators was poor anyway.

We repeated both experiments 50 times, each time with a different random reordering of the problem instances, and a different random seed for the algorithms, if randomized. Unless otherwise stated, all results reported are 95% confidence bounds, evaluated on 50 runs. For both experiments, the parallel execution of the algorithms was simulated, using stored runtime data;[10] the time values reported only include the algorithm runtimes.[11]

We assess the performance of GambleTA by comparing it with the uniform time allocator $\mathbf{s}_{\mathcal{U}} = (1/K, ..., 1/K)$ alone; and the one of an *oracle*, with foresight of the runtime values, which only executes, for each problem instance, the algorithm that will be fastest. If $t_G(j)$ is the runtime of our time allocator on problem instance j, $t_k(j)$ is the runtime of algorithm a_k, then $t_O(j) = \min_k\{t_k(j)\}$ is the runtime of the oracle, and $t_{\mathcal{U}} = Kt_O$ is the runtime of the uniform share. We will describe the performance of the allocator until task m reporting the *cumulative time* $\sum_{j=1}^{m} t_G(j)$, and the *cumulative overhead*

$$\frac{\sum_{j=1}^{m} t_G(j) - t_O(j)}{\sum_{j=1}^{m} t_O(j)}, \tag{23}$$

relative to the performance of the oracle. These are fair performance indicators, also for a per instance selection technique, but do not capture the performance on a single instance. Plotting this information averaged on multiple runs is problematic, as the order of the instances is different for every run, and in both benchmarks the runtimes may differ of several orders of magnitude. We will then plot the performance on *each* instance, and for *each* run, against the runtime of the oracle, and the uniform share.

[10] Unfortunately, doing research on an online method does not have the benefits of just using one, as comparing with the performance of an oracle requires the knowledge of all runtimes, which means that, for the first experiments, we also had to solve all satisfiable problems with Satz–Rand.

[11] Including the overhead of the quantile evaluations, the model update, etc., would not be fair, as all these operations are implemented in unoptimized, and rather bloated, Matlab code, while the a_k are written in C. WW, as other nonparametric methods, has a very cheap learning phase, which consists in sorting independently the event times and the d dimensions of the covariates $\mathbf{x} \in \mathbb{R}^d$. The cost of prediction is d searches on the sorted covariate data, and the cost of (22). Quantiles can also be evaluated just by searching a value on a sorted list. To give a rough idea, we report the profile of a single run on the SAT-UNSAT benchmark: on 1, 899 problems, two WW models were updated once per problem, for a total of 4.6 s. The hazard generating function (22) was called about 280, 000 times in total, as each of the allocators uses it in the optimisation process (see note 9) : the cost was 88 s. An additional 3 min was spent in merging and re-sorting hazard vectors, to evaluate the hazard of the portfolio. The total runtime of the portfolio alone on the problem sequence would have been about 24 min. These figures would obviously change passing to a C implementation. Simple optimizations, like preserving order when merging two hazard vectors, would further improve the situation. The fact that the data is sorted would allow for more advanced optimizations, based for example on balanced trees, with a cost $O(\log n)$, n being the number of samples, both for search and insertion. Regarding memory requirements, the model would collect K samples for each solved task. On a modern machine, this amount of data would not cause any problem, even with long task sequences, but once there is enough data one can start to reduce the number of stored samples, for example merging neighboring hazards.

To underline the improvement during the problem sequence, we will also report separate statistics for the first and second halves of the two problem sequences.

7.1 Satisfiability problems

Satisfiability (SAT) problems [22] constitute a standard benchmark in AI. A conjunctive normal form CNF(k,n,m) problem consists in finding an instantiation of a set of n Boolean variables that simultaneously satisfies a set of m clauses, each being the logical OR of k literals, chosen from the set of variables and their negations. A problem instance is termed *satisfiable* (SAT) is there exists at least one of such instantiations, otherwise it is *unsatisfiable* (UNSAT). An instance is considered solved if a single solution is found, or if unsatisfiability is proved. With $k = 3$ the problem is NP-complete. Satisfiability of an instance depends in probability on the clauses to variables ratio: a *phase transition* [52] can be observed at $m/n \approx 4.3$, at which an instance is satisfiable with probability 0.5. This probability quickly goes to 0 for m/n above the threshold, and to 1 below.

SAT solvers can be broadly classified in two categories: *complete* solvers, that execute a backtrack search on the tree of possible variable instantiations, and are guaranteed to determine the satisfiability of a problem in a finite, but possibly unfeasibly high, amount of time; and *local search* (LS) solvers, that cannot prove unsatisfiability, but are usually faster than complete solvers on satisfiable problems. In other words, a local search solver can only be applied to satisfiable instances: at the threshold, there is a 0.5 probability that the solver will run forever. The RTD of a complete solver will have $F(\infty) = 1$ with a finite 1 quantile, for any value of m/n; while a local search solver has a $F(\infty) = 0.5$ on instances at the 4.3 threshold. Users of LS interested in such benchmarks have then to first filter out unsatisfiable instances by running a complete solver, in order to test the local search algorithm on SAT instances only. This means that, at the phase transition, local search implies an additional cost, equal to the performance of a complete solver, which obviously does not make it competitive for such problem instances.

Our first experiment was performed using a portfolio of two SAT solvers from the two categories above. As a benchmark, we used the complete set of `uf-n-m` and `uuf-n-m` instances from SATLIB [30]. These are randomly generated instances at the phase transition, with n ranging from 20 (resp. 50 for the unsat) to 250, 100 instances for each size, and m varying accordingly. The instances are subdivided in groups of satisfiable (`uf*`) and unsatisfiable (`uuf*`) instances. We merged all groups in a single sequence, of 1, 899 problems[12] in total, that was randomly re-ordered for each run of the experiment.

As a complete solver we picked Satz–Rand [25], a version of Satz [47] in which random noise influences the choice of the branching variable. Satz is a modified version of the complete DPLL procedure, in which the choice of the variable on which to branch next follows an heuristic ordering, based on first and second level unit propagation. Satz–Rand differs in that, after the list is formed, the next variable to branch on is randomly picked among the top h fraction of the list. We present

[12] This odd number is due to the fact that instance `uuf-200-860` number 100 is missing in the online archive. Note also that the smallest n for the unsatisfiable instances is 50, so there are 1, 000 SAT and 899 UNSAT instances in total, making the SAT probability for the whole set slightly higher than 0.5.

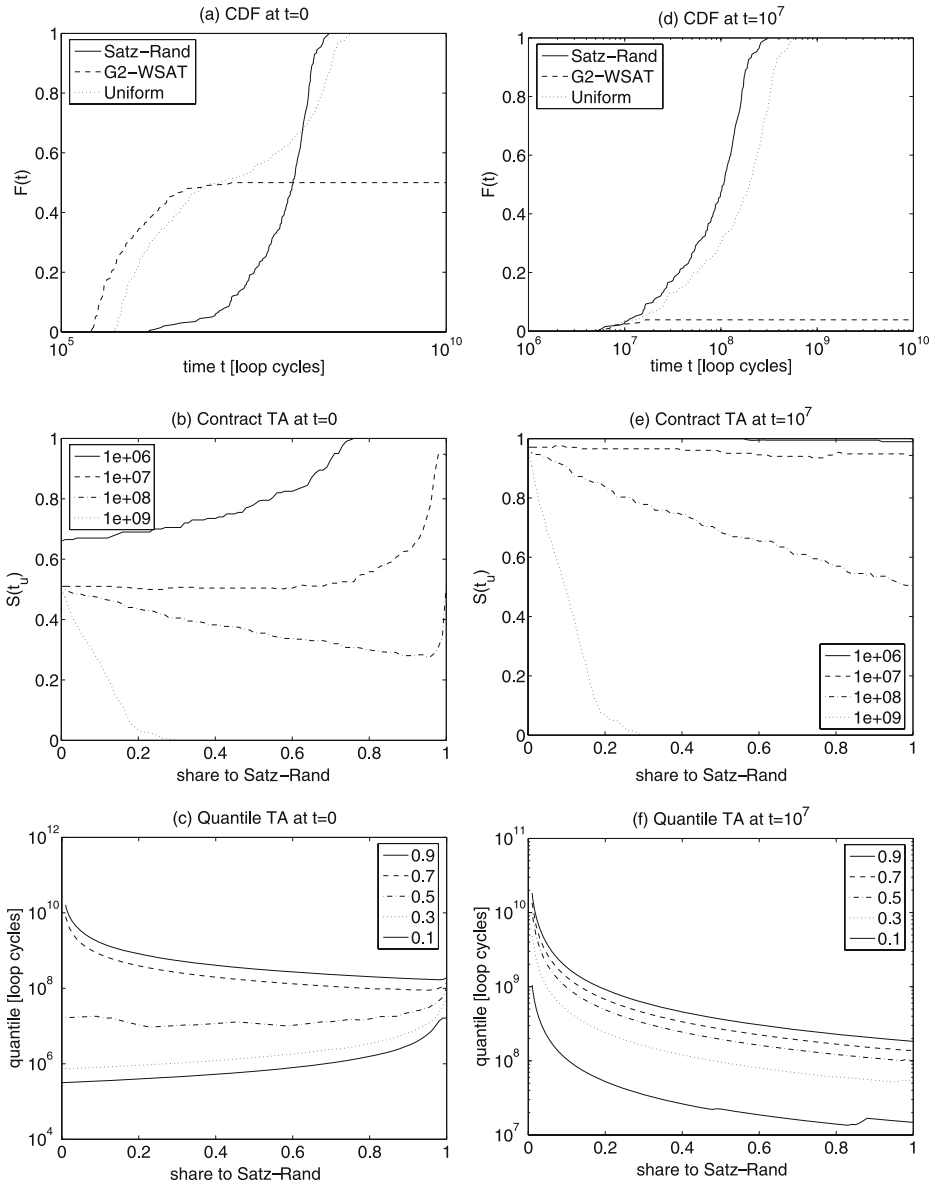

Fig. 1 SAT-UNSAT problems. These plots illustrate the functioning of the Contract (2) and Quantile (3) time allocators (Section 4), and are not generated from a run of GambleTA, but from a RTD estimate on problems of size $n = 250$ only. Left column: situation at $t = 0$. Right column: after $t = 10^7$ of uniform parallel run (5×10^6 for each algorithm). *Top*: RTD of the single algorithms, and of the uniform share. *Middle*: survival probability $S_{A,\mathbf{s}}(t_u)$ (*vertical axis*) at a time contract t_u, for different values of the share s_1 assigned to Satz–Rand (*horizontal axis*), and different values of the time contract t_u (*different lines*). *Bottom*: quantiles of runtime for different values of α (*different lines*), and different values of time share allocated to Satz–Rand (*horizontal axis*). The minimum of each line in (b, c, e, f) is the share allocation decided by the corresponding TA

results with the heuristic starting from the most constrained variables, as suggested also in [47], noise parameter set to 0.4, and the restart mechanism disabled, as the RTD of the algorithm does not display heavy-tailed behavior [25] for this n/m ratio. As a local search solver we used G2-WSAT [48]: for this algorithm, we set a high noise parameter (0.5), as advisable for problems at the phase threshold, and the diversification probability at the default 0.05. As both solvers are randomized, we also used a different random seed for each run.

As we needed a common measure of time, and the CPU runtime measures are quite inaccurate (see also [29], p. 169), we modified the original code of the two algorithms adding a counter, that is incremented at every loop in the code. The resulting time measure was consistent with the number of backtracks, for Satz–Rand, and the number of flips, for G2-WSAT. All runtimes reported for this benchmark are expressed in these loop cycles: on a 2.4 GHz machine, 10^9 cycles take about 1 min.

The only feature used for the model WW was n, the number of variables in the SAT problem, as the clauses-to-variable ratio m/n is practically constant. Δt_0 and t_{min} where both set to 10^4, the order of magnitude of the initialization cost of both algorithms on the smallest problem size.

This algorithm set/problem set combination is quite interesting. G2-WSAT almost always dominates the performance of Satz–Rand on satisfiable instances, while the latter is obviously the winner on all unsatisfiable ones, on which the runtime of G2-WSAT is infinite.

This situation is visualized in Fig. 1a, which plots the empirical CDF of the runtimes for the two solvers, resulting from an estimate for a single random seed, on the two sets of larger instances (`uf-250`, `uuf-250`). One can clearly notice the advantage of G2-WSAT on satisfiable instances, represented by the small lower quantiles (below 10^6). From quantile 0.5 on, the RTD remains flat, reflecting the fact that half of the instances are unsatisfiable. Satz–Rand starts solving problems later, and is competitive with G2-WSAT only on a small number of satisfiable instances, but is able to solve also all the unsatisfiable ones, as indicated by the fact that the RTD reaches 1, i.e., the quantile t_1 is finite. The third line in the plot, labeled "uniform," represents the RTD of the uniform portfolio $\mathbf{s}_{\mathcal{U}} = (0.5, 0.5)$.

Algorithm selection would be easy in this case, if not for the fact that the satisfiability of an instance *cannot* be predicted in any way, before attempting solution. As G2-WSAT is incomplete, any sensible single algorithm selection technique would select Satz–Rand on all problems. The performance of this algorithm alone is better than the one of the uniform share, but obviously worse than the performance of the oracle, as this latter can profit from its foresight, and solve SAT instances with G2-WSAT.

Figure 2a displays the evolution of the cumulative time during the task sequence, comparing for each task i the cumulative performance of GambleTA $\sum_{j<i} t_G(j)$ to the cumulative performance of the oracle $\sum_{j<i} t_O(j)$, and of the uniform share $\mathbf{s}_{\mathcal{U}}$ $(K \sum_{j<i} t_O(j))$. The performance of Satz–Rand is also plotted, as this algorithm can solve all the problems. Lines represent *upper* confidence bounds, evaluated on 50 runs.

Figure 2b plots the cumulative overhead (23) of GambleTA, during the problem sequence. Here the dotted lines represent upper and lower 95% confidence bounds. GambleTA is quite quick in converging to the final performance, and then seems to oscillate; averaged on 50 runs, it ends the problem sequence with a cumulative

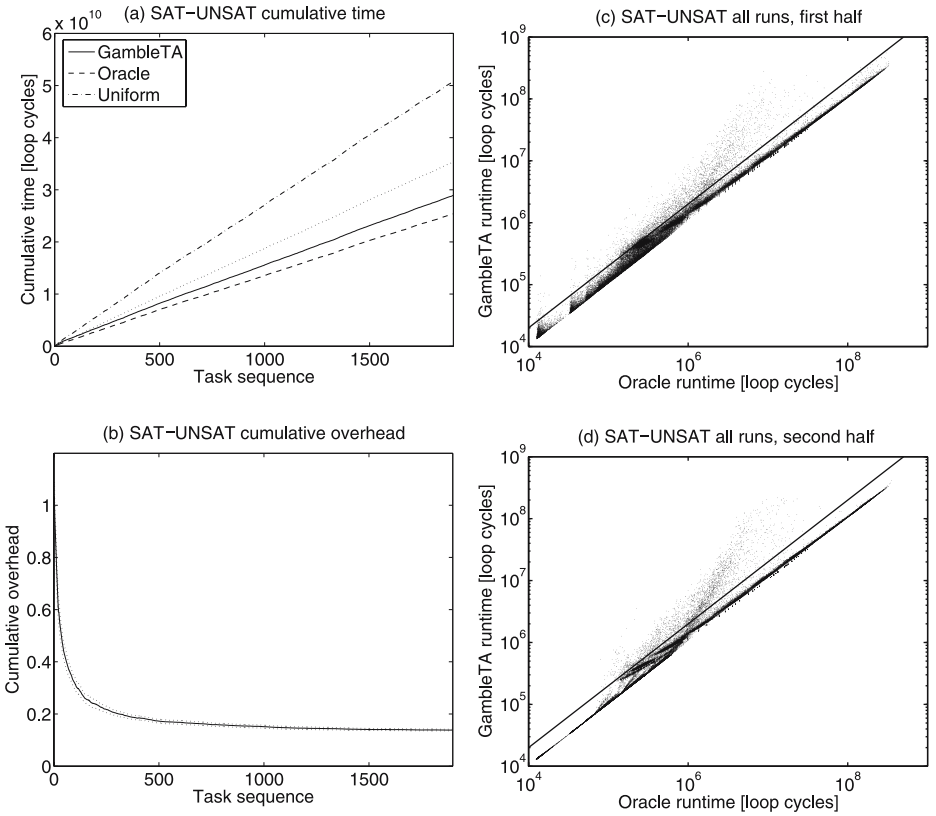

Fig. 2 SAT-UNSAT problems. **a** Cumulative time on the SAT-UNSAT problem set. Upper 95% confidence bounds on 50 runs, with random reordering of the problems. GambleTA is our time allocator. Oracle is the lower bound on performance. Uniform is the (0.5,0.5) share. Satz–Rand is the single algorithm. **b** Cumulative overhead (23) on the SAT-UNSAT problem set. Upper and lower confidence limits. *Right column*: Performance of GambleTA compared to the oracle, on *all* problems and for all runs. $50 \times 1,899/2 = 47,475$ points per plot. **c** First half of the sequence. **d** Second half. The diagonal (*not marked*) is the performance of Oracle. The *continuous line* above the diagonal is the performance of Uniform. Note that this line is crossed by many runs, especially for runtimes around 10^6. The biggest improvements in the second part of the sequence can be seen on very easy and very hard problems. See also Fig. 3, and Table 1

overhead of about 14%. Note that this figure includes the performance at the beginning of the sequence, when the model is still poorly trained.

Examining a single run, it can be observed that most of the allocators quickly learn to start solving each problem using the local search algorithm, and later switch to Satz–Rand if no solution is found by G2-WSAT.

As there are only two algorithms in the set, we can easily visualize the time allocators (see Section 4). Using the same data from Fig. 1a, in Fig. 1b, we plot the survival probability $S_{A,\mathbf{s}}(t_u)$ (vertical axis) at a time contract t_u, for different values of the share s_1 assigned to Satz–Rand (horizontal axis), and different values of the time contract t_u (different lines). Figure 1c displays an analogous plot for the quantile

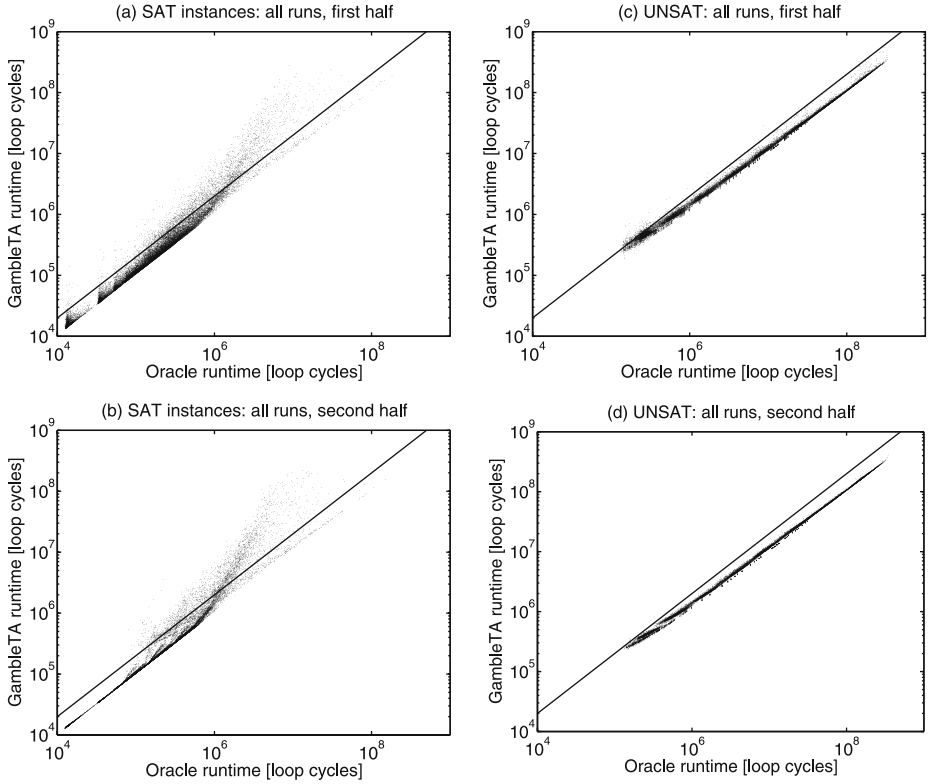

Fig. 3 SAT-UNSAT problems. Performance of GambleTA compared to the oracle, on *all* problems and for all runs, separated for SAT (*left column*: $50 \times 1,000/2 = 25,000$ points per plot) and UNSAT (*right column*: $50 \times 899/2 = 22,475$ points per plot) problems. *Top*: first half of the sequence. *Bottom*: second half. Note that this distinction is unavailable to the algorithm: the data was filtered a posteriori from the data of Fig. 2c,d, and refers to the same experiment, with SAT and UNSAT instances randomly mixed. The order of problem instances is different for every run, so the same instance might be met at different stages of the learning process. The diagonal (not marked) is the performance of Oracle. The continuous line above the diagonal is the performance of Uniform. Note that this line is crossed by many runs, especially for SAT instances, for runtimes above 10^6

minimization method: this time the ordinates report the logarithm of the quantile $t_{A,\mathbf{s}}(\alpha)$ for the portfolio, and different lines correspond to different values of the required solution probability α.

You can notice that the optimum of $\mathbf{s}$ varies according to the parameter of the time allocator (see Section 4): for low values of the contract t_u, and the quantile α, the optimum is at $s_1 = 0$, which means that only G2-WSAT is run, notwithstanding the 0.5 survival probability at ∞.

If both algorithms are run in parallel, for 10^7 loops in total, without solving the problem, we get the situation depicted in the right column of Fig. 1. The RTD of the two algorithms have been shifted and scaled, as in (9), and the one of G2-WSAT has almost disappeared. Given the time already spent, there is only a very small probability that G2-WSAT will solve the problem. This situation is reflected in the plots of the contract (Fig. 1e) and quantile (Fig. 1f) allocators: now the optimum of

Table 1 Various performance indicators for GambleTA, evaluated over the first and second halves of each problem sequence, averaging over 50 runs

		First half	Second half
SU	GTA	$1.46 \times 10^{10} \pm 2.14 \times 10^{8}$	$1.42 \times 10^{10} \pm 2.11 \times 10^{8}$
	OR	$1.27 \times 10^{10} \pm 2.06 \times 10^{8}$	$1.26 \times 10^{10} \pm 1.92 \times 10^{8}$
	OVH	0.153 ± 0.0058	0.124 ± 0.0034
	WTU	0.0739 ± 0.00296	0.0576 ± 0.00269
S	GTA	$8.07 \times 10^{8} \pm 4.64 \times 10^{7}$	$8.47 \times 10^{8} \pm 4.58 \times 10^{7}$
	OR	$3.05 \times 10^{8} \pm 1.47 \times 10^{7}$	$2.99 \times 10^{8} \pm 1.32 \times 10^{7}$
	OVH	1.66 ± 0.127	1.86 ± 0.143
	WTU	0.119 ± 0.0044	0.109 ± 0.0050
U	GTA	$1.37 \times 10^{10} \pm 2.01 \times 10^{8}$	$1.35 \times 10^{10} \pm 2.15 \times 10^{8}$
	OR	$1.23 \times 10^{10} \pm 2.01 \times 10^{8}$	$1.24 \times 10^{10} \pm 1.88 \times 10^{8}$
	OVH	0.117 ± 0.00481	0.0822 ± 0.0029
	WTU	0.024 ± 0.0037	0.0003 ± 0.0002
WDP	GTA	$5.72 \times 10^{7} \pm 6.48 \times 10^{5}$	$5.51 \times 10^{7} \pm 6.57 \times 10^{5}$
	OR	$5.45 \times 10^{7} \pm 6.44 \times 10^{5}$	$5.37 \times 10^{7} \pm 6.44 \times 10^{5}$
	OVH	0.0502 ± 0.0022	0.026 ± 0.0016
	WTU	0.176 ± 0.0014	0.148 ± 0.0025

Ninety-five percent confidence intervals. SU: SAT-UNSAT benchmark. S: SAT instances, filtered from SU. U: UNSAT instances, filtered from SU. Note that these two do not refer to separate experiments, but are extracted from the results on the SAT-UNSAT problem sequence. WDP: Winner Determination Problem. Indicators: GTA: cumulative performance of GambleTA. OR: cumulative performance of the Oracle. OVH: cumulative overhead of GambleTA, with respect to the Oracle (23). WTU: fraction of problems on which GambleTA is worse than Uniform ($t_U = Kt_O$)

the lines is at $s_1 = 1$ for all values of the parameters, except the smallest, which means that most allocators would only run Satz–Rand.

During the course of a run, Exp4 gradually selects a mixture of three quantile allocators, with small values for α (0.2, 0.3, 0.4). Note that the predictions of the WW model, and thus the decisions of the time allocators, are solely based on previously observed runs. The view of the time allocators is similar to the one in Fig. 1: only, there 200 samples for each algorithm are available, for the same covariate ($n = 250$), and this results in a much smoother model than the one typically available during the initial part of the task sequence. The surfaces (in this case lines) optimised by the time allocators look smooth anyway, especially for low values of the parameters, but the contract allocator tends to look flat for large intervals of s_1 values.

The simple tactic found by GambleTA is not always effective, and can actually result in a performance much worse than the uniform share, on a single instance. We show this in Fig. 2c,d, where the runtimes of GambleTA are scatter-plotted against the one of the oracle, for *all* the 1, 899 instances, and *all* the 50 runs. The two plots only distinguish among instances met during the first half of the sequence, and the

second: all other order information is lost. Note that, as the order of instances is picked randomly for each run, a same instance can figure in both plots: but it would represent two different runs, with different random seeds for the a_k, and would likely map to different points. We did not plot the diagonal, which would be the performance of the oracle, as it would interfere with the data. The continuous line above the diagonal represents the performance of the uniform share $t_{\mathcal{U}} = Kt_O$. There are many points above this line, which indicates a performance worse than uniform (WTU). The biggest improvement, from the first half (Fig. 2c) to the second (Fig. 2d), seems to be on really easy and really hard instances, with low and high runtimes respectively.

In order to further analyze this situation, in Fig. 3 we repeat the same scatter-plots using the same data, but distinguishing among satisfiable and unsatisfiable instances. It is now clear that the cloud of poor performance still visible in Fig. 2d is entirely represented by satisfiable instances, on which the runtime of the fastest algorithm (probably G2-WSAT) is between 10^6 and 10^7 loops. We can now make an hypothesis: looking back at Fig. 1a,d, we see that this is the time range on which the runtime distributions of the two algorithms overlap (at least for $n = 250$ variables). In other words, the longest successful runs of G2-WSAT and the shortest ones of Satz–Rand are in this range. The surfaces of the time allocators will be similar to the ones in Fig. 1e,f.

In Table 1 we display a few performance statistics, separately for the two halves of the task sequence. GTA labels the cumulative time of GambleTA, OR the one of the oracle. OVH represents the cumulative overhead (23), evaluated only on the respective half. WTU stands for "worse than uniform." It measures the fraction of task instances on which the performance is worse than the uniform $t_{\mathcal{U}} = Kt_O$.

The first block in the table (SU) refers to the full set of instances, as solved by GambleTA. The second (S) and third (U) respectively refer to the satisfiable and unsatisfiable instances alone. We can see that, in terms of the number of instances, only on 11% of satisfiable instances a WTU performance is observed, but this is enough to give a very high overhead value: the overhead is actually slightly worse in the second half of the sequence. But we have to bear in mind that this situation results from a 6% of the total number of problems, on which the runtime of G2-WSAT is unusually long. GambleTA is willing to pay this price, in order to avoid running G2-WSAT for too long on a potentially unsatisfiable instance.

The performance is much better on the unsatisfiable instances, as they are characterized by much longer runtime values, and the overhead of trying G2-WSAT first is low. Here the WTU instances go down to less than one on a thousand, and the overhead at less than 9%. On the whole set, the performance for the second half is a 13% overhead, and less than 7% WTU. Due to the difficulty of the task, we do not expect more than a marginal improvement in the performance from the use of more sophisticated modeling techniques, or more features.

7.2 Winner determination problem

The Auction Winner Determination Problem (WDP) [45] is an interesting combinatorial optimisation problem, where a set of agents allocate money on n bids over m goods, and the winning subset of bids, that maximizes the sum of the amounts bidden, must be determined. The agents have limited amounts of money, and are allowed to

specify XOR constraints over the bidden goods, and the selected winning subset has also to satisfy these constraints. The problem is NP-hard.

In [45], to which we refer for more details and references, the hardness of randomly generated WDP instances is modeled, describing the performance of a Linear Programming software (CPLEX), and an ad-hoc solver (CASS). The runtime of these solvers is related to 28 instance features, including the size (n,m), and serves as an input for a regression routine aimed at learning a predictive model of runtime value, conditioned on instance features. The performance of the models is assessed using mean squared error on the logarithm of predicted values, which suggests a parametric assumption of the run-time distribution being log-normal. Censored runtimes ("capped" runs in the terminology of the paper) are treated as the uncensored, and it is argued that the impact of this approximation on model precision is low. The resulting models are actually quite precise in terms of the proposed error measure. The performance of CPLEX dominates CASS, but on about 1/4 of the instances this situation is inverted. In such a case, a per set selection technique would always select CPLEX. As an interesting example application of these models, the authors propose a per instance algorithm selection technique, in which the expected fastest algorithm is picked based on the model's predictions. In the original paper, the model is trained on runtime data obtained by solving a large number of instances, censoring runs that exceed a predetermined threshold of 12 h for CASS. On a test set of unseen instances, the model performs efficient selection, detecting the instances on which CASS is faster, and allowing the portfolio to improve on the performance of CPLEX alone. The overhead (23), compared to the performance of the oracle, is reported to be 8%, excluding a small additional factor due to the cost of computing features.

The runtime data for the two algorithms were obtained online.[13] The data consists of various small fixed size problem sets, and one large variable size set. After discarding a few instances, for which the time values were censored for both algorithms, the variable size set has 7, 145 instances, and the fixed size sets sum to 3, 519, for a total of 10, 664 problems. On these, CASS dominates on 2, 278, while CPLEX is faster on the remaining 8, 386. None of the two algorithms could solve all the problems before capping. The runtimes of the whole data set sum to almost nine years.

We repeated the experiment with GambleTA, solving the whole set of instances. As the solvers are not randomized, here the only difference among runs is the random ordering of problem instances. The runtimes in the data set are reported in seconds. Some runs were indicated with a 0 runtime, which means that they were too fast for the granularity of the clock (0.01). In these cases, we replaced the 0 value with 0.001. We then set t_{min} to 0.001, and left t_{max} at 10^{10}, which is oversized in this case, as the maximum runtime value in the set is 5×10^5. The initial time interval was set as $\Delta t_0 = 0.01$. The model was allowed only two covariate values, the number of bids and the number of goods, representing the size of the problems.

Figure 4a,b report the cumulative time during the task sequence, and the cumulative overhead, again comparing with the ideal performance of the oracle (23). The last block of Table 1 reports the same performance indicators described in the

[13] http://www.cs.ubc.ca/~kevinlb/downloads/db-data.zip.

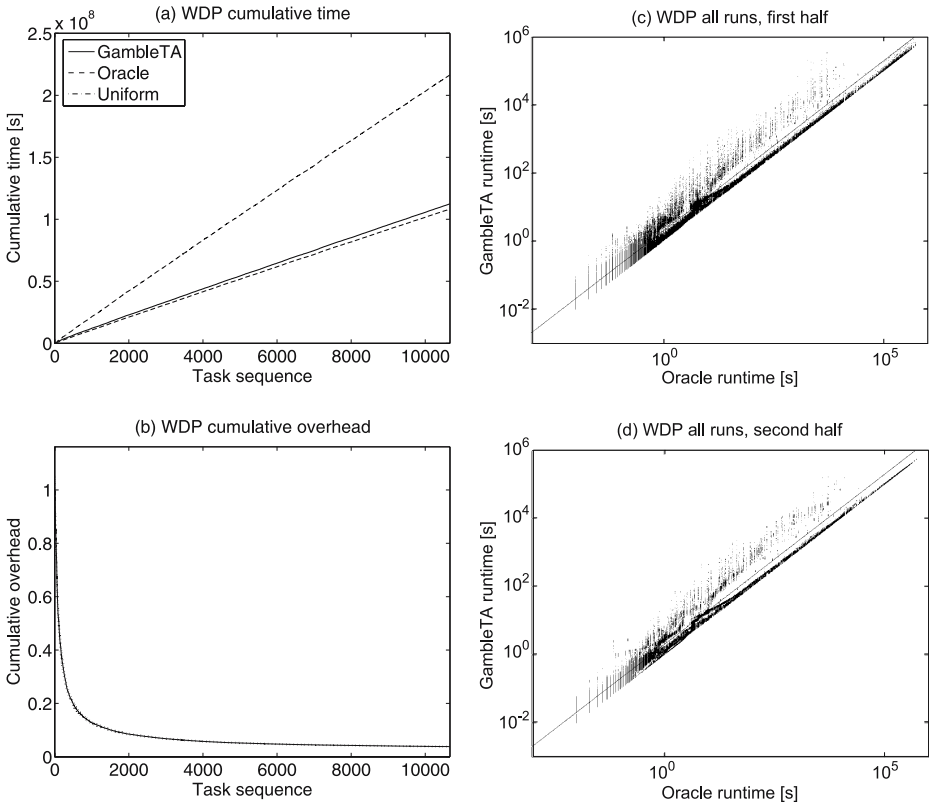

Fig. 4 WDP problems. **a** Cumulative time on the problem set. Upper 95% confidence bounds on 50 runs, with random reordering of the problems. GambleTA is our time allocator. Oracle is the lower bound on performance. Uniform is the (0.5,0.5) share. **b** Cumulative overhead (23) on the WDP problem set. Upper and lower confidence bounds. The overall final performance is 4%: in the second part only, the cumulative overhead is less than 3% (see Table 1). *Right column*: Performance of GambleTA compared to the oracle, on *all* problems and for all runs. $50 \times 10,664/2 = 266,600$ points per plot. **c** First half of the sequence. **d** Second half. The *vertical lines* reflect the fact that the algorithms are deterministic: runs differ only in the random order of the instances. The *diagonal* (*not marked*) would be the performance of Oracle. The *continuous line above* the diagonal would be the performance of Uniform. Note that this line is crossed by many runs. See also Table 1

previous subsection. The cumulative overhead during the second part of the problem sequence was less than 3%, while WTU performance was observed for about 15% of the problem instances. Figure 4c,d display a scatter-plot of the runtime of GambleTA against the one of the oracle, on all runs, again distinguishing among the first and second half of the problem sequence. Examining the latter, one can notice that the instances for which the runtime of GambleTA was worse than Uniform (represented by points above the line) can mostly be solved in less than 10 s. In other words, GambleTA is less precise for instances that have a minor impact on the cumulative runtime, which in this case is very close to the one of the Oracle. For this benchmark, Exp4 favored a mixture of two quantile allocators ($\alpha = 0.2, 0.3$), and the greedy contract allocator, which was discarded on the previous benchmark.

8 Discussion

The experiments gave quite impressive results. In the first case, the *dynamic* time allocator GambleTA managed to solve an algorithm selection problem that cannot be solved in a similar way by any static technique. In the second case, performance was competitive with the one of a static offline selection technique, built based on advanced knowledge of the problem domain, including dozens of problem-specific features, and which required quite a long training time. On this latter point we have to remark that in [45] no attempt was made at reducing the training cost, the interest of the authors being more focused on the precision of the estimated models.

The idea of performing algorithm selection based on runtime interaction with the algorithms is not at all new (see Section 2). Most fully dynamic methods are oblivious, i. e., with no knowledge transfer from one problem to the next; in most non-oblivious methods, the model is trained off-line, at a prohibitively high computational cost, as there is no principled method to decide when to stop the training phase. GambleTA takes the best of both worlds: the model allows to retain knowledge from past experiments, but is trained online, with a negligible overhead. The bandit problem solver Exp4 guarantees the optimal amount of exploration: the model is exploited as soon as it allows to improve on the uniform share. At this stage, the model is visibly rough, but can already serve the purpose of algorithm selection. Time allocation is fully dynamic, and shares can be updated an arbitrary number of times. To our knowledge, the most closely related approaches are [44, 59]. In the former, reinforcement learning, which can be seen as a generalization of bandit problems, is used, but at the algorithm level. The resulting method shares many of the positive features of GambleTA, as it is also online, and dynamic. In the dynamic method described in [58, 59], the algorithm priorities are updated repeatedly, but the dynamic sharing schedule is decided per set, and offline. In [40], the dynamic selection is only based on the initial evolution of the state, and the probability distributions are assumed to belong to a finite set, known a priori: also in this case the model is learned offline. In [10], an oblivious technique is presented, based on a contract on execution time, but with no knowledge transfer across problem instances. In [11, 12, 73], a bandit problem solver is used, but at a lower level, to perform oblivious per-instance algorithm selection. Compared to our previous work [17, 18], this article replaced the heuristic aspects, both in mapping model predictions to time allocation shares (Section 4), and in controlling the exploration-exploitation balance (Section 6).

At its upper level, the method is practically parameter-less. The bandit problem solver can set its only parameter optimally, based on the length of the task sequence. If the latter is not known, an initial estimate can be used, and periodically updated [1, 2]. In this case the optimal regret (20) is guaranteed with respect to the actual cumulative reward of the best expert. This modification was already tested, with analogous results. Different values of Δt_0 can only affect the performance with a logarithmic factor. The use of a logarithm for rewarding the algorithms allows to set t_{min} and t_{max}, respectively, to a very small and a very large value, such that only the knowledge of a very loose bound on execution time is required. Design decisions, including the choice of the time allocators, and model(s) to use, as well as the relative parameters, can be taken with a redundant approach, and their refinement can be left to Exp4. In the presented experiments we used a non-parametric model, which is slower to converge than a parametric one, but can converge to an arbitrary

distribution, so it does not require any a priori hypothesis about the runtime distributions of the algorithms. If such hypothesis are available, but unsure, an additional copy of each time allocator, based on the parametric model, can be added, and Exp4 will decide which model to use, with a $\sqrt{\ln 2}$ impact on its regret. At the lowest level, the choice of the algorithms composing $\mathcal{A}$, as well as the relative parameters, is still left to the user.

The amount of prior knowledge required by the experiments was quite low: the only inputs used for the model where one or two features, representing the size of the problems, and *time*. GambleTA has a *black-box* view of the algorithms, and can be applied to any decision problem solver. Optimisation problems can be treated, if a target on performance can be set in advance.

With respect to the previous parametric model [17, 18], the nonparametric method used here also allows to greatly reduce the modeling overhead, which is now negligible. According to [77], WW suffers from the curse of dimensionality, so it should be replaced in order to profit from a larger set of features. Including time-varying covariates, to condition the prediction also on the dynamic state of the algorithms, will also require more advanced models [43, 49, 55]: the approximation used in [17, 18] was abandoned. The regret of Exp4 will scale well with the number of algorithms K, and the number of time allocators N, with order $O(\sqrt{K \ln N})$. The time allocators perform an optimisation in $[0, 1]^K$, constrained to a space of size $K - 1$, as $\mathbf{s}$ has to sum to one. As there are no guarantees of unimodality, they will all suffer from the curse of dimensionality, so, for much larger algorithm sets, some approximations should be introduced.

GambleTA is highly modular. On the higher level, different bandit problem solvers could be compared, possibly starting from the variations described in [2]. On a level below, the model based time allocators could be replaced, or combined, with any other algorithm selection technique. It would be enough to express its decision as a share vector $\mathbf{s}$. In the simplest case, one could add an additional fixed allocator for each algorithm in the set, to quickly detect situations in which a single algorithm dominates the others. Also oblivious techniques, as the ones in [10, 21], could be easily integrated.

Section 4 is based on a single machine. In future research we plan to address a more realistic scenario, in which a *cluster* of machines has to be allocated, one algorithm per machine. Another alternative implementation could be based on setting the priorities of the algorithms through the operating system.

If we go back to the initial section, and look at the list of issues of a typical model-based algorithm selection technique, we realize that at least two of them do not hold for GambleTA. It never solves the same problem twice. And the simple fact of looking at the runtime of the algorithms allows it to improve its initial time allocation decisions. The first problem remains open: one feature that is still lacking is that the method cannot react to a misprediction of the model during a single task, which could be caused by an "outlier" problem instance, on which the behavior of the algorithms is radically different from what seen so far in the problem sequence. We will focus on this issue in our future research.

Acknowledgements We would like to thank Faustino Gomez, for his invaluable support during writing; Alexey Chernov and Jan Poland, for the always useful brainstorming; Yannet Interian, and Laura Spierdijk, for their expert advice; and Kevin Leyton-Brown, for kind assistance with the WDP data.

References

1. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: Gambling in a rigged casino: the adversarial multi-armed bandit problem. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 322–331. IEEE Computer Society Press, Los Alamitos, CA (1995)
2. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: The nonstochastic multiarmed bandit problem. SIAM J. Comput. **32**(1), 48–77 (2002)
3. Battiti, R., Protasi, M.: Reactive search, a history-sensitive heuristic for max-sat. ACM J. Exp. Algorithms **2**, 2 (1997)
4. Beck, C.J., Freuder, E.C.: Simple rules for low-knowledge algorithm selection. In: Régin, J.C., Rueher, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004. Lecture Notes in Computer Science, vol. 3011, pp. 50–64. Springer, Berlin Heidelberg New York (2004)
5. Beran, R.: Nonparametric regression with randomly censored survival data. Technical report, University of California, Berkeley (1981)
6. Berry, D.A., Fristedt, B.: Bandit Problems: Sequential Allocation of Experiments. Chapman & Hall, London (1985)
7. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Langdon, W., et al. (eds.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11–18. Morgan Kaufmann, San Mateo, CA (2002)
8. Boddy, M., Dean, T.L.: Deliberation scheduling for problem solving in time-constrained environments. Artif. Intell. **67**(2), 245–285 (1994)
9. Borrett, J.E., Tsang, E.P.K., Walsh, N.R.: Adaptive constraint satisfaction: The quickest first principle. In: Wahlster, W. (ed.) Proceedings of the 12th European Conference on Artificial Intelligence, pp. 160–164. Wiley, Chichester, UK (1996)
10. Carchrae, T., Beck, J.C.: Applying machine learning to low knowledge control of optimization algorithms. Comput. Intell. **21**(4), 373–387 (2005)
11. Cicirello, V.A., Smith, S.F.: Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming – CP 2004. Lecture Notes in Computer Science, vol. 3258, pp. 197–211. Springer, Berlin Heidelberg New York (2004)
12. Cicirello, V.A., Smith, S.F.: The max k-armed bandit: A new model of exploration applied to search heuristic selection. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA, pp. 1355–1361 (2005)
13. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Grefenstette, J. (ed.) Proceedings of an International Conference on Genetic Algorithms and Their Applications, Carnegie-Mellon University, July 24–26, 1985. Lawrence Erlbaum Associates, Hillsdale NJ (1985)
14. Etzioni, O.: Embedding decision-analytic control in a learning architecture. Artif. Intell. **49**(1–3), 129–159 (1991)
15. Fürnkranz, J.: On-line bibliography on meta-learning (2001). EU ESPRIT METAL Project (26.357): A Meta-learning Assistant for Providing User Support in Machine Learning Mining. Http://faculty.cs.byu.edu/~cgc/Research/MetalearningBiblio/metal-bib.html
16. Gagliolo, M., Schmidhuber, J.: Gambling in a computationally expensive casino: Algorithm selection as a bandit problem. NIPS 2006 Workshop on Online Trading of Exploration and Exploitation, Whistler Canada, 8 December 2006
17. Gagliolo, M., Schmidhuber, J.: A neural network model for inter-problem adaptive online time allocation. In: Duch, W., et al. (eds.) Artificial Neural Networks: Formal Models and Their Applications – Proceedings ICANN 2005. Part 2. Lecture Notes in Computer Science, vol. 3697, pp. 7–12. Springer, Berlin Heidelberg New York (2005)
18. Gagliolo, M., Schmidhuber, J.: Dynamic algorithm portfolios. Ninth international symposium on artificial intelligence and mathematics, Fort Lauderdale FL, 4–6 January 2006
19. Gagliolo, M., Schmidhuber, J.: Impact of censored sampling on the performance of restart strategies. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming – CP 2006. Lecture Notes in Computer Science, vol. 4204, pp. 167–181. Springer, Berlin Heidelberg New York (2006)
20. Gagliolo, M., Schmidhuber, J.: Learning restart strategies. In: IJCAI 2007 – Twentieth International Joint Conference on Artificial Intelligence (2007) (in press)

21. Gagliolo, M., Zhumatiy, V., Schmidhuber, J.: Adaptive online time allocation to search algorithms. In: Boulicaut, J.F., et al. (eds.) Machine Learning: ECML 2004. Proceedings of the 15th European Conference on Machine Learning, pp. 134–143. Springer, Berlin Heidelberg New York (2004) (Extended tech. report available at http://www.idsia.ch/idsiareport/IDSIA-23-04.ps.gz)
22. Gent, I., Walsh, T.: The search for satisfaction. Technical report, Department of Computer Science, University of Strathclyde (1999)
23. Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Mach. Learn. **54**(3), 187–193 (2004)
24. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
25. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1-2), 67–100 (2000)
26. Hansen, E.A., Zilberstein, S.: Monitoring and control of anytime algorithms: A dynamic programming approach. Artif. Intell. **126**(1–2), 139–157 (2001)
27. Harick, G.R., Lobo, F.G.: A parameter-less genetic algorithm. In: Banzhaf, W., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1867–1875. Morgan Kaufmann, San Mateo, CA (1999)
28. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975)
29. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco, CA (2004) ISBN: 1558698729
30. Hoos, H.H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. In: Gent, T.W.I.P., Maaren, H.v. (ed.) SAT 2000, pp. 283–292. IOS Press, Amsterdam, The Netherlands (2000) (http://www.satlib.org)
31. Horvitz, E., Ruan, Y., Gomes, C.P., Kautz, H.A., Selman, B., Chickering, D.M.: A bayesian approach to tackling hard computational problems. In: Breese, J.S., Koller, D. (eds.) UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, pp. 235–244. Morgan Kaufmann, San Mateo, CA (2001)
32. Horvitz, E.J., Zilberstein, S.: Computational tradeoffs under bounded resources (editorial). Artif. Intell. **126**(1–2), 1–4 (2001) (Special Issue)
33. Huberman, B.A., Lukose, R.M., Hogg, T.: An economic approach to hard computational problems. Science **275**, 51–54 (1997)
34. Hutter, F., Hamadi, Y.: Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Technical Report. MSR-TR-2005-125, Microsoft Research, Cambridge, UK (2005)
35. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming – CP 2006. Lecture Notes in Computer Science, vol. 4204, pp. 213–228. Springer, Berlin Heidelberg New York (2006)
36. Ibrahim, J.G., Chen, M.H., Sinha, D.: Bayesian Survival Analysis. Springer, Berlin Heidelberg New York (2001)
37. Jr., D.W.H., Lemeshow, S.: Applied Survival Analysis: Regression Modeling of Time to Event Data. Wiley, New York (1999)
38. Kaelbling, L., Littman, M., Moore, A.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
39. Kaplan, E., Meyer, P.: Nonparametric estimation from incomplete samples. J. Am. Stat. Assoc. **73**, 457–481 (1958)
40. Kautz, H.A., Horvitz, E., Ruan, Y., Gomes, C.P., Selman, B.: Dynamic restart policies. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI), pp. 674–681 (2002)
41. Van Keilegom, I., Akritas, M., Veraverbeke, N.: Estimation of the conditional distribution in regression with censored data : a comparative study. Comput. Stat. Data Anal. **35**, 487–500 (2001)
42. Keller, J., Giraud-Carrier, C.: ECML 2000 workshop on meta-learning: building automatic advice strategies for model selection and method combination. In: Eleventh European Conference on Machine Learning (ECML-2000), 30 May–2 June, Barcelona, Spain (2000)
43. van der Laan, M.J., Robins, J.M.: Unified Methods for Censored Longitudinal Data and Causality. Springer, Berlin Heidelberg New York (2003)

44. Lagoudakis, M.G., Littman, M.L.: Algorithm selection using reinforcement learning. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), pp. 511–518. Morgan Kaufmann, San Mateo, CA (2000)
45. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In: Hentenryck, P.V. (ed.) Principles and Practice of Constraint Programming – CP 2002. Lecture Notes in Computer Science, vol. 2470. Springer, Berlin Heidelberg New York (2002)
46. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: Methodology and a case study on combinatorial auctions. J. ACM (submitted)
47. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, vol. 1, pp. 366–371. Morgan Kaufmann, San Mateo, CA (1997)
48. Li, C.M., Huang, W.: Diversification and determinism in local search for satisfiability. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, Lecture Notes in Computer Science, vol. 3569, pp. 158–172. Springer, Berlin Heidelberg New York (2005)
49. Li, G., Doss, H.: An approach to nonparametric regression for life history data using local linear fitting. Ann. Stat. **23**, 787–823 (1995)
50. Li, H.: Censored data regression in high dimension and low sample size settings for genomic applications. Technical Report no. 9, University of Pennsylvania (2006)
51. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
52. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of sat problems. In: Proceedings 10th National Conf. on Artificial Intelligence, pp. 459–465 (1992)
53. Moore, A.W., Lee, M.S.: Efficient algorithms for minimizing cross validation error. In: Cohen, W.W., Hirsh, H. (eds.) Proceedings of the Eleventh International Conference (ICML). Machine Learning, pp. 190–198. Morgan Kaufmann, San Mateo, CA (1994)
54. Nelson, W.: Applied Life Data Analysis. Wiley, New York (1982)
55. Nielsen, J., Linton, O.: Kernel estimation in a nonparametric marker dependent hazard model. Ann. Stat. **23**, 1735–1748 (1995)
56. Nudelman, E.: Empirical approach to the complexity of hard problems. Ph.D. thesis, Stanford University, CA (2005)
57. 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.) Principles and Practice of Constraint Programming – CP 2004. Lecture Notes in Computer Science, vol. 3258, pp. 438–452. Springer, Berlin Heidelberg New York (2004)
58. Petrik, M.: Learning parallel portfolios of algorithms. Master's thesis, Comenius University (2005)
59. Petrik, M.: Statistically optimal combination of algorithms. SOFSEM 2005 – 31st Annual Conference on Current Trends in Theory and Practice of Informatics, Slovak Republic, 22–28 January, 2005
60. Petrik, M., Zilberstein, S.: Learning static parallel portfolios of algorithms. Ninth international symposium on artificial intelligence and mathematics, Fort Lauderdale FL, 4–6 January 2006
61. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), pp. 743–750. Morgan Kaufmann, San Mateo, CA (2000)
62. Pratt, L., Thrun, S.: Guest editors' introduction. Mach. Learn. **28**, 5 (1997) (Special Issue on Inductive Transfer)
63. Rice, J.R.: The algorithm selection problem. In: Rubinoff, M., Yovits, M.C. (eds.) Advances in Computers, vol. 15, pp. 65–118. Academic, New York (1976)
64. Robbins, H.: Some aspects of the sequential design of experiments. Bull. Am. Math. Soc. **58**, 527–535 (1952)
65. Russell, S.J., Wefald, E.H.: Principles of metareasoning. Artif. Intell. **49**(1–3), 361–395 (1991)
66. Russell, S.J., Zilberstein, S.: Anytime sensing, planning, and action: A practical model for robot control. In: Bajcsy, R. (ed.) Proceedings of the International Conference on Artificial Intelligence (IJCAI-93), Chambéry, France, pp. 1402–1407. Morgan Kaufmann, San Mateo, CA (1993)
67. Sałustowicz, R.P., Schmidhuber, J.: Probabilistic incremental program evolution. Evol. Comput. **5**(2), 123–141 (1997)

68. Schmidhuber, J.: Optimal ordered problem solver. Mach. Learn. **54**, 211–254 (2004) (Short version in NIPS 15, p. 1571–1578, 2003)
69. Schmidhuber, J., Zhao, J., Wiering, M.: Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. Mach. Learn. **28**, 105–130 (1997) (Based on: Simple principles of metalearning. TR IDSIA-69-96, 1996)
70. 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)
71. Solomonoff, R.J.: Progress in incremental machine learning. Technical Report. IDSIA-16-03, IDSIA (2003)
72. Spierdijk, L.: Nonparametric conditional hazard rate estimation: a local linear approach. Technical Report. TW Memorandum, University of Twente (2005)
73. Streeter, M.J., Smith, S.F.: An asymptotically optimal algorithm for the max k-armed bandit problem. In: Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI). AAAI Press, Menlo Park, CA (2006)
74. Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Berlin Heidelberg New York (1995)
75. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artif. Intell. Rev. **18**(2), 77–95 (2002)
76. Wang, J.L.: Smoothing hazard rate. In: Armitage, P., et al. (eds.) Encyclopedia of Biostatistics, 2nd Edition, vol. 7, pp. 4986–4997. Wiley, New York (2005)
77. Wichert, L., Wilke, R.A.: Application of a simple nonparametric conditional quantile function estimator in unemployment duration analysis. Technical Report. ZEW Discussion Paper No. 05-67, Centre for European Economic Research (2005)