Ann Math Artif Intell (2011) 61:49–86
DOI 10.1007/s10472-011-9228-z

Algorithm portfolio selection as a bandit problem with unbounded losses

Matteo Gagliolo · Jürgen Schmidhuber

Published online: 1 April 2011

Abstract We propose a method that learns to allocate computation time to a given set of algorithms, of unknown performance, with the aim of solving a given sequence of problem instances in a minimum time. Analogous *meta-learning* techniques are typically based on models of algorithm performance, learned during a separate *offline* training sequence, which can be prohibitively expensive. We adopt instead an *online* approach, named GambleTA, in which algorithm performance models are iteratively updated, and used to guide allocation on a sequence of problem instances. GambleTA is a general method for selecting among two or more alternative algorithm portfolios. Each portfolio has its own way of allocating computation time to the available algorithms, possibly based on performance models, in which case its performance is expected to improve over time, as more runtime data becomes available. The resulting *exploration-exploitation* trade-off is represented as a bandit problem. In our previous work, the algorithms corresponded to the arms of the bandit, and allocations evaluated by the different portfolios were mixed, using a solver for the bandit problem with expert advice, but this required the setting of an arbitrary bound on algorithm runtimes, invalidating the optimal regret of the solver. In this paper, we propose a simpler version of GambleTA, in which the allocators correspond to the arms, such that a single portfolio is selected for each instance. The selection is represented as a bandit problem with partial information, and an *unknown* bound on losses. We devise a solver for this game, proving a bound on its expected regret. We present experiments based on results from several

M. Gagliolo (✉)
CoMo, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
e-mail: mgagliol@vub.ac.be

J. Schmidhuber
IDSIA, Galleria 2, 6928 Manno (Lugano), Switzerland
e-mail: juergen@idsia.ch

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

solver competitions, in various domains, comparing GambleTA with another online method.

Keywords Algorithm selection · Algorithm portfolios · Meta learning · Online learning · Multi-armed bandit problem · Survival analysis · Las Vegas algorithms · Computational complexity · Combinatorial optimization · Constraint programming · Satisfiability

Mathematics Subject Classifications (2010) 68T05 · 68T20 · 68W27 · 68Q25 · 62N99 · 62G99

1 Introduction

Decades of research in the fields of Machine Learning and Artificial Intelligence brought us a variety of alternative algorithms for solving many kinds of problems. Algorithms often display variability in performance quality, and computational cost, depending on the particular problem instance being solved: in other words, there is no single "best" algorithm. While the "trial and error" approach is still popular, attempts to automate algorithm selection are not new [40], and have grown to form a consistent and dynamic area of research on *Meta-Learning* [43]. Algorithm *portfolios* [25] are a generalization of single algorithm selection, in which computation time is shared among the available algorithms: in this case, the result of the selection process is a *schedule* [42], according to which the algorithms are executed.

The objective of selection depends on the application at hand. In this paper, we consider the problem of allocating computation time to a set of *Las Vegas* algorithms (LVA) [3], i.e., algorithms whose performance on a single instance coincides with their runtime, which is in general a random variable. More precisely, we consider *generalized* LVAs [23], which are not guaranteed to solve a given instance in a finite time. This class includes all solvers for *decision* or *search* problems, where the aim is to find a solution, or prove that none exist; but also solvers for *optimization* problems, if the aim is to find a solution with a given quality, or to find a globally optimal solution, and prove its optimality. In all these cases, an obvious performance measure is runtime, especially because, in many problems of practical importance, this quantity grows exponentially in the size of the instance. We will therefore focus on the task of minimizing the time to solve a given set of problems.

Many selection methods follow an *offline* learning scheme, in which the availability of a large training set of performance data for the different algorithms is assumed. This data is used to learn a model that maps (*problem*, *algorithm*) pairs to expected performance, and later used to select and run, for each new problem instance, only the algorithm that is expected to give the best results. While this approach is reasonable, it ignores the computational cost of the initial training phase: collecting a representative sample of performance data has to be done via solving a set of training problem instances, and each instance is solved repeatedly, at least once for each of the available algorithms, possibly more if the algorithms are randomized. Furthermore, these training instances are assumed to be representative of future ones, as the model is not updated after training. In other words, there is an obvious trade-off between the *exploration* of algorithm performances on different problem instances, aimed at learning the model, and the *exploitation* of the best

algorithm/problem combinations, based on the model's predictions. This trade-off is typically ignored in offline algorithm selection, and the size of the training set is chosen heuristically. In our previous work [11, 16, 17], we have instead kept an *online* view of algorithm selection, in which the only input available to the meta-learner is a set of algorithms, of unknown performance, and a sequence of problem instances that have to be solved. Rather than artificially subdividing the problem set into a training and a test set, we iteratively update the model each time an instance is solved, and use it to guide algorithm selection on the next instance.

Bandit problems [2] offer a solid theoretical framework for dealing with exploration-exploitation trade-offs in the online setting. One important obstacle to the straightforward application of a bandit problem solver to algorithm selection is that most existing solvers assume a bound on losses to be available beforehand. Setting this bound is not trivial when considering runtime as a loss, as such quantity can easily vary across several orders of magnitude. In [17, 18] we dealt with this issue heuristically, fixing the bound in advance. In this paper, we present a modification of an existing bandit problem solver [6], which allows it to deal with an unknown bound on losses, while retaining a bound on the expected regret. This allows us to propose a simpler version of the algorithm selection framework GambleTA, originally introduced in [17].

Instead of selecting among alternative algorithms, GambleTA chooses among alternative *time allocators* (TA), i.e., methods for allocating computation time to a set of algorithms. One of the allocators is a uniform portfolio, running all algorithms in parallel with equal priority. The remaining ones are adaptive portfolios, where the schedule is set based on runtimes observed on previously solved instances: the performance of these allocators is therefore expected to improve as more runtime data is available. Problem instances are solved sequentially. For each instance, the bandit problem solver picks one of the allocators; once the instance is solved, the total time spent is considered as a loss, to evaluate the performance of the TA, while the runtimes of the single algorithms are used to update the performance models used by the adaptive TAs. As the bandit problem solver can provide a bound on the regret with respect to the best arm, the total runtime spent will converge to the one of the best TA.

The rest of the paper is organized as follows. Section 2 precises some terminology used in the rest of the paper, and discusses related work. Section 3 briefly recalls the time allocators introduced in [17], and describes a more recent one from [47]. Section 4 presents the novel version of GambleTA. Section 5 introduces Exp3Light-A, a bandit problem solver for unbounded loss games, along with its bound on regret. Experiments with data from solver competitions are reported in Section 6, comparing with results from [45]. Section 7 concludes the paper.

2 Related work

Before presenting related research, it will be useful to precise the meaning of a few key concepts: whenever possible, the most widely used term has been adopted, accompanied by a reference. In general terms, a *time allocator* (TA) can be defined as any method which solves problem instances by allocating computation time to a set of algorithms, on one or more processors. In other words, a time allocator does

not *solve* problem instances directly, it *uses* available solvers to this aim. We will often refer to a set of N algorithms $\mathcal{A} = \{a_1, \ldots, a_N\}$; and to a set of M problem instances $\mathcal{B} = \{b_1, \ldots, b_M\}$. To solve these, the algorithms are executed on one or more processors, according to a *schedule* $\mathbf{s} \in \mathcal{S}$ [8]. The role of the TA is precisely that of generating[1] a schedule $\mathbf{s}$, which can be used to solve $\mathcal{B}$ using $\mathcal{A}$.

Existing TAs may differ in the way the schedule $\mathbf{s}$ is represented, as well as in its allowed values $\mathcal{S}$. In the following, we limit the discussion to methods where the algorithms are not modified, and cannot interact;[2] however, we do not pose further restrictions on what a TA can do with the a_n. If an algorithm is randomized, it may be replicated an arbitrary number of times, each copy being initialized with a different random seed. Note that the a_n may also be different parameter settings of the same algorithm (*parameter selection*), or different models, to be used by a single algorithm (*model selection*).

Time allocation can be performed once for a whole set of problem instances, or repeated independently for each instance (*per set* and *per instance* allocation, respectively [26]). Another independent classification can be made among *static* and *dynamic* schedules [36]. Static schedules are stationary, and are set before starting any a_n. Dynamic schedules can be a function of time $\mathbf{s} = \mathbf{s}(t)$, i.e., they can change *while* the algorithms are being executed.

Rather than being equally spread across the above categories, the literature on time allocation is clustered around specific combinations of these ideas. A large corpus of research considers the selection of a single algorithm, independently for each instance. In these papers, selection is based on performance models, conditioned on features of the instance, and learned offline (Section 2.1). Another set of papers addresses the sequential allocation of multiple runs of the same algorithm, or of multiple algorithms in parallel, based on the runtime distributions of the algorithms on the current instance, which are assumed to be available. This line of research includes the foundational work on restart strategies and algorithm portfolios (Section 2.2). More recently, some authors considered the optimal allocation based directly on a runtime sample, proving that finding an optimal allocation is itself an NP-hard problem. The runtime data is collected both offline and online (Section 2.3). Further references can be found in [13, 44].

2.1 Model based selection, per instance

Per instance selection is usually based on a predictive model of the performance of each algorithm, conditioned on *features* of the instance, a set of numerical or categorical variables related to its difficulty. This idea was proposed already by [40], and later adopted by several authors. Its implementation is typically based on an *offline* approach: a set of "training" problem instances is solved repeatedly with each of the available algorithms, in order to collect a performance sample. Based

[1] A time allocator is itself an algorithm, but, to avoid confusion, we will use the terms *algorithm* or *solver* to refer to the algorithms in the set $\mathcal{A}$; and to the terms *(time) allocator*, *method* or *technique* to refer to the upper-level TA which *uses* the algorithms in the set.

[2] Examples of time allocation which we do not consider here include continuous parameter tuning and control [4]; and sequential composition of algorithms, as search in program space [12], and anytime algorithm scheduling [24].

on this sample, a predictive model of performance is learned, mapping (*instance*, *algorithm*) pairs to the expected performance: in practice, this can be obtained using a separate model for each algorithm, conditioned on instance features. The models can later be used to perform per instance selection: for each new instance, the features are evaluated, and the performance of each algorithm is predicted. Based on these predictions, the algorithm expected to obtain the best performance is selected, and used to solve the instance.

For LVAs, an implementation of this idea was proposed in [30], and later improved in [35, 49]. In this case, the runtime is predicted, and minimized. For each available algorithm, an *empirical hardness model* [30] is learned offline, based on the runtimes on several training problem instances: after the initial learning phase, the expected performance of each algorithm on a new problem instance is predicted, based on instance features, and the best algorithm is selected accordingly.[3] A particularly successful application of this approach is SATzilla [35, 48, 49], which won several medals at the last SAT competitions (see Section 6). SATzilla is a full-fledged algorithm selection method for SAT solvers, in which some of the design decisions are also automated, such as the composition of the set of algorithms, as well as the choice of which instance features to use.

2.2 Model based allocation, per instance

The selection of a single algorithm is not the most general way of exploiting the performance diversity of a set of LVAs: different algorithms can be combined running them in parallel, in an algorithm portfolio [21, 25]. Moreover, if the algorithms are randomized, their performance may vary among different runs: in some cases, even the performance of a single algorithm may be improved combining different runs, in a portfolio, or periodically restarting the algorithm with a different random seed, according to a *restart strategy* [32]. In this line of research, the allocation is based on the *runtime distribution* (RTD) of each algorithm on the current instance, assumed to be available. The RTD of the resulting portfolio is evaluated analytically, and optimized according to some criterion. Time allocation can be *static*, e.g. represented by a *resource sharing* schedule, where each algorithm uses a constant share of computation time [21, 25]; or dynamic, e.g. according to a *task switching* schedule, specifying which algorithm is active at a given time [9]. The problem of estimating the RTDs is not tackled, so this line of work remains at a theoretical level.

While the basic idea can be traced back to the field of global optimization (see e.g. [29], [34]), it is first applied to search algorithms by [25], who borrow the term *portfolio* from finance, to point out that the method can reduce the "risk" of wasting computation time. A theoretical and empirical validation of the portfolio approach is provided by [21], who find its advantage to depend essentially on the RTD of the algorithm(s) at hand, and suggest that practical approaches to algorithm portfolio design should feature mechanisms for estimating such distributions.

The above papers consider static shares, also termed *resource sharing* schedules [42]. In other work on algorithm portfolios, a different machine model is considered,

[3] While they actually perform single algorithm selection, the authors label their approach "portfolio", in the more general sense of a combination of several algorithms.

where a single algorithm is active at a given time, and allocation consists in selecting a dynamic schedule, termed *task-switching* [42, 47] or *suspend-resume* schedule [9], according to which the execution of the different algorithms is interleaved. Such a schedule can be represented as a sequence of pairs (n, τ), indicating the index n of the algorithm, and the corresponding computation time value τ. Often, restrictions on the possible schedules are considered, for example a finite set of possible time values τ, allowing to represent schedule space as a tree. One such approach based on runtime distributions is proposed in [8, 9], respectively focusing on algorithms running on separate processors, and alternating on a single processor. Also in this work, the RTDs are assumed to be known *a priori*, and the expected value of a cost function, accounting for both wall-clock time and resources usage, is minimized. A task switching schedule is evaluated offline, using a branch-and-bound algorithm to find the optimal one in the tree of possible schedules. The computational complexity of the approach is exponential in the number of algorithms, due to the tree search.

2.3 Model free allocation, per set

Recently, some authors proposed practical methods to evaluate per set portfolios, directly based on a runtime sample, without explicitly modeling the RTDs [38, 42, 47]. Also in this case the runtime sample is collected offline, with the exception of [47], who also consider online versions of their methods. The problem of optimizing the share is proved to be NP-hard [38, 42]: a 4-optimal approximation, which can be evaluated in polynomial time, is proposed in [47].

In [38], a per set, static, resource sharing schedule is evaluated, for both decision and optimization problem solvers. For optimization, the solution quality at a preassigned time value is maximized, while the total runtime is minimized for decision problems. In both cases, the schedules are evaluated based on the actual performances of each algorithm, available beforehand, and bounds on the performance on further instances are given based on the PAC learning framework [33], under the assumption that the distribution generating instances remains the same. The problem of optimizing the share is found to be NP-hard. Dynamic resource sharing schedules are considered in [36, 37]. The schedules can change at a finite set of equally spaced time values, and the last allocation is kept indefinitely, resulting in a total of b time intervals. The problem of selecting the schedule is formulated as a Markov Decision Process (MDP, [39]), and a variation of dynamic programming is used to select the per-set optimal schedule.

Both resource sharing and task switching, among deterministic algorithms, are considered in [42], where the time to solve a set of problem instances is minimized with a static, per set schedule. It is proved that, given a resource sharing schedule, it is possible to find a task switching schedule whose performance is equal or better. After showing that even finding an ε approximation[4] of the per-set-optimal resource sharing schedule is NP-hard for some $\varepsilon > 0$, the authors present an algorithm for evaluating the optimal resource sharing schedule, whose complexity is $O(M^{N-1})$, and mention another algorithm for the optimal task switching schedule, based on dynamic programming, with complexity $O(M^{N+1})$. They then propose an offline

[4] I.e., a share which solves $\mathcal{B}_0$ in at most $(1 + \varepsilon)$ times the runtime of the optimal share.

allocator: during an initial learning phase, the runtime of each algorithm on each instances is observed, and an optimal schedule is evaluated. The schedule can then be used to solve further problem instances, generated from the same distribution, with a probabilistic bound on the regret, depending on sample size.

Based on this work, [47] propose a polynomial time method for evaluating a "greedy" 4-optimal task switching schedule, proving that finding a better approximation is NP-hard. They also propose an online version of their allocator, in which some of the instances are solved with all available algorithms, in order to collect training data, based on which the greedy schedule is updated, and used to solve other instances. The decision of whether to "explore" algorithm performance, or "exploit" the greedy schedule, is taken independently for each instance, using a *label efficient forecaster*[5] [5], which allows to bound the regret compared to the offline greedy strategy, and whose complexity is also exponential in N.

A different online method is proposed in [46], where a per set optimal task switching schedule is learned while solving a sequence of instances, again with a 4-optimal performance in the limit. Recall that a task-switching schedule can be specified with a sequence of pairs (n, τ), meaning "run a_n for a time τ". In this case, the schedule is composed of L segments of equal length, and its duration is limited by a timeout B, so $\tau = B/L$. For each segment, a separate copy of a bandit problem solver (Exp3 [2]) picks one of the N algorithms, receiving a reward if the algorithm solves the instance. In this way, each BPS learns to select a single algorithm, to be run in the corresponding slice of the schedule.

While evaluated online, in both cases the allocation is still per set. [46] introduces also per instance selection, limited to discrete features: for v distinct feature values, v copies of the BPS learn separately the corresponding best choice, for each slice of the schedule. This is equivalent to subdividing instances based on feature values, and repeating the process independently for each set. Both approaches are validated using runtime data from several solver competitions [45]. We compare them to our method in Section 6, using the same data.

3 Time allocators

In this section we recall the time allocators originally introduced in [17], and describe an alternative use of the greedy allocator from [47].

Consider a portfolio of N algorithms $\mathcal{A} = \{a_1, \ldots, a_N\}$, solving the same problem instance. The a_n are executed in parallel, and share the computational resources of a single machine according to a *resource sharing* schedule, or *share*, $\mathbf{s} = \{s_1, .., s_N\}$, $s_n \geq 0$, $\sum_{n=1}^{N} s_n = 1$, i.e., the schedule space $\mathcal{S}$ is the standard $(N-1)$ *simplex* Δ^{N-1}. Here and in the following we assume an "ideal" machine, with no task switching overhead, where, for any amount δt of machine time, a portion $\delta t_n = s_n(t)\delta t$ is allocated to a_n. With this notation, single algorithm selection can be represented

[5] Label efficient forecasting is a variant of the bandit problem where the loss is a function of an unknown outcome, related to each trial (in this case the runtimes of all algorithms on a single instance): the gambler can decide, independently for each trial, to obtain the actual outcome, paying an additional cost (which in this case coincides with the sum of the runtimes of all algorithms on the current instance).

as a share with a single $s_n = 1$. A *uniform* algorithm portfolio is instead executed according to a share $\mathbf{s}_\mathcal{U} = (1/N, ..., 1/N)$: note that this simple allocation solves any problem instance in a time $N \min_n\{t_n\}$, so it is always a factor N worse than the best possible performance. In the following two subsections, we show how the RTD of a portfolio can be evaluated, and used to optimize the share.

3.1 RTD of a portfolio

The runtime of each a_n on the current problem instance is a random variable $T_n \in [0, \infty]$, fully described by its RTD. A common representation[6] of the RTD is the *cumulative distribution function* (CDF) $F_n(t)$, expressing the probability that a_n will solve the instance by time t. In the following, where necessary, we will refer to this function with the term *instance RTD*, to distinguish it from the *set RTD*, describing instead the probability of solving a randomly picked instance from a given set.

The instance is solved as soon as one of the algorithms finds a solution: therefore, also the runtime of the portfolio $T_\mathcal{A}$ is a random variable, depending on the T_n, and on the share $\mathbf{s}$, as

$$T_\mathcal{A}(\mathbf{s}) = \min_n \left\{ \frac{T_n}{s_n} \right\}. \tag{1}$$

The distribution of $T_\mathcal{A}$ can then be calculated based on the RTDs of the single algorithms $F_n(t)$, and the share $\mathbf{s}$: the derivation is more intuitive if we reason in terms of the *survival function* $S(t) = 1 - F(t)$. At a given time t, each a_n has used a share of computation time $s_n t$. The probability $S_\mathcal{A}(t; \mathbf{s})$ of *not* having solved the instance is equal to the joint probability that no single algorithm has found a solution within its time share. Assuming the runtimes T_n to be independent, this joint probability can be evaluated as the product of the individual survival functions $S_n(s_n t)$:

$$S_\mathcal{A}(t; \mathbf{s}) = \prod_{n=1}^{N} S_n(s_n t), \tag{2}$$

which, in CDF form, corresponds to

$$F_\mathcal{A}(t; \mathbf{s}) = 1 - \prod_{n=1}^{N} [1 - F_n(s_n t)]. \tag{3}$$

Note that the assumption of independence of the T_n, which allows to express (2) as a product, is justified by the fact that the a_n do not interact, and by the use of the *actual* RTDs of the a_n on the current instance.

3.2 Static allocation

Equation (2) describes the RTD of a portfolio as a parametric function of the share $\mathbf{s}$, allowing to formulate time allocation as an optimization problem, in which some

[6] So common that the two terms are used interchangeably by many authors, including us.

quantity derived from the RTD is optimized with respect to $\mathbf{s}$. In [17], we proposed three alternative allocators:

Definition 1 (Expected time allocator) TA_E. The expected runtime value is minimized with respect to $\mathbf{s}$:

$$\mathbf{s}_E = \arg\min_{\mathbf{s}} E\{T_{\mathcal{A}}(\mathbf{s})\}. \tag{4}$$

Definition 2 (Contract allocator) $\mathrm{TA}_C(t_u)$. If an upper bound, or *contract*, t_u on runtime is imposed, one can instead use (2) to select the $\mathbf{s}$ that maximizes the probability of solution within the contract, minimizing $S_{\mathcal{A}}(t_u; \mathbf{s})$:

$$\mathbf{s}_C(t_u) = \arg\min_{\mathbf{s}} S_{\mathcal{A}}(t_u; \mathbf{s}). \tag{5}$$

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

$$\mathbf{s}_Q(\alpha) = \arg\min_{\mathbf{s}} t_{\mathcal{A}}(\alpha; \mathbf{s}). \tag{6}$$

All three allocators above require optimizing a function of $\mathbf{s} \in \Delta^{N-1}$. If the models of the S_n are parametric, a gradient of the above quantities can be computed analytically, depending on the particular parametric form: in any case, optimization can be performed numerically. The surfaces optimized by the three methods will in general be different, and have minima at different values of $\mathbf{s}$: in the last two cases, they will also depend on the values of t_u and α respectively. Unfortunately, in no case there is a guarantee of unimodality, so these methods are subject to a "curse of dimensionality", determined by the fact that the search space size is exponential in the number of algorithms N. In practice, even for a small N, it is advisable to repeat the optimization process multiple times, with different random initial values for $\mathbf{s}$. Examples of these surfaces for $N = 2$ algorithms are reported in [13, 17]. The allocation of multiple CPUs is derived in [19].

In the literature, the instance RTDs are assumed to be available a priori. Aiming at a practical implementation of our methods, we looked at the task of estimating such distributions, finding a vast amount of useful research in the field of survival analysis, a branch of statistics which studies the distribution of random events in time [28]. Such estimation can be carried out using regression models, conditioned on features of the instance [13, 14, 17]. This is analogous to the scalar regression of expected runtime in single algorithm selection (Section 2.1), but here the whole RTD is predicted. An advantage of survival analysis methods is that they allow to take into account *censored* observations, as the duration of unsuccessful runs. This allows to use the portfolio itself to sample the RTDs, avoiding the need of solving the same instance multiple times: when an instance is solved, we obtain one uncensored runtime observation for the successful algorithm, and one censored observation for each of the $N - 1$ remaining algorithms. These observations can then be used to update the corresponding models. Using estimates of the instance RTDs results in suboptimal allocations, as it violates the independence assumption behind (2)

[13, 14]; however, comparative experiments with more correct models showed that the impact on allocation performance is small [13, Sec. 9.7].

3.3 Dynamic allocation

A dynamic schedule is more general than a static one: [42] also proved that, with respect to total runtime, the best per set task switching allocation cannot be worse than the best per set resource sharing allocation. Their proof is based on the runtimes $\{t_n(m)\}$ of each the N deterministic algorithms on each of the M instances. Consider a single instance b, and randomized algorithms $\{a_n\}$. Simply replacing the $t_n(m)$ with the runtimes corresponding to the m-th run of a_n, with random seed m, we can reformulate their theorem referring it to the runtime on a single instance, summed for M different runs. If M is large, dividing by M we would obtain an estimate of the expected runtime. We can therefore state that, with respect to expected runtime, also the best per instance task switching allocation cannot be worse than the best per instance resource sharing allocation.

A task switching schedule can be described by a sequence of pairs (n, τ), indicating the index of the algorithm, and the corresponding computation time value. In our notation, it can be represented as a time-varying resource sharing schedule $\mathbf{s}(t)$, with the additional constraint that each $s_n(t) \in \{0, 1\}$, such that only one a_n has $\mathbf{s}_n(t) = 1$ for any t. Removing such constraint allows to obtain more general *dynamic* shares $\mathbf{s}(t)$. In practice, most work on task switching schedules considers a discrete, finite set of values for τ, defining time allocation as a discrete optimization problem. In this section we consider dynamic shares $\mathbf{s}(t)$ which can change their value at a discrete set of time intervals $\{\tau_1, \tau_2, \ldots\}$, but we allow such set to be infinite.

If a time allocator TA can be conditioned on the current state $\mathbf{x}_n$ of each algorithm, or at least on the runtime y_n spent so far, such a dynamic $\mathbf{s}(t)$ can be obtained simply updating a static share $\mathbf{s}$ at predefined time intervals $\{\tau_1, \tau_2, \ldots\}$, as described in Algorithm 1.

Algorithm 1 Dynamic Algorithm Portfolio

```
Problem instance b
Algorithm set A = {a_1, ..., a_N}
Static/dynamic features x_n of a_n on b.
A(b; s) solves instance b with share s
Time allocator TA(b, {x_n}, {y_n})
Set y_n := 0, n = 1, ..., N
while b not solved do
   update τ; s := TA(b, {x_n}, {y_n})
   run A(b; s) for a maximum time τ
   update x_n; y_n := y_n + s_n τ, n = 1, ..., N
end while
```

For simplicity, in Algorithm 1 we considered a resource sharing $\mathbf{s}$, but the same approach can be applied to an arbitrary time allocator, with a different schedule representation, provided that its allocation is a nontrivial function of $\{\mathbf{x}_n, y_n\}$, or just $\{y_n\}$. If the allocator is based on performance models, this can be obtained conditioning the models on $\{\mathbf{x}_n, y_n\}$. Regarding the RTD based allocators presented

in the previous section, each of them can be used in Algorithm 1, if the RTD models can be conditioned on time-varying covariates $\mathbf{x}_n$; otherwise, each $S_n(t)$ can simply be updated conditioning on the time spent y_n. Writing $t' = t - y$, the RTD of the portfolio after a time $y = \sum_n y_n$ is

$$S_{\mathcal{A}}(t') = \prod_{n=1}^{N} \frac{S_n(y_n) - S_n(t' + y_n)}{S_n(y_n)}. \tag{7}$$

An additional design decision is required to set the sequence of time intervals τ. These may be derived according to some optimality criterion, dictated by the TA itself, or set heuristically. Regarding the latter option, using a constant τ has the disadvantage of requiring an initial guess of the typical runtimes of the algorithms, with the risk of updating time allocation too often, or too rarely, if the guess proves wrong. In our experiments, we simply doubled τ at each round, starting from a small initial value τ_1: in this way, time allocation is updated $O(\log_2 t_{\mathcal{A}})$ times during a run of duration $t_{\mathcal{A}}$. The infinite set of time values considered is then $\{\tau_1, 2\tau_1, \ldots, 2^i\tau_1, \ldots\}$.

3.4 Greedy task-switching schedules

Another allocator which can be evaluated dynamically, as in Algorithm 1, is the greedy task-switching schedule (8) of [47]. This method consists in concatenating a sequence of pairs (n, τ) such that running a_n for a time τ maximizes the rate at which instances are solved.

Be $G(t; \mathbf{s})$ a function expressing the number of instances in $\mathcal{B}$ that would be solved in a time t executing $\mathcal{A}$ according to a task switching schedule $\mathbf{s}$. The greedy schedule is evaluated incrementally, appending to the current $\mathbf{s}_k = \{(n_1, \tau_1), \ldots, (n_k, \tau_k)\}$ the pair (n_{k+1}, τ_{k+1}) such that

$$(n_{k+1}, \tau_{k+1}) = \arg\max_{(n,\tau)} \frac{G(\tau + \mathrm{T}_k; \{\mathbf{s}_k, (n, \tau)\}) - G(\mathrm{T}_k; \mathbf{s}_k)}{\tau}, \tag{8}$$

where $\mathrm{T}_k = \sum_{j=1}^{k} \tau_j$ is the duration of $\mathbf{s}_k$. The functions $G(t; \mathbf{s})$ can be evaluated given the runtimes $\{t_n(m)\}$, as $G(t; \mathbf{s}) = \mathrm{card}(\{b_m : t_{\mathcal{A}}(m; \mathbf{s}) \leq t\})$.

The expected value of $G(t; \mathbf{s})$ can be estimated for an arbitrary $\mathbf{s} = \{(i_1, \tau_1), (i_2, \tau_2), \ldots\}$ based on the RTDs on the set of instances $\{F_n(t)\}$, as

$$G(t; \mathbf{s}) = M \sum_{n=1}^{N} F_n\left(\sum_{n_j == n} \tau_j\right). \tag{9}$$

Given (7), the rate maximized in (8) can then be written as $F_n(t|t \geq y_n)/(t - y_n)$, and the schedule evaluated dynamically: in this case, the sequence $\{\tau_1, \tau_2, \ldots\}$ is set by the TA itself. We can then define the following:

Definition 4 (Greedy allocator) TA_S. Be $y_n^{(k)} = \sum_{n_j = n} \tau_j$ the time allocated so far to a_n by the schedule $\mathbf{s}_k = \{(n_1, \tau_1), \ldots, (n_k, \tau_k)\}$. The next portion of the schedule is

$$(n_{k+1}, \tau_{k+1}) = \arg\max_{(n,\tau)} \frac{F_n(\tau + y_n^{(k)}) - F_n(y_n^{(k)})}{[1 - F_n(y_n^{(k)})]\tau}. \tag{10}$$

Based on [47], it should be possible to prove that using the instance RTDs in (10) would generate a per instance 4-optimal task switching schedule. An advantage of (10) over our allocators is that it can be evaluated in a time $O(N)$, as it requires N line searches to find the optimal τ for each algorithm.

4 Online time allocation

In its most basic form [41], the *multi-armed bandit* problem is faced by a gambler, playing a sequence of trials against an K-armed slot machine. At each trial, the gambler chooses one of the available arms, whose losses are randomly generated from different *stationary* distributions. The gambler incurs in the corresponding loss, and, in the *full information* game, she can observe the losses that would have been paid pulling any of the other arms. A more optimistic formulation can be made in terms of positive rewards. The aim of the game is to minimize the *regret*, defined as the difference between the cumulative loss of the gambler, and the one of the best arm. A bandit problem solver (BPS) can be described as a mapping from the history of the losses observed so far, to a probability distribution $\mathbf{p} = (p_1, ..., p_K)$ over the K arms, which will be used to pick an arm at the subsequent trial.

More recently, the original restricting assumptions have been progressively relaxed, allowing for *non-stationary* loss distributions, *partial* information (only the loss for the pulled arm is observed), and *adversarial* bandits that can set their losses in order to deceive the player. In [2], a reward game is considered, where no statistical assumptions are made about the process generating the rewards. In this *non-oblivious adversarial* setting, before the player makes his choice, the losses of each arm are set as an arbitrary function of the entire history of the game. Note that this setting encompasses stationary and non-stationary stochastic bandits, as nothing forbids the adversary to set the losses randomly. Notwithstanding these pessimistic hypotheses, the authors devise probabilistic gambling strategies for the full and the partial information games, with bounds on the expected regret.

Let us now see how to represent algorithm selection for *decision* problems as a bandit problem, with the aim of minimizing solution time. Consider a sequence $\mathcal{B} = \{b_1, \ldots, b_M\}$ of M instances of a decision problem, for which we want to minimize solution time, and a set of N algorithms $\mathcal{A} = \{a_1, \ldots, a_N\}$, such that each b_m can be solved by each a_k. It is straightforward to describe algorithm selection as a multi-armed bandit problem, where "pick arm k" means "run algorithm a_k on next problem instance". Runtimes t_k can be viewed as losses, generated by a rather complex mechanism, i.e., the algorithms a_k themselves, running on the current problem. The information is partial, as the runtime for other algorithms is not available, unless we decide to solve the same problem instance again. In a worst case scenario one can receive a "deceptive" problem sequence, starting with problem instances on which the performance of the algorithms is misleading, so this bandit problem should be considered adversarial. As BPS typically minimize the regret with respect to a single arm, this approach would allow to implement *per set* selection, of the overall best algorithm. An example can be found in [18], where we presented an online method for learning a per set estimate of an optimal restart strategy.

Unfortunately, per set selection is only profitable if one of the algorithms dominates the others on all problem instances. This is usually not the case: it is often observed in practice that different algorithms perform better on different problem

instances. A per instance selection scheme, taking an independent decision for each problem instance, or even a per set schedule, where multiple algorithms are combined, can both have a great advantage.

One possible way of exploiting the nice theoretical properties of a BPS in the context of algorithm selection, while allowing for the improvement in performance of per instance selection, is to use the BPS at an upper level, to select among alternative algorithm selection techniques. Consider again the algorithm selection problem represented by $\mathcal{B}$ and $\mathcal{A}$. Introduce a set of K arbitrary time allocators, as defined in Section 3, $\mathcal{T} = \{\text{TA}_1, \ldots, \text{TA}_K\}$. At this higher level, one can use a BPS to select among different time allocators, working on the same algorithm set $\mathcal{A}$. In this case, "pick arm k" means "use TA_k to allocate time to $\mathcal{A}$ for solving the next problem instance". In the long term, the BPS would allow to select, on a *per set* basis, the TA_k that is best at allocating time to algorithms in $\mathcal{A}$ on a *per instance* basis. The resulting "Gambling" Time Allocator (GambleTA) is described in Algorithm 2, where $t_k(m)$ is the runtime of TA_k on instance b_m.

Algorithm 2 GambleTA ($\mathcal{A}$, $\mathcal{T}$, BPS) Gambling Time Allocator.

```
Algorithm set A with N algorithms
Problem set B with M instances
A set T = {TA_k} of K time allocators
A bandit problem solver BPS

initialize BPS (K, M)
for each instance b_j, j = 1, ..., M do
   pick time allocator I(j) = k with probability p_k from BPS.
   solve problem b_j using TA_I on A, in a time t_I(j)
   observe loss l_{I(j)} = t_{I(j)}(j)
   update BPS
end for
```

If the BPS allows for non-stationary arms, it can also deal with time allocators that are *learning* to allocate time. This is indeed the motivation for adopting this two-level selection scheme, as it allows to combine in a principled way the exploration of algorithm behavior, which can be represented by the uniform allocator, and the exploitation of this information by a model-based allocator, whose model is being learned online, based on results on the sequence of problems met so far. If more time allocators are available, they can be made to compete, using the BPS to sample their performances.

An interesting feature of this selection scheme is that the hypothesis that each algorithm is capable of solving each problem can be relaxed, requiring instead that *at least one* of the a_n can solve a given b_m, but *each* TA_k can solve each b_j: this can be ensured in practice by eventually[7] imposing a $s_n > 0$ for all a_n. This allows to use interesting combinations of complete and incomplete solvers in $\mathcal{A}$ [13, 17].

[7] In our implementation, the time allocators are based on RTD models, updated dynamically by conditioning on runtime spent. When the allocation cannot be evaluated (e.g. because the samples are still empty, or there are no observations larger than the current runtime), the allocation defaults to the uniform share. In this way, all algorithms will eventually be executed, satisfying the hypothesis.

Note that any bound on the regret of the BPS will determine a bound on the regret of GambleTA with respect to the best time 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 is effective, a bound can still be obtained by including the uniform share in the set of TAs.

4.1 Time allocators as experts

The original version of GambleTA (GambleTA4 in the following) [17] was based on a more complex alternative, inspired by the bandit problem with *expert* advice [2]. In this problem, two games are going on in 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 arms. The experts can be arbitrary functions of the history of observed rewards, and give a different advice for each trial. At a higher level, a *full information* game is played, with the K 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 advises, in order to generate the distribution for the lower level arms. In GambleTA4, the time allocators played the role of the experts, each suggesting a different resource sharing $\mathbf{s}$, on a per instance basis; while the arms of the lower level game were the N algorithms, to be run in parallel with the mixture share. Exp4 [2] was used as the BPS: Unfortunately, the bounds for this solver cannot be extended to GambleTA4 in a straightforward manner, as the loss function itself is not convex. Moreover, Exp4 cannot deal with unbounded losses, so we had to adopt an heuristic reward attribution instead of using the plain runtimes. GambleTA4 is therefore less sound than GambleTA: however, compared on a toy problem, the two versions displayed a similar performance [20].

5 Unbounded losses

A common issue of the above approaches is the difficulty of setting reasonable upper bounds on the time required by the algorithms. This renders a straightforward application of most BPS problematic, as a known bound on losses is usually assumed, and used to tune parameters of the solver. Underestimating this bound can invalidate the bounds on regret, while overestimating it can produce an excessively "cautious" algorithm, with a poor performance. Setting in advance a good bound is particularly difficult when dealing with algorithm runtimes, which can easily exhibit variations of several order of magnitudes among different problem instances, or even among different runs on a same instance [22].

Some interesting results regarding games with *unbounded* losses have recently been obtained. In [6, 7], the authors consider a full information game, and provide two algorithms which can adapt to unknown bounds on signed rewards. Based on this work, [1] provide a Hannan consistent algorithm for losses whose bound grows in the number of trials i with a known rate i^{ν}, $\nu < 1/2$. This latter hypothesis does not fit well our situation, as we would like to avoid any restriction on the sequence of problems: a very hard instance can be met first, followed by an easy one. In this sense, the hypothesis of a constant, but unknown, bound is more suited: in GambleTA,

this unknown bound would correspond to the worst performance of the worst time allocator on the sequence of instances which have to be solved. In [6], Cesa-Bianchi et al. also introduce an algorithm for loss games with partial information (EXP3LIGHT), which requires losses to be bound, and is particularly effective when the cumulative loss of the best arm is small. In this subsection we introduce a variation of this algorithm that allows it to deal with an unknown bound on losses. Both algorithms make no statistical assumption about the losses, and can therefore be applied to an arbitrary game, be it stationary, nonstationary, or adversarial.

EXP3LIGHT [6, Sec. 4] is a modified version of the weighted majority algorithm [31], in which the cumulative losses for each arm are obtained through an unbiased estimate.[8] The game is subdivided in a sequence of epochs $r = 0, 1, \ldots$: in each epoch, the probability distribution over the arms is updated at every trial, proportional to $\exp(-\eta_r \tilde{L}_k)$, $\tilde{L}_k$ being the current unbiased estimate of the cumulative loss for arm k. Assuming an upper bound 4^r on the smallest loss estimate, $\min_k\{\tilde{L}_k\} \leq 4^r$, η_r is set as:

$$\eta_r = \sqrt{\frac{2(\log K + K \log M)}{(K 4^r)}}. \tag{11}$$

When the bound 4^r is trespassed, a new epoch starts, and r and η_r are updated accordingly.

Algorithm 3 EXP3LIGHT (K, M).

K arms, M trials;
losses $l_k(i) \in [0, 1] \; \forall \, i = 1, ..., M, k = 1, \ldots, K$;
initialize epoch $r := 0$, $L_E(0) := 0$, $\tilde{L}_k(0) := 0$ for $k = 1, \ldots, K$;
initialize η_r according to (11).
for each trial $i = 1, ..., M$ **do**
  set $p_k(i) \propto \exp(-\eta_r \tilde{L}_k(i-1))$, $\sum_{k=1}^{K} p_k(i) = 1$;
  pick arm $I(i) = k$ with probability $p_k(i)$;
  incur loss $l_E(i) := l_{I(i)}(i)$;
  evaluate unbiased loss estimates:
  $\tilde{l}_{I(i)}(i) := l_{I(i)}(i)/p_{I(i)}(i)$, $\tilde{l}_k(i) := 0$ for $k \neq I(i)$;
  update cumulative losses:
  $L_E(i) := L_E(i-1) + l_E(i)$,
  $\tilde{L}_k(i) := \tilde{L}_k(i-1) + \tilde{l}_k(i)$, for $k = 1, \ldots, K$,
  $\tilde{L}^*(i) := min_k\{\tilde{L}_k(i)\}$;
  if $(\tilde{L}^*(i)) > 4^r$ **then**
    start next epoch $r := \lceil \log_4(\tilde{L}^*(i)) \rceil$;
    update η_r according to (11).
  end if
end for

[8] For a given round, and a given arm with loss l and pull probability p, the estimated loss $\tilde{l}$ is l/p if the arm is pulled, 0 otherwise. This estimate is unbiased in the sense that its expected value, with respect to the process extracting the arm to be pulled, equals the actual value of the loss: $E\{\tilde{l}\} = pl/p + (1-p)0 = l$.

Algorithm 3 describes EXP3LIGHT in more detail. Here and in the following, we consider a partial information game with K arms, and M trials; an index (i) labels the value of a quantity at trial $i \in \{1, \ldots, M\}$; k labels quantities related to the k-th arm, $k \in \{1, \ldots, K\}$; index E refers to the loss incurred by the bandit problem solver, and $I(i)$ indicates the arm chosen at trial (i), so it is a discrete random variable with value in $\{1, \ldots, K\}$; index r represents quantities related to the r-th *epoch* of the game, which consists of a sequence of 0 or more consecutive trials; log with no index is the natural logarithm.

Theorem 5 from [6] proves the following bound on the expected regret of EXP3LIGHT, of order $O(K(\sqrt{L^*(M)\log M} + \log M))$, which holds if η_r is updated according to (11):

$$E\{L_E(M)\} - L^*(M) \leq 2\sqrt{2(\log K + K\log M)K(1 + 3L^*(M))} + (2K+1)(1 + \log_4(3M+1)). \tag{12}$$

The original algorithm assumes losses in $[0, 1]$. We first consider a game with a known, finite bound $\mathcal{L}$ on losses, and solve it using EXP3LIGHT, simply dividing all losses by $\mathcal{L}$. Based on (12), it is easy to prove the following:

Theorem 1 Regret of EXP3LIGHT for bounded losses. *Consider a bandit problem with losses $l_k \in [0, \mathcal{L}]$. If $L^*(M)$ is the actual loss of the best arm after M trials, and $L_E(M) = \sum_{i=1}^{M} l_{I(i)}(i)$ is the actual loss of* EXP3LIGHT *(K, M), updated dividing each observed loss by $\mathcal{L}$, the expected value of the regret is bounded as:*

$$E\{L_E(M)\} - L^*(M) \leq 2\sqrt{6\mathcal{L}(\log K + K\log M)KL^*(M)} + \mathcal{L}\left[2\sqrt{2\mathcal{L}(\log K + K\log M)K} + (2K+1)(1 + \log_4(3M+1))\right]. \tag{13}$$

The above bound depends on $\mathcal{L}$ with $O(K(\sqrt{\mathcal{L}L^*(M)\log M} + \mathcal{L}(\sqrt{\mathcal{L}\log M} + \log M)))$.

We now introduce a simple variation of EXP3LIGHT, which does not require the knowledge of the bound $\mathcal{L}$ on losses, and uses EXP3LIGHT (Algorithm 3) as a subroutine. EXP3LIGHT-A (Algorithm 4) is based on the doubling trick used in [6] for a different solver, playing a *full* information game with unknown bound on losses. The game is organized in a sequence of epochs $u = 0, 1, \ldots$, which are not related to the epochs of EXP3LIGHT. A new epoch is started with the appropriate u whenever a loss larger than the current $\mathcal{L}_u$ is observed. In each epoch, EXP3LIGHT is *restarted*[9] using a bound $\mathcal{L}_u = 2^u$.

[9] According to Cesa-Bianchi (2008, personal communication), a bound for the original EXP3LIGHT can be proved for an adaptive η_r (11), replacing the total number of trials M with the current trial i. This should allow for a potentially more efficient variation of EXP3LIGHT-A, in which EXP3LIGHT is not restarted at each epoch, and can retain the information on past losses. However, in practice we observed only a few restarts during the initial portion of the sequence: therefore, we do not expect a dramatic performance improvement from such variation.

Algorithm 4 EXP3LIGHT-A (K, M) A solver for bandit problems with partial information and an *unknown* (but finite) bound on losses.

K arms, M trials
losses $l_j(i) \in [0, \mathcal{L}] \; \forall \, i = 1, ..., M, j = 1, \ldots, K$
unknown $\mathcal{L} < \infty$
initialize epoch $u = 0$, EXP3LIGHT (K, M)
for each trial $i = 1, ..., M$ **do**
  pick arm $I(i) = j$ with probability $p_j(i)$ from EXP3LIGHT
  incur loss $l_E(i) = l_{I(i)}(i)$
  if $l_{I(i)}(i) > 2^u$ **then**
    start next epoch $u = \lceil \log_2 l_{I(i)}(i) \rceil$
    restart EXP3LIGHT $(K, M - i)$
  else
    update EXP3LIGHT with loss $(l_{I(i)}(i)/2^u)$ for arm $I(i)$
  end if
end for

A bound for EXP3LIGHT-A can be derived from (13):

Theorem 2 Regret of EXP3LIGHT-A. *If $L^*(M)$ is the loss of the best arm after M trials, and $\mathcal{L} < \infty$ is the actual, unknown bound on losses, the expected value of the regret of* EXP3LIGHT-A (K, M) *is bounded as:*

$$E\{L_E(M)\} - L^*(M) \leq 4\sqrt{3\lceil \log_2 \mathcal{L} \rceil \mathcal{L} (\log K + K \log M) K L^*(M)} + 2\lceil \log_2 \mathcal{L} \rceil \mathcal{L} \left[\sqrt{4\mathcal{L}(\log K + K \log M) K} + (2K + 1)(1 + \log_4(3M + 1)) + 2 \right]. \quad (14)$$

The proof is given in the Appendix. The regret obtained by EXP3LIGHT-A is $O(K(\sqrt{\mathcal{L} L^*(M) \log \mathcal{L} \log M} + \mathcal{L} \log \mathcal{L}(\sqrt{\mathcal{L} \log M} + \log M)))$: comparing with the regret of the original EXP3LIGHT with a known maximum loss $\mathcal{L}$ (13), we can see that the price of not knowing in advance the value $\mathcal{L}$ is a multiplicative factor $\log \mathcal{L}$. EXP3LIGHT-A can be useful when $\mathcal{L}$ is high, but L^* is relatively small, as we expect in our time allocation setting if the algorithms exhibit huge variations in runtime, but at least one of the TAs eventually converges to a good performance. We can then use it as a BPS for selecting among different time allocators in GAMBLETA.

6 Experiments

In this section we present experiments with runtime data[10] from several recent solver competitions, the same used in [45], and compare GAMBLETA with their offline [47]

[10] While all data is available online, we obtained it directly from Matt Streeter, who kindly saved us the time consuming task of formatting it.

and online [46] greedy allocators (Section 2.3), respectively labeled OffG-Oracle and OnG-Exp3 in the following. The full set of results is available in [13]: here, we limit to the most relevant competitions, in terms of number of instances.

In the actual competitions, each contestant had to solve the same sequence of problem instances: for practical reasons, the runtime of each algorithm on each instance was limited to a maximum "timeout" value. Following the same approach of [45], we simulate the execution of GambleTA based on the runtime values recorded during the competition, using the set of contestants as the algorithm set, and discarding instances which none of the algorithms could solve before timeout; and we evaluate the performance of an allocator reporting the number of instances solved before timeout, the total runtime T (including timeouts), and the speed-up over the Winner of the competition,[11] defined as

$$\mathrm{SU} = \frac{\mathrm{T}_W}{\mathrm{T}}, \tag{15}$$

where T_W is the total runtimes spent by the Winner. In addition, we also report the performances of the Uniform time allocator $\mathbf{s}_{\mathcal{U}} = (1/N, ..., 1/N)$ alone; and the one of an ideal Oracle, with foresight of the runtime values, which only executes, for each problem instance, the algorithm that will be fastest, thus achieving the best possible performance. For each allocator, we also report the *overhead* over the total runtime of the oracle $\mathrm{T}_O = \sum_n \min_n \{t_n(m)\}$,

$$\mathrm{OVH} = \frac{\mathrm{T} - \mathrm{T}_O}{\mathrm{T}_O}, \tag{16}$$

measured regardless of the number of instances solved. This quantity is 0 for Oracle, and up to N for Uniform, if this solves all instances. As GambleTA is randomized (in its BPS component), and its performance may depend on the order of the instances, we repeated each experiments 20 times, each time with a different random reordering of the instances, and a different random seed for the BPS. We will report the results of each run in graphical form, and 95% confidence bounds[12] in tabular form. We will also report the number of runs on which GambleTA is worse than Uniform (WTU), and worse than OnG-Exp3 (WTG).

All experiments were performed in Matlab; the runtimes reported do not take into account the additional computation performed by GambleTA (e.g. updating the RTD models, evaluating the share), again for a coherent comparison with [45], but also because our implementation is far from efficient.

GambleTA is a framework for time allocation, rather than an actual allocator. Before presenting the results, in the next subsection we describe the precise settings adopted in the experiments.

[11] As in [45], we consider as Winner the algorithm which solves most instances, breaking ties based on time. Note that this is not necessarily the criterion that was used in the actual competition.

[12] Evaluated based the Z distribution. Upper confidence bounds will be reported for quantities which we want to minimize, as the total runtime T ant the overhead ovh; lower bounds for those which should be maximized, as the number of solved instances, and the speedup SU.

6.1 Settings

All experiments were performed using the same settings. As BPS, we chose Exp3Light-A (Algorithm 4). All the allocators described in Section 3 were employed, in their dynamic version (Algorithm 1). More precisely, the set of allocators $\mathcal{T}$ consisted of

1. The uniform allocator $\mathrm{TA}_{\mathcal{U}}$.
2. The expected time allocator TA_{Et}.
3. A quantile allocator TA_{Q}, with $\alpha = 0.25$.
4. A contract allocator TA_{C}, with a dynamic $t_c = \tau_k$
5. Our version (10) of the greedy task switching allocator from [47], labeled TA_{Gr}.

The uniform allocator is an obliged choice: it allows to limit the cost of the initial instances, and guarantees that in the worst case the performance of Uniform will be attained, which is often already good in practice, as we will see in the experiments. The remaining allocators are all based on models of the RTDs of the algorithms. When an instance is solved, the models of all algorithms are updated, obviously censoring the runtimes of the unsuccessful algorithms.

While instance features could be obtained online for some competitions, they were not available for most of them. Streeter [45] also presents experiments where the names of directories containing the instances are used as discrete features. As it is not always clear which directories were used, we will compare instead with the per set version of their method, presenting experiments where no features are used: to model the RTD, we simply used the Kaplan-Meier estimator [27].

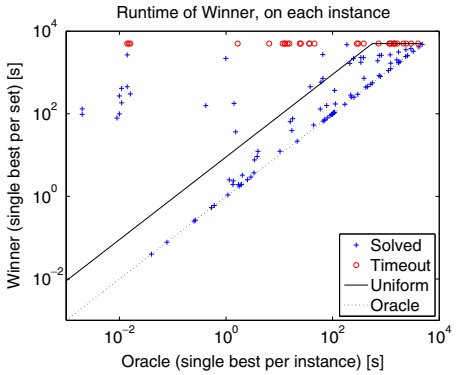
For TA_{Et}, TA_{Q}, and TA_{C}, the RTD of the portfolio was evaluated as in (2), based on the survival functions of each algorithm, estimated by the models. The

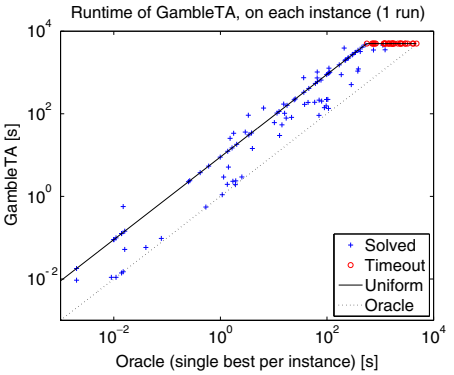
Table 1 Summary of results: instances solved before timeout, by GambleTA and comparison terms

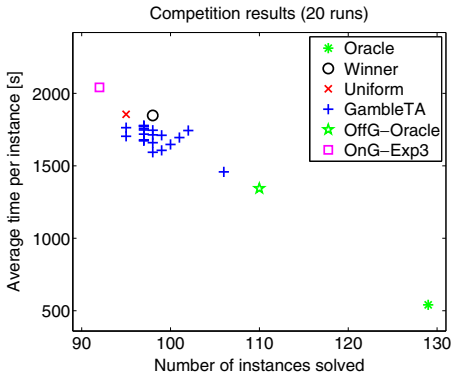
Competition	n. algs.	n. insts.	Winner	Uniform	GambleTA	OnG-Exp3
SAT '07 crafted	9	129	**98**	95	97.3	92
SAT '07 industrial	10	166	**139**	132	132.2	134
SAT '07 random	14	411	257	302	**309.0**	294
SAT '09 application	14	229	**205**	182	189.6	–
SAT '09 crafted	10	187	**156**	153	155.4	–
SAT '09 random	8	547	435	435	**447.4**	–
QBF '07 formal verif.	16	728	621	641	**647.0**	–
QBF '07 HC formulas	16	287	**286**	283	283.8	–
MaxSAT '07 MaxSAT	13	790	**788**	758	773.9	–
PB '07 opt. small ints.	16	396	270	343	**349.0**	–
CP '06 binary ext.	15	1,140	**1,093**	1,068	1,077.6	–

Each line corresponds to a competition, where the competing algorithms had to solve a set of instances

The performance of Oracle is omitted as, by definition, it always coincides with the number of instances. Winner is the a-priori unknown best algorithm in the competition. Uniform is the portfolio of all competing algorithms, sharing resources equally. OnG-Exp3 [46] is another online portfolio approach, starting from scratch as GambleTA. See Figs. 1–11 for detailed results, including average runtimes







	solved	avg. t	SU	OVH
ORACLE	129	540.7	3.42	0
WINNER	98	1847.4	1	2.42
UNIFORM	95	1855.4	1.00	2.43
GAMBLETA	97.3	1725.7	1.07	2.19
OFFG-ORACLE	110	1344	1.37	1.49
ONG-EXP3	92	2041	0.91	2.77

On 20 runs: won 10, WTU 0 WTG 0.

Fig. 1 SAT'07, Hand-crafted, 9 algorithms, 129 instances, timeout 5,000 s. WINNER: `minisatSAT`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA). ONG-EXP3 is the only fair comparison term

share was optimized numerically.[13] The allocation was updated dynamically, as in Algorithm 1, using an exponentially spaced sequence of time intervals $\tau_i = 2^i \tau_1$, with $\tau_1 = 1$ second.

All the allocators default to the static uniform share whenever their computation cannot be carried out for some reason, for example when, after several dynamic updates, the probability of solution is estimated to be 0 for all algorithms. Using a non-parametric method, this is guaranteed to happen as soon as the current runtime is larger than the maximum observation in the sample. Therefore, each TA will eventually allocate a portion of time to each algorithm, satisfying our hypothesis (see p. 13).

Using a non-parametric method, the resulting estimate can be improper, with $F(\infty) < 1$: in such a case, TA_{Et} cannot be evaluated, as the expected time is infinite. As this happened quite frequently with values of $F(\infty)$ very close to 1, we decided to allow for a small "tolerance" ε, evaluating the expected time when $1 - F(\infty) \leq \varepsilon$, and allocating uniformly otherwise. We arbitrarily set $\varepsilon = 0.01$.

[13] Using the Matlab function `fmincon`.

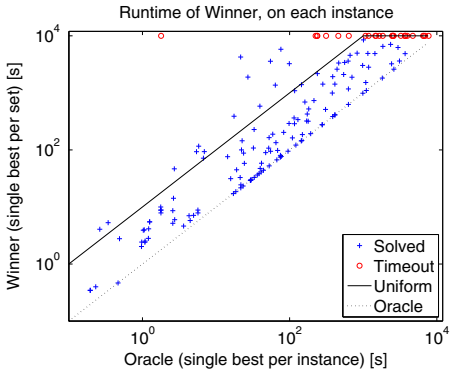
The quantile parameter α of TA_Q was also chosen arbitrarily, based on the observation that high quantiles produce allocations similar to TA_{Et} (see [13, Sec. 6.1.1]). If none of the algorithms reaches the quantile, allocation is uniform.

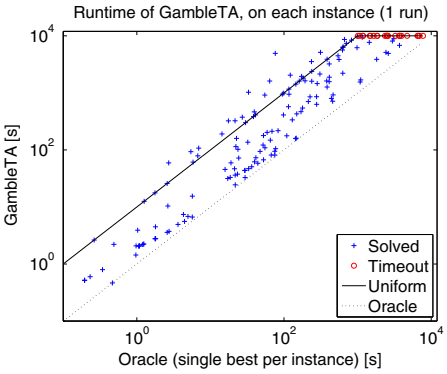
For TA_C, we wanted to avoid fixing a contract time t_c, as this should depend on the range of runtimes observed: we therefore decided to use the time of the next update, $t_c = \tau_i$, $i = 0, 1, \ldots$. In this way, each $(\mathbf{s}_i, \tau_i)$ is such that $S_{\mathcal{A}}(\tau_i; \mathbf{s}_i)$ is minimal. If all algorithms have a $S_n(\tau_i) = 1$, allocation is uniform.

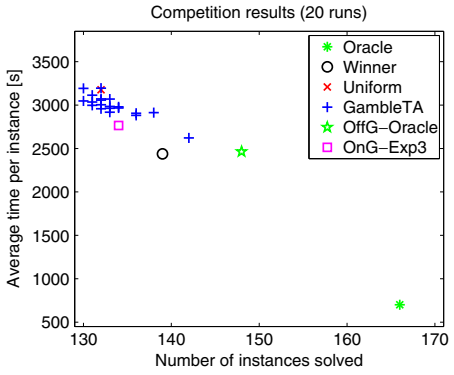
In these allocators, the τ_i are set heuristically. In TA_{Gr}, they are instead set optimally, based on (10). Given the actual RTDs, this allocator is guaranteed to be 4-optimal with respect to the best per set allocation. We used estimates of the RTDs instead, showing experimentally that they already allow to obtain a good performance.

6.2 Plotting results

A summary of results is provided in Table 1. Detailed graphical and numerical results for each benchmark are reported in Figs. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11, using the



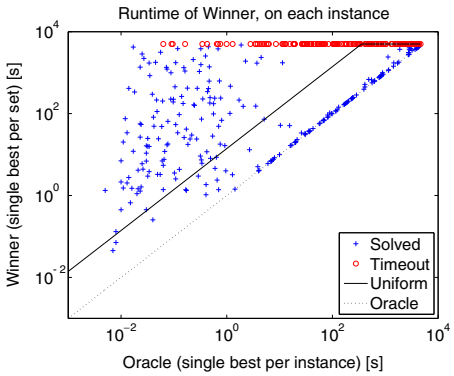


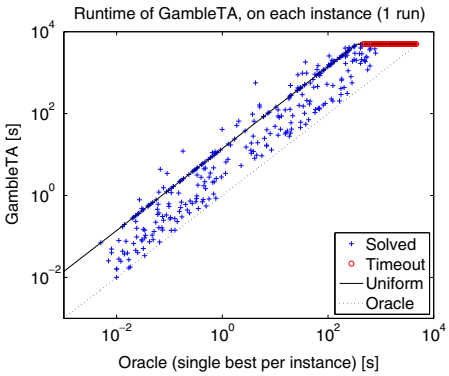


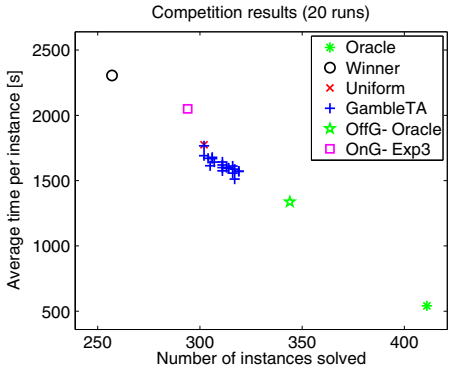
	solved	avg. t	SU	OVH
ORACLE	166	700.3	3.48	0
WINNER	139	2438.3	1	2.48
UNIFORM	132	3175.9	0.77	3.54
GAMBLETA	132.2	3039.1	0.80	3.34
OFFG-ORACLE	148	2464	0.99	2.52
ONG-EXP3	134	2765	0.88	2.95

On 20 runs: won 1, WTU 5 WTG 1.

Fig. 2 SAT'07, Industrial, 10 algorithms, 166 instances, timeout 10,000 s. WINNER: `Rsat`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA). ONG-EXP3 is the only fair comparison term







	solved	avg. t	SU	OVH
ORACLE	411	540.8	4.26	0
WINNER	257	2305.4	1	3.26
UNIFORM	302	1774.9	1.30	2.28
GAMBLETA	309.0	1639.6	1.41	2.03
OFFG-ORACLE	344	1337	1.72	1.47
ONG-EXP3	294	2050	1.12	2.79

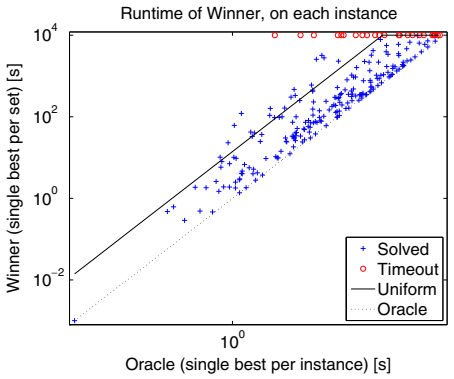
On 20 runs: won 20, WTU 0 WTG 0.

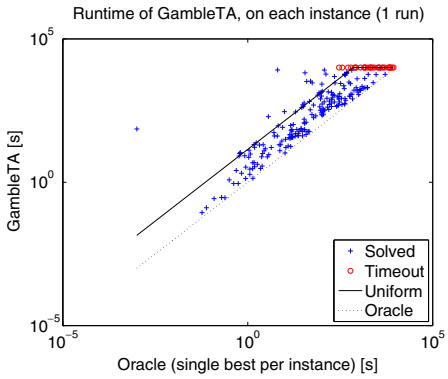
Fig. 3 SAT'07, Random, 14 algorithms, 411 instances, timeout 5,000 s. WINNER: `March KS`. UNIFORM would have won. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA). ONG-EXP3 is the only fair comparison term

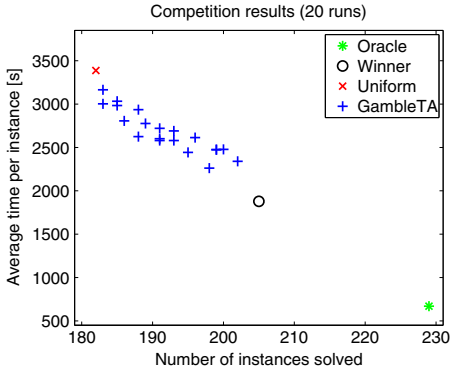
same kind of plots. In the following we describe each plot in detail, giving examples for the SAT 2007 competition, Random category, discussed in the next section.

Log-log comparison with Oracle (see e.g. Fig. 3, top). In these plots, each point corresponds to a single problem instance. On the left one, the runtime of the WINNER on each instance (vertical axis) is compared to that of the ORACLE (horizontal axis). Points off the diagonal correspond to instances where the per set best is outperformed by a different algorithm, while points above the line of UNIFORM corresponds to instances where also a uniform portfolio of all algorithms is faster (i.e., WINNER is WTU). Instances which the WINNER could not solve before the timeout are represented by circles, whose ordinate is the timeout.

Such plots can be used to visualize benchmark characteristics and difficulty. Plots where all points lie on the diagonal represent benchmarks in which a single algorithm dominates all others. The more there are points off the diagonal, and the farther they are, the more the benchmark becomes interesting, as this means that the performances of the algorithms are very diverse, a situation which can be exploited







	solved	avg. t	SU	OVH
ORACLE	229	669.6	2.74	0
WINNER	205	1835.6	1	1.74
UNIFORM	182	3384.2	0.54	4.05
GAMBLETA	189.6	2769.8	0.67	3.14

On 20 runs: won 0, WTU 0.

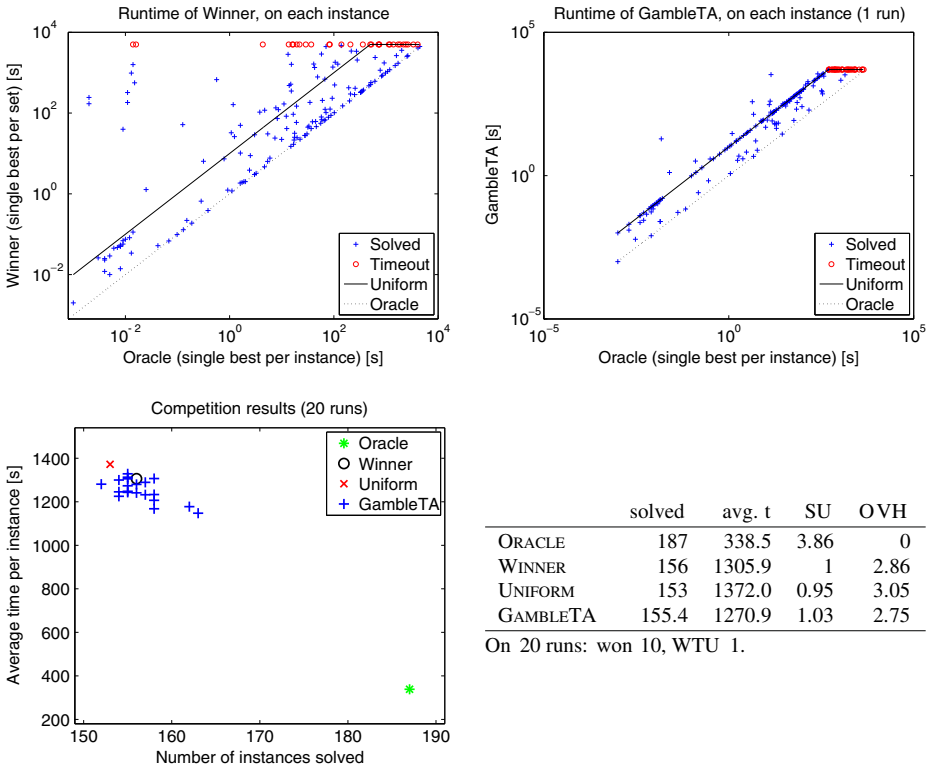
Fig. 4 SAT'09, Application, 14 algorithms, 229 instances, timeout 10,000 s. WINNER: `precosat`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

by a time allocator. In the case reported in Fig. 3, UNIFORM would have won the competition, as one could guess looking at the many points above the UNIFORM line.

We will use the same kind of plots to report the performance of GAMBLETA on each instance: in this case we report results from a single run (with random seed 1), to make the plot readable, and the comparison with WINNER easier. Points which are exactly on the line of UNIFORM are likely to correspond to instances for which the BPS picked the uniform allocator. Note that information about the order with which instances are solved is lost in this kind of plots.

Overall performance (see e.g. Fig. 3, bottom left). The average time T/M vs. the number of instances solved, for each run of GAMBLETA, and for the comparison terms. In these plots, each point corresponds to the whole sequence of instances. The star[14] corresponds to OFFG-ORACLE (8). Note that this method is based on the

[14]For other competitions, only the speedup was reported in [45], therefore we could only recover the average time: in those cases, we represent its performance with an horizontal line.



	solved	avg. t	SU	OVH
ORACLE	187	338.5	3.86	0
WINNER	156	1305.9	1	2.86
UNIFORM	153	1372.0	0.95	3.05
GAMBLETA	155.4	1270.9	1.03	2.75

On 20 runs: won 10, WTU 1.

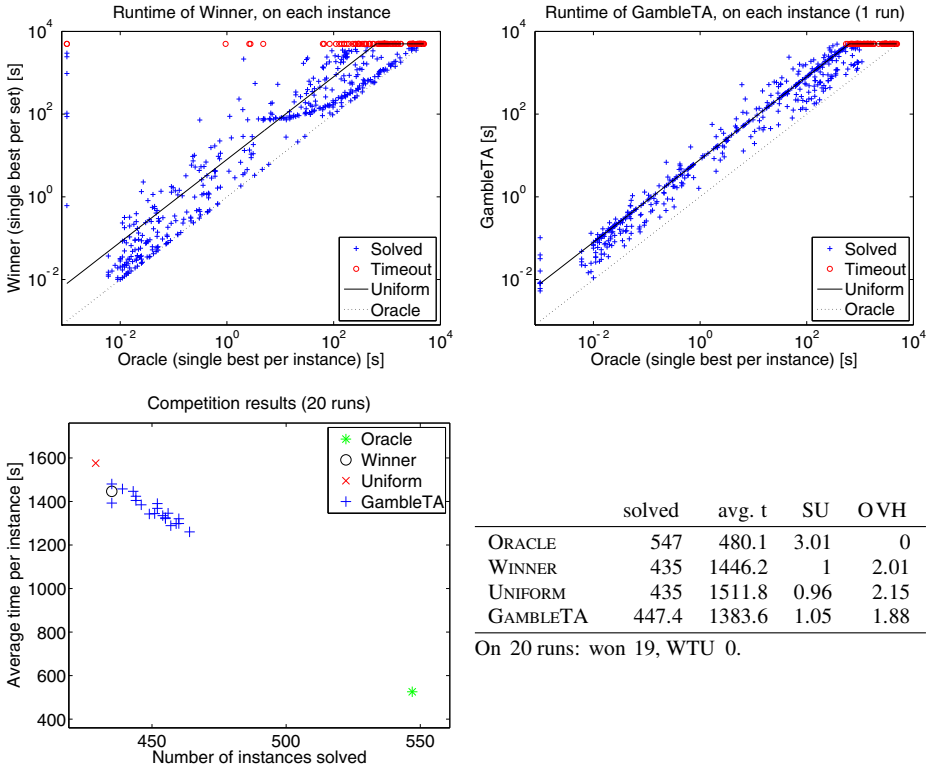
Fig. 5 SAT'09, Crafted, 10 algorithms, 187 instances, timeout 5,000 s. WINNER: `clasp`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

a priori knowledge of algorithm runtimes, so it should rather be seen as the ideal performance of a per set allocator: more precisely, it is proved to be at most 4 times worse than the per set optimal task switching schedule. As the best per instance schedule is always ORACLE, it also gives an idea of the potential gap among per set and per instance allocation, which is more pronounced in benchmarks where algorithm performances vary a lot across instances.

Regarding the competition, these plots allow to appreciate the gap among WINNER and ORACLE, which is often important, as in this case. The relative positions of UNIFORM and WINNER allows to see whether an uniform portfolio of all contestants would have solved more instances than the winner, as in this case.

6.3 SAT solvers competitions

Three of the competitions at the SAT 2007 conference were among SAT solvers, on different categories of instances, both SAT and UNSAT: hand-crafted, industrial, random. For these, we can also compare with the results of another online allocator, ONG-EXP3 [46]. While SATZILLA took part to these competitions, we



	solved	avg. t	SU	OVH
ORACLE	547	480.1	3.01	0
WINNER	435	1446.2	1	2.01
UNIFORM	435	1511.8	0.96	2.15
GAMBLETA	447.4	1383.6	1.05	1.88

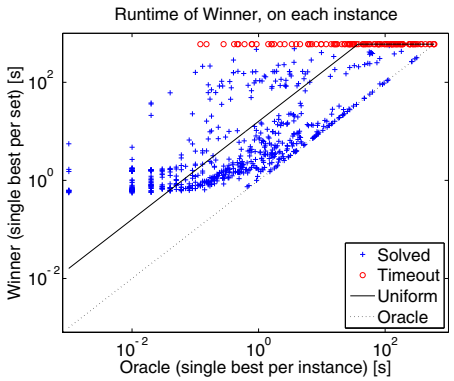
On 20 runs: won 19, WTU 0.

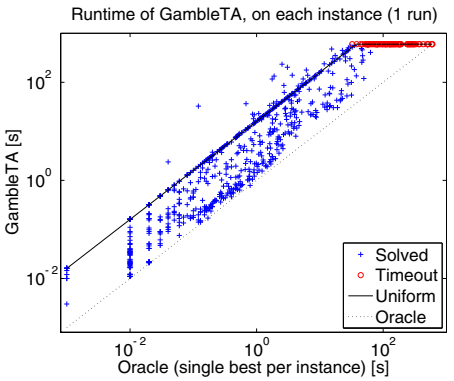
Fig. 6 SAT'09, Random, 8 algorithms, 547 instances, timeout 5,000 s. WINNER: `SATzilla`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

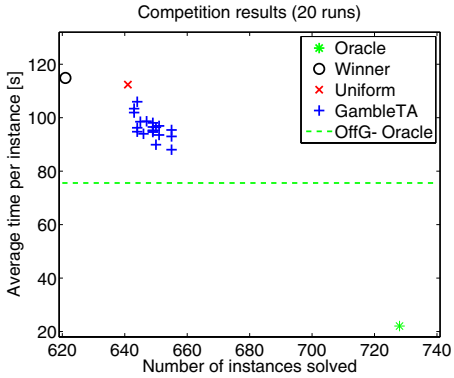
cannot compare this method to GAMBLETA or ONG-EXP3 in terms of algorithm selection performance, as SATZILLA used a different set of algorithms, and it is an offline method. To compare fairly with it, we should use the same algorithm set, and preliminarily solve the same set of training instances.[15]

The results for both GAMBLETA and ONG-EXP3 are obtained on the same data, and both algorithms start from scratch, so in this case the comparison is fair. We also report the results of OFFG-ORACLE [46]: in this case a comparison is obviously not

[15]Moreover, for these and other competitions, we rank solvers based only on the number of instances solved, breaking ties according to the time spent, as in [45]. The actual scoring system used in the 2007 edition was more complex, as it accounted also for which instances were solved: for example it attributed more points for instances which were solved by less contestants. Ours is an obliged choice as it is the way in which the results for our comparison terms were presented, and we do not know their performance on each instance. According to the actual scoring system, in 2007 SATZILLA won the gold medal in both crafted and random categories. The ranking we use corresponds to the one presented here: http://www.cril.univ-artois.fr/SAT07/results/ranking.php?idev=11. The actual scores are available here: http://www.satcompetition.org/.







	solved	avg. t	SU	OVH
ORACLE	728	22.1	5.20	0
WINNER	621	114.9	1	4.20
UNIFORM	641	112.4	1.02	4.09
GAMBLETA	647.0	97.8	1.18	3.43
OFFG-ORACLE	–	76	1.52	2.42

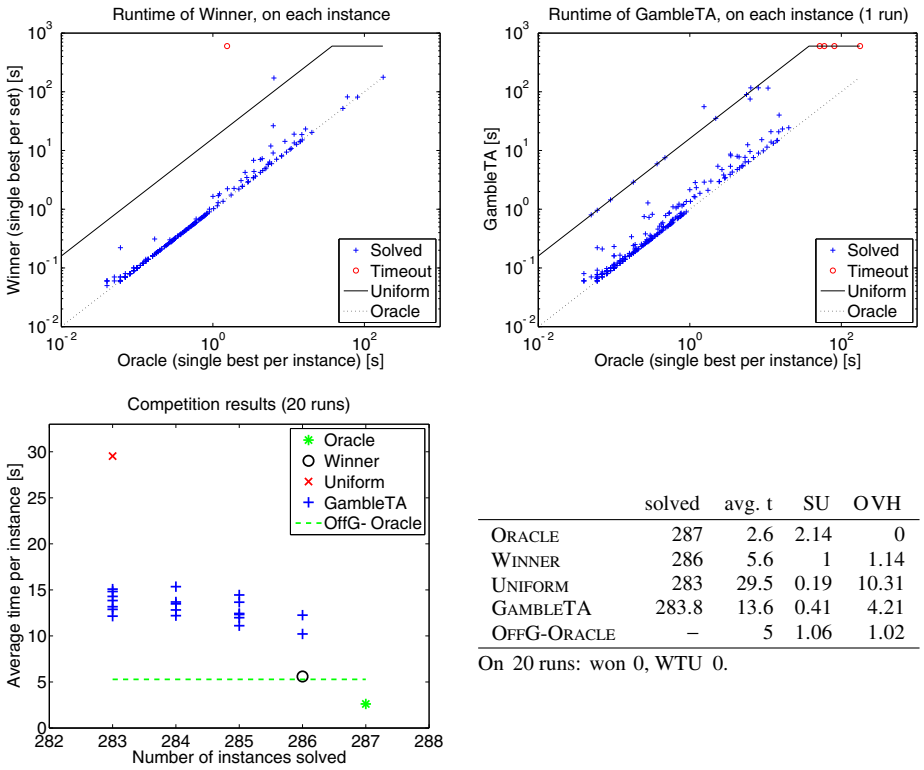
On 20 runs: won 20, WTU 0.

Fig. 7 QBF'07, Formal verification, 16 algorithms, 728 instances, timeout 600 s. WINNER: `AQME-C4.5`. UNIFORM would have won. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

fair, as this allocator is based on prior knowledge of all runtimes. The performance of this allocator may be seen as a 4-approximation of the optimal per set allocation, which is the ideal lower bound also for GAMBLETA, as in this case we did not use any features, so we are also performing per set allocation.

Figure 1 reports results for the hand-crafted category. The problem sequence consisted of artificially created SAT and UNSAT instances, 129 of which could be solved during the competition, with number of variables ranging from 45 to 19,000, and clause-to-variable ratio between 2.67 (underconstrained) and 89 (heavily over-constrained). The winner in this category, `minisatSAT`, could solve 98 instances: its performance is compared to ORACLE in the top-left plot of Fig. 1. This algorithm timed out on 31 instances (represented by the red circles): apart these and a few others instances on which it is WTU, its performance is otherwise similar to ORACLE. This algorithm more or less dominates the scene, together with SATZILLA, which ranked second in this case. The other contestants lag far behind.

A similar situation can be observed in the industrial category (Fig. 2), consisting of 166 hard instances from real industrial applications, mostly hardware verification,



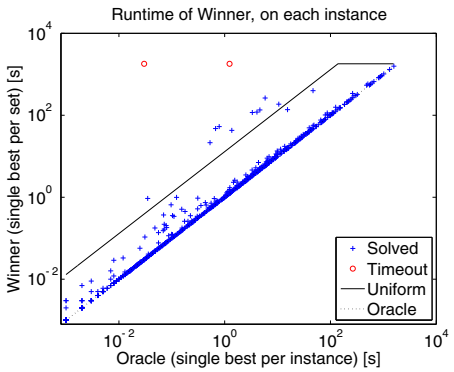
	solved	avg. t	SU	OVH
ORACLE	287	2.6	2.14	0
WINNER	286	5.6	1	1.14
UNIFORM	283	29.5	0.19	10.31
GAMBLETA	283.8	13.6	0.41	4.21
OFFG-ORACLE	–	5	1.06	1.02

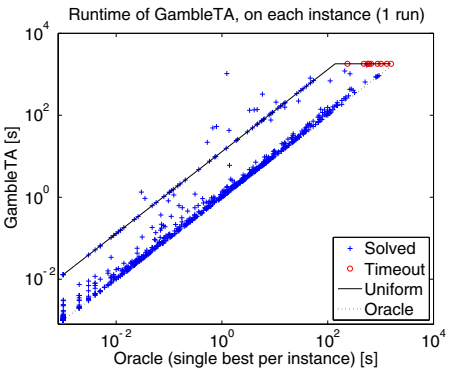
On 20 runs: won 0, WTU 0.

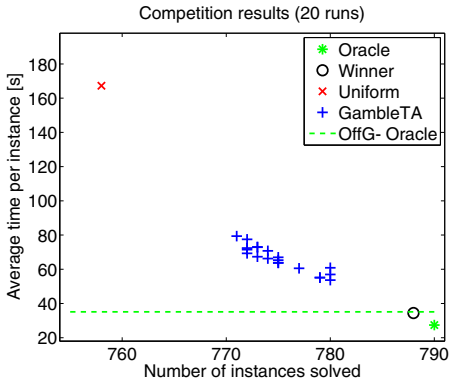
Fig. 8 QBF'07, Horn clause forms., 16 algorithms, 287 instances, timeout 600 s. WINNER: `ncQuBE1.1`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

ranging from 505 to more than 2 millions variables, with ratios between 2.58 and 163. In this case the dominating algorithms are `picosat` and `Rsat`: both solve 139 instances, but the latter is faster. Apart one exception, all instances where the winner times out are hard also for other algorithms who can solve them (see the red circles at the top-right in the plot for WINNER, top left in Fig. 2).

In both categories, the performances of WINNER, UNIFORM, GAMBLETA and ONG-EXP3 are similar. Out of 20 runs, GAMBLETA wins the hand-crafted category 10 times, and it always improves over UNIFORM and ONG-EXP3. The situation is slightly worse in the industrial category: here GAMBLETA wins only on 1 run, which is clearly an outlier (see Fig. 2, bottom-right plot), and it is outperformed by UNIFORM (WTU) and ONG-EXP3 (WTG) on 5 and 16 runs respectively. In both cases, GAMBLETA and WINNER obtain a comparable performance: with such short instance sequences, further improvements are arguably difficult. The situation changes in the random category (Fig. 3), which consists of 411 randomly generated instances, with a number of variables between 45 and 19,000, and ratio between 2.67 and 89. The winner, `March KS`, can only solve 257 instances, and is WTU on many of them (the points above the continuous line in the top-left plot). The improvement obtained by







	solved	avg. t	SU	OVH
ORACLE	790	27.4	1.26	0
WINNER	788	34.4	1	0.26
UNIFORM	758	167.3	0.21	5.10
GAMBLETA	773.9	68.9	0.50	1.51
OFFG-ORACLE	–	35	0.98	0.28

On 20 runs: won 0, WTU 0.

Fig. 9 Max-SAT'07, Max-SAT, 13 algorithms, 790 instances, timeout 1,800 s. WINNER: `maxsatz`. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

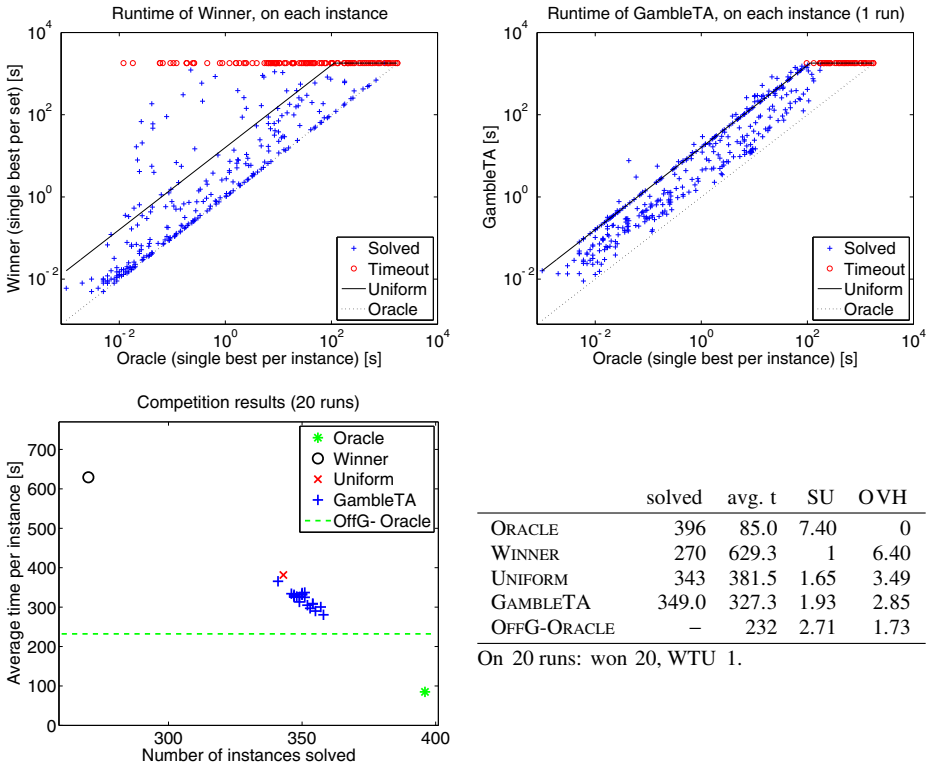
GAMBLETA can be seen already in the log-log comparison with the ORACLE (top-right) where only a few points are WTU. In this case both ONG-EXP3 and UNIFORM would have won the competition, and GAMBLETA manages to do better, solving between 302 and 319 instances.

Figures 4–6 report instead results of the three categories of the 2009 edition[16] of the competition. In that occasion, UNIFORM would not have won in any of the category. On 20 runs, GAMBLETA wins 10 and 19 times in the crafted and random categories, respectively.

6.4 Other competitions

Another competition was held at the SAT 2007 conference among quantified Boolean formulas (QBF) solvers. This problem is a generalization of the SAT

[16] In this edition, the scoring system was simplified to the same criterion we use here, such that the winner was the algorithm which solved the most instances, in the least time.



	solved	avg. t	SU	OVH
ORACLE	396	85.0	7.40	0
WINNER	270	629.3	1	6.40
UNIFORM	343	381.5	1.65	3.49
GAMBLETA	349.0	327.3	1.93	2.85
OFFG-ORACLE	–	232	2.71	1.73

On 20 runs: won 20, WTU 1.

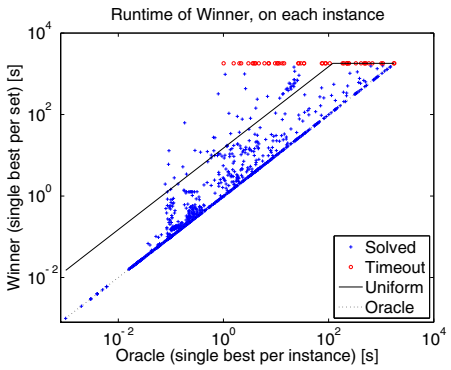
Fig. 10 PB'07, Opt. small ints., 16 algorithms, 396 instances, timeout 1,800 s. WINNER: `bsolo3`.. UNIFORM would have won. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

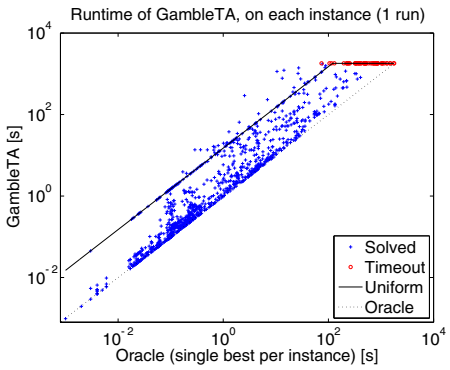
problem, where clauses can be formed using also the operators ∃ and ∀, in addition to the negation. In Figs. 7 and 8 we present results for the two categories of formal verification (728 solved instances) and Horn clause formulas (287 solved instances), respectively: the remaining categories consisted of less than 100 instances.

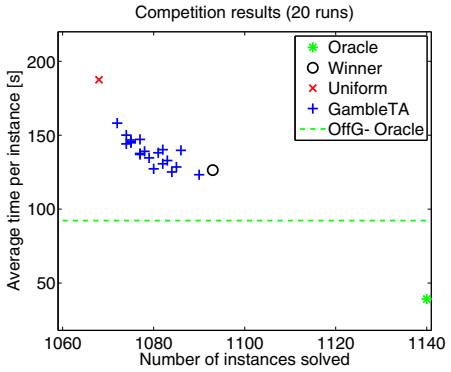
On the formal verification benchmark (Fig. 7), GAMBLETA improves sensibly on the performance of the winner, which does not clearly dominate, and can only solve 621 instances: also UNIFORM would have won in this case, but GAMBLETA is always better. The weighted overhead on the whole sequence is about 0.23.

The Max-SAT'07 competition was also held at the SAT 2007 conference. In this case, the contestants were solving optimization problems: the runtimes reported are the times to find the global optimum, and prove its optimality. Figure 9 reports results for the Max-SAT category of the competition, the one with the most instances. The performance of WINNER is closer to ORACLE, its overhead being 0.25. GAMBLETA has a relatively poor performance, with an overhead of 1.51, but it improves a lot over UNIFORM.

The pseudo-Boolean optimization problem (or zero-one integer programming) consists in minimizing a function of Boolean variables, subject to algebraic







	solved	avg. t	SU	OVH
ORACLE	1140	39.2	3.23	0
WINNER	1093	126.4	1	2.23
UNIFORM	1068	187.5	0.67	3.78
GAMBLETA	1077.6	141.9	0.89	2.62
OFFG-ORACLE	–	92	1.37	1.35

On 20 runs: won 0, WTU 0.

Fig. 11 CPAI'06, Binary ext., 15 algorithms, 1,140 instances, timeout 1,800 s. WINNER: VALCSP3 .. *Top left* Log-log comparison among the runtime of WINNER (*vertical axis*) and ORACLE (*horizontal axis*), on each instance (*plus*, *circle* if timeout). *Top right* Same plot for GAMBLETA (1 run). *Bottom left* Average time vs. instances solved before timeout, for GAMBLETA (20 runs) and comparison terms. *Bottom right* Numerical results (conf. bounds on 20 runs for GAMBLETA)

constraints. The PB'07 track at SAT 2007 consisted of five categories. In Fig. 10 we present results for the largest one, (optimization, small integers, linear constraints). GAMBLETA wins all 20 runs, and also UNIFORM would have won.

The CPAI'06 competition was held at the 2006 conference on Constraint Programming (CP 2006). Figure 11 present results for one of the categories for the Constraint Satisfaction problem. Also in this case, as in Max-SAT, GAMBLETA improves over UNIFORM, but not over WINNER.

6.5 Bandit problem solver performance

In this section we study the behavior of the BPS on some of the competitions.

Figure 12 reports the total number of pulls for each arm, i.e., the number of instances solved by each allocator. As expected, the UNIFORM allocator is used less times, which indicates that the remaining model-based allocators eventually obtain a better performance. Their use varies greatly among different runs, and different benchmarks, but the expected time allocator (Et) is used less often.

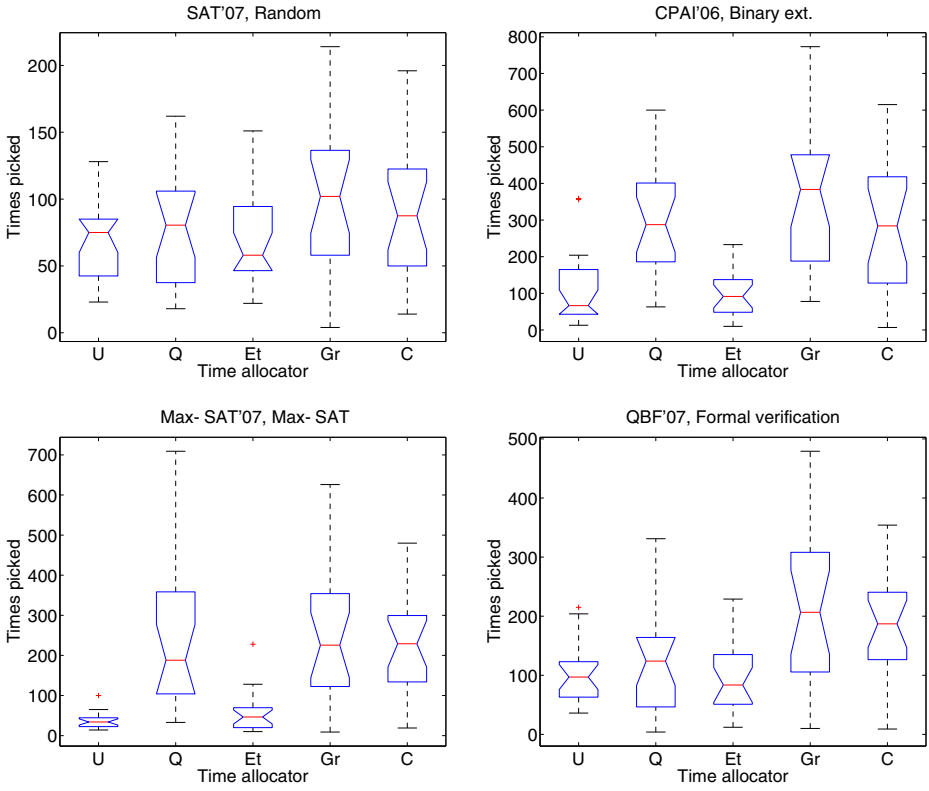


Fig. 12 BPS: Number of pulls for each arm (number of instances solved with each TA, on 20 runs). In this and the following box-plot graphs, the *central (red) line* represents the median, while *box* ends correspond to the upper and lower quartiles. The whiskers extend to a maximum of 1.5 times the interquantile range, and points exceeding this interval are marked as outliers (*red plus signs*). Notches on the sides of the box correspond to a 95% confidence interval on the median

To further investigate the impact of each allocator, in Figs. 13 and 14 we report the performance of GambleTA, in terms of cumulative time, obtained with different TA sets. The label "All" refers to the baseline GambleTA with all 5 allocators (Section 6.1). The following four labels refer to the deletion of a single allocator: "WoQ" stands for "Without Quantile", and so on for Expected Time (Et), Greedy (Gr), Contract (C). The remaining four labels refer to sets of two allocators, where Uniform is combined with a single model-based allocator. The random reordering of the instances was the same for each benchmark. While the performance of the single allocators varies on the different benchmarks, "All" is always competitive with the best single allocator, which confirms that the BPS follows the performance of the best arm.[17]

[17]The fact that sets with more allocators seem to be consistently better is counter-intuitive: it can be understood considering what happens when the BPS selects a suboptimal arm. If there are only two allocators, the suboptimal one will be Uniform; if there are several allocators, as in All, WoQ, etc., it will likely be another model-based allocator, with a better performance.

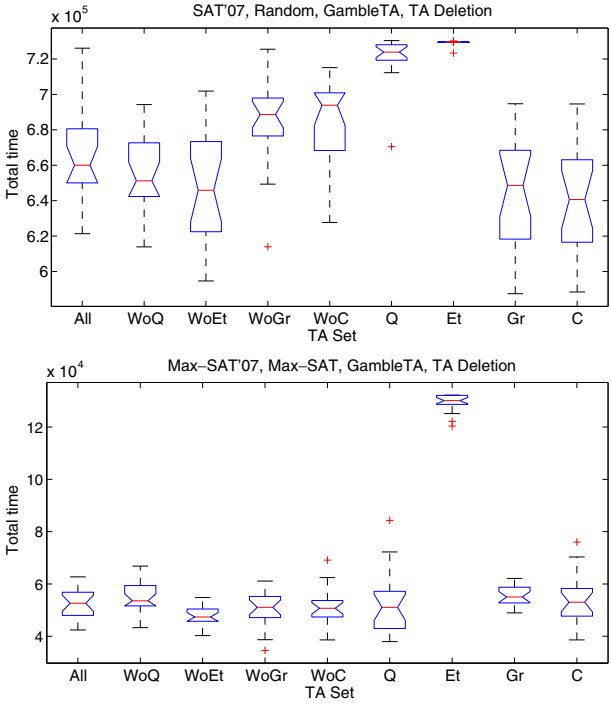


Fig. 13 BPS: Deletion experiments with different TA sets (20 runs). *All* GambleTA with 5 allocators. *WoQ* Without the quantile allocator. *Q* Only with Uniform and quantile allocators. Analogous for expected time (*Et*), greedy (*Gr*) and contract (*C*) allocators

6.6 Discussion

Table 1 summarizes the results of all reported experiments. A first surprising outcome is that Uniform is already competitive with Winner, even better in several cases. This may seem counter-intuitive, as the performance of this trivial portfolio is always N times slower than Oracle: but we have seen that Winner can be several orders of magnitude worse than Oracle on some of the instances, and the competitions where Uniform would have won are indeed those where the variations in performance are more pronounced.

To analyze the results for GambleTA, we can roughly classify these scenarios in two categories, based on whether there is or not a single algorithm whose performance is good enough to almost dominate the others. When this is the case (as with, e.g., the Horn Clause formulas of QBFEVAL'07, Fig. 8, Max-SAT, Fig. 9), the performance of the best per set algorithm, Winner, is close to the performance of the best per instance, Oracle, and there is not much margin for improvement: in these cases GambleTA is worse than Winner, but still comparable. When instead Winner has a poor performance compared to Oracle, there is a great potential performance improvement: this situation is met, for example, in SAT'07 Random (Fig. 3), QBFEVAL'07 Formal Verification (Fig. 7), PB'07 Optimization, small integer constraints (Fig. 10). In these cases, GambleTA improves greatly over Winner.

Moreover, the performance of GambleTA is consistently better than Uniform: GambleTA wins all the competitions where Uniform would have won, and some

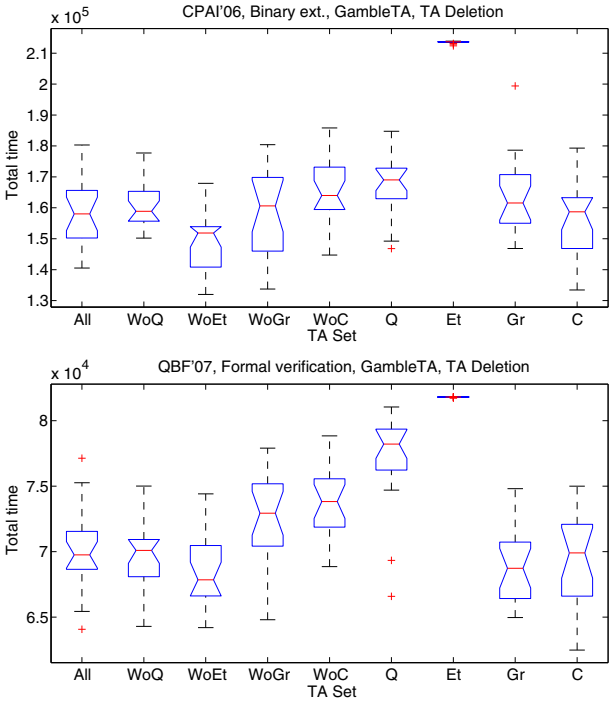


Fig. 14 BPS: Deletion experiments with different TA sets (20 runs). *All* GambleTA with 5 allocators. *WoQ* without the quantile allocator. *Q* only with Uniform and quantile allocators. Analogous for expected time (*Et*), greedy (*Gr*) and contract (*C*) allocators

more. This was expected, as Uniform is one of the arms of the BPS in GambleTA, so the performance of GambleTA cannot be much worse: the fact that it is consistently better is an indirect sign that the model based allocators are improving over Uniform. This improvement was often observed to be surprisingly fast: usually already after 50 instances or so the performance converges to the one observed at the end of the sequence. This seems to suggest that a rough RTD model already allows to allocate time efficiently.

Regarding the single allocators, it seems that none is irreplaceable: their performance varies on the different benchmarks, and GambleTA is quite successful in exploiting the best one, thanks to the bandit problem solver (Section 6.5). The systematic disadvantage of the expected time allocator, visible on most benchmarks, may be due to the use of a non-parametric estimator, which often results in improper RTDs. The greedy allocator, based on [47], has the advantage of being faster to evaluate, as its complexity is $O(N)$, and easier to implement, as it keeps a single algorithm active.

7 Conclusions

We introduced Exp3Light-A, a bandit problem solver for loss games with partial information and an unknown bound on losses, extending the work of [6]. Based on this, we proposed a simpler version of GambleTA [17], a framework for online algorithm portfolio selection.

The problems which we consider are all those problems for which the only performance criterion is solution time, such as decision or search problems (find a solution, or prove that none exists), decision versions of optimization problems (find a solution of given quality), combinatorial optimization (find a solution and prove its optimality). The algorithms can be generalized Las Vegas Algorithms (gLVA) [23], meaning that their runtime is a random variable, possibly infinite. This includes complete algorithms, based on exhaustive search, and incomplete algorithms, based on local search. The requirements for using GambleTA are trivial: that each instance can be solved by at least one of the algorithms, and by all allocators. The latter condition can be easily satisfied when the former holds.

The idea behind GambleTA is to alternate a "default" way of allocating time, not requiring any prior knowledge on algorithm performance, (the Uniform portfolio) with one or more methods which *learn* to allocate time, based on runtimes observed so far. The ratio behind this idea is to exploit predictable regularities in algorithm performance, while reducing the cost of its exploration. The idea is implemented using a bandit problem solver (BPS): the alternative allocators correspond to different arms of the bandit, each instance constitutes a trial, and the time spent solving it represents the loss. Compared to related work, GambleTA can be seen as a practical implementation of model-based portfolios (Section 2.2), where the performance models are learned online, starting from scratch, instead of being available *a priori*. In the terminology used in Section 2, it is therefore an online method, while other properties (static vs. dynamic, per set vs. per instance) depend on the time allocators used.

The BPS used should be able to deal with unbounded losses, as it is difficult to predict a maximum runtime. The use of Exp3Light-A avoids the setting of any additional parameter, and provides a bound on regret with respect to the best time allocator. The choices of the algorithm set, and of the time allocators, are still left to the user. Any existing allocator, model-based or not, can be employed, including alternative algorithm portfolio techniques (Section 2.2, 2.3), or single algorithm selection methods (Section 2.1). In this paper, we used the RTD-based allocators introduced in [17], in their dynamic, per set version, and a simple variation, also based on estimated RTDs, of the greedy allocator from [47].

Several experiments were performed in a variety of hard practical settings, using data from solver competitions in different fields. The performance was quite robust: our method obtains competitive results while starting from scratch. often improving over the best algorithm. We compared our results with those of OffG-Oracle [46], an offline method based on a priori knowledge of the runtimes. For three of the competitions, we could also compare our results with those of another online method, OnG-Exp3 [46], obtaining similar or better results.

The most promising and challenging direction for future research is that of an extension to optimization problems. The most general performance model for optimization algorithms is a bivariate distribution, relating runtime to solution quality. Such distribution can be analyzed considering runtime as a dependent variable, and modeling the *solution quality distribution* (SQD) for an arbitrary runtime value [23]. In statistical terminology, this is an example of *longitudinal data*, which can be described using *mixed effects* models [10]. In [15] we presented preliminary experiments, showing that nonlinear mixed-effects models can be used to predict the performance of optimization algorithms. The issue with such models is their

computational complexity, which scales badly with the size of the sample, rendering their use in time allocation problematic.

Acknowledgements We would like to thank Nicolò Cesa-Bianchi, and Matt Streeter, for useful remarks on their respective work. This research was supported by the Hasler foundation with grant n. 2244, and by the Swiss National Science Foundation (SNF), with grant for prospective researchers n. PBTI22 – 118573.

Appendix: Bound on the regret of Exp3Light-A

In this appendix we prove Theorem 2, following the proof technique employed in [6, Theorem 4]. Be i_u the last trial of epoch u, i. e. the first trial at which a loss $l_{I(i)}(i) > 2^u$ is observed. Write cumulative losses during an epoch u, *excluding* the last trial i_u, as $L^{(u)} = \sum_{i=i_{u-1}+1}^{i_u-1} l(i)$, and let $L^{*(u)} = \min_j \sum_{i=i_{u-1}+1}^{i_u-1} l_j(i)$ indicate the optimal loss for this subset of trials. Be $U = u(M)$ the *a priori* unknown epoch at the last trial. In each epoch u, the bound (13) holds with $\mathcal{L}_u = 2^u$ for all trials except the last one i_u, so noting that $\log(M - i) \leq \log(M)$ we can write:

$$\begin{aligned} E\{L_E^{(u)}\} - L^{*(u)} \leq\ & 2\sqrt{6\mathcal{L}_u(\log K + K\log M)KL^{*(u)}} \\ & + \mathcal{L}_u\left[2\sqrt{2\mathcal{L}_u(\log K + K\log M)K}\right. \\ & \left. + (2K+1)(1+\log_4(3M+1))\right]. \end{aligned} \tag{17}$$

The loss for trial i_u can only be bound by the next value of $\mathcal{L}_u$, evaluated *a posteriori*:

$$E\{l_E(i_u)\} - l^*(i_u) \leq \mathcal{L}_{u+1}, \tag{18}$$

where $l^*(i) = \min_j l_j(i)$ indicates the optimal loss at trial i.

Combining (17,18), and writing $i_{-1} = 0, i_U = M$, we obtain the regret for the whole game:[18]

$$\begin{aligned} E\{L_E(M)\} - \sum_{u=0}^{U} L^{*(u)} - \sum_{u=0}^{U} l^*(i_u) \leq & \sum_{u=0}^{U}\left\{2\sqrt{6\mathcal{L}_u(\log K + K\log M)KL^{*(u)}}\right. \\ & + \mathcal{L}_u\left[2\sqrt{2\mathcal{L}_u(\log K + K\log M)K}\right. \\ & \left.\left. + (2K+1)(1+\log_4(3M+1))\right]\right\} \\ & + \sum_{u=0}^{U}\mathcal{L}_{u+1}. \end{aligned} \tag{19}$$

[18] Note that all cumulative losses are counted from trial $i_{u-1}+1$ to trial $i_u - 1$. If an epoch ends on its first trial, (17) is zero, and (18) holds. Writing $i_U = M$ implies the worst case hypothesis that the bound $\mathcal{L}_U$ is exceeded on the last trial. Epoch numbers u are increasing, but not necessarily consecutive: in this case the terms related to the missing epochs are 0.

The first term on the right hand side of (19) can be bounded using Jensen's inequality

$$\sum_{u=0}^{U} \sqrt{a_u} \leq \sqrt{(U+1)\sum_{u=0}^{U} a_u},$$

with

$$\begin{aligned} a_u &= 24\mathcal{L}_u(\log K + K\log M)KL^{*(u)} \\ &\leq 24\mathcal{L}_{U+1}(\log K + K\log M)KL^{*(u)}. \end{aligned} \tag{20}$$

The other terms do not depend on the optimal losses $L^{*(u)}$, and can also be bounded noting that $\mathcal{L}_u \leq \mathcal{L}_{U+1}$.

We now have to bound the number of epochs U. This can be done noting that the maximum observed loss cannot be larger than the unknown, but finite, bound $\mathcal{L}$, and that

$$U + 1 = \lceil \log_2 max_i l_{I(i)}(i) \rceil \leq \lceil \log_2 \mathcal{L} \rceil, \tag{21}$$

which implies

$$\mathcal{L}_{U+1} = 2^{U+1} \leq 2\mathcal{L}. \tag{22}$$

In this way we can bound the sum

$$\sum_{u=0}^{U} \mathcal{L}_{u+1} \leq \sum_{u=0}^{\lceil \log_2 \mathcal{L} \rceil} 2^u \leq 2^{1+\lceil \log_2 \mathcal{L} \rceil} \leq 4\mathcal{L}. \tag{23}$$

We conclude by noting that

$$\begin{aligned} L^*(M) &= min_i L_i(M) \\ &\geq \sum_{u=0}^{U} L^{*(u)} + \sum_{u=0}^{U} l^*(i_u) \geq \sum_{u=0}^{U} L^{*(u)}. \end{aligned} \tag{24}$$

Inequality (19) then becomes:

$$\begin{aligned} E\{L_E(M)\} - L^*(M) \leq\ & 2\sqrt{6(U+1)\mathcal{L}_{U+1}(\log K + K\log M)KL^*(M)} \\ & + (U+1)\mathcal{L}_{U+1}\Big[2\sqrt{2\mathcal{L}_{U+1}(\log K + K\log M)K} \\ & + (2K+1)(1+\log_4(3M+1))\Big] + 4\mathcal{L}. \end{aligned} \tag{25}$$

Plugging in (21, 22) and rearranging, we obtain (14).

References

1. Allenberg, C., Auer, P., Györfi, L., Ottucsák, G.: Hannan consistency in on-line learning in case of unbounded losses under partial monitoring. In: Balcázar, J.L., Long, P.M., Stephan, F. (eds.) Algorithmic Learning Theory—ALT. LNCS, vol. 4264, pp. 229–243. Springer (2006)

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. Babai, L.: Monte-Carlo Algorithms in Graph Isomorphism Testing. Technical Report 79-10, Univ. de Montréal, Dép. de mathématiques et de statistique (1979)
4. Battiti, R., Brunato, M., Mascia, F.: Reactive search and intelligent optimization. In: Operations Research/Computer Science Interfaces, vol. 45. Springer Verlag, Berlin (2008)
5. Cesa-Bianchi, N., Lugosi, G., Stoltz, G.: Minimizing regret with label efficient prediction. IEEE Trans. Inf. Theory **51**(6), 2152–2162 (2005)
6. Cesa-Bianchi, N., Mansour, Y., Stoltz, G.: Improved second-order bounds for prediction with expert advice. In: Auer, P., Meir, R. (eds.) 18th Annual Conference on Learning Theory—COLT. LNCS, vol. 3559, pp. 217–232. Springer (2005)
7. Cesa-Bianchi, N., Mansour, Y., Stoltz, G.: Improved second-order bounds for prediction with expert advice. Mach. Learn. **66**(2–3), 321–352 (2007)
8. Finkelstein, L., Markovitch, S., Rivlin, E.: Optimal schedules for parallelizing anytime algorithms: the case of independent processes. In: Eighteenth National Conference on Artificial Intelligence—AAAI, pp. 719–724. AAAI Press (2002)
9. Finkelstein, L., Markovitch, S., Rivlin, E.: Optimal schedules for parallelizing anytime algorithms: the case of shared resources. J. Artif. Intell. Res. **19**, 73–138 (2003)
10. Fitzmaurice, G., Davidian, M., Verbeke, G., Molenberghs, G.: Longitudinal Data Analysis. Chapman & Hall/CRC Press (2008)
11. Gagliolo, M., Zhumatiy, V., and Schmidhuber, J.: Adaptive online time allocation to search algorithms. In: Boulicaut, J.F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) Machine Learning: ECML 2004. Proceedings of the 15th European Conference on Machine Learning. LNCS, vol. 3201, pp. 134–143. Springer (2004)
12. Gagliolo, M.: Universal search. Scholarpedia **2**(11), 2575 (2007)
13. Gagliolo, M.: Online dynamic algorithm portfolios. PhD thesis, IDSIA/University of Lugano, Lugano, Switzerland (2010)
14. Gagliolo, M., Legrand, C.: Algorithm survival analysis. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 161–184. Springer, Berlin, Heidelberg (2010)
15. Gagliolo, M., Legrand, C., Birattari, M.: Mixed-effects modeling of optimisation algorithm performance. In: Stützle, T., Birattari, M., Hoos, H.H. (eds.) Engineering Stochastic Local Search Algorithms—SLS. LNCS, vol. 5752, pp. 150–154. Springer (2009)
16. 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—ICANN, Proceedings, Part 2. LNCS, vol. 3697, pp. 7–12. Springer, Berlin (2005)
17. Gagliolo, M., Schmidhuber, J.: Learning dynamic algorithm portfolios. Ann. Math. Artif. Intell. **47**(3–4), 295–328 (2006)
18. Gagliolo, M., Schmidhuber, J.: Learning restart strategies. In: Veloso, M.M. (ed.) Twentieth International Joint Conference on Artificial Intelligence—IJCAI, vol. 1, pp. 792–797. AAAI Press (2007)
19. Gagliolo, M., Schmidhuber, J.: Towards distributed algorithm portfolios. In: Corchado, J.M., et al. (eds.) International Symposium on Distributed Computing and Artificial Intelligence—DCAI. Advances in Soft Computing, vol. 50, pp. 634–643. Springer (2008)
20. Gagliolo, M., Schmidhuber, J.: Algorithm selection as a bandit problem with unbounded losses. In: Blum, C., Battiti, R. (eds.) Learning and Intelligent Optimization. 4th International Conference, LION 4, Venice, Italy, January 18–22 (2010) Selected Papers, Lecture Notes in Computer Science, vol. 6073, pp. 82–96. Springer, Berlin, Heidelberg (2010)
21. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1–2), 43–62 (2001)
22. 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)
23. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Morgan Kaufmann (2004)
24. Horvitz, E.J., Zilberstein, S.: Computational tradeoffs under bounded resources (editorial). Artif. Intell. **126**(1–2), 1–4 (2001)
25. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **27**, 51–53 (1997)
26. 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)

27. Kaplan, E.L., Meyer, P.: Nonparametric estimation from incomplete samples. J. Am. Stat. Assoc. **73**, 457–481 (1958)
28. Klein, J.P., Moeschberger, M.L.: Survival Analysis: Techniques for Censored and Truncated Data, 2nd edn. Springer, Berlin (2003)
29. Kolen, J.F.: Faster learning through a probabilistic approximation algorithm. In: IEEE International Conference on Neural Networks, vol. 1, pp. 449–454 (1988)
30. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In: Van Hentenryck, P. (ed.) ICCP: International Conference on Constraint Programming (CP). LNCS, vol. 2470, pp. 556–572. Springer (2002)
31. Littlestone, N., Warmuth, M.K.: The weighted majority algorithm. Inf. Comput. **108**(2), 212–261 (1994)
32. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993)
33. Mitchell, T.M.: Machine Learning. McGraw-Hill Science/Engineering/Math (1997)
34. Muselli, M., Rabbia, M.: Parallel trials versus single search in supervised learning. In: Simula, O. (ed.) Second International Conference on Artificial Neural Networks—ICANN, pp. 24–28. Elsevier (1991)
35. 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. LNCS, vol. 3258, pp. 438–452. Springer (2004)
36. Petrik, M.: Learning Parallel Portfolios of Algorithms. Master's thesis, Comenius University (2005)
37. Petrik, M.: Statistically Optimal Combination of Algorithms. Presented at SOFSEM (2005)
38. Petrik, M., Zilberstein, S.: Learning parallel portfolios of algorithms. Ann. Math. Artif. Intell. **48**(1–2), 85–106 (2006)
39. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley-Interscience, New York (1994)
40. Rice, J.R.: The algorithm selection problem. In: Rubinoff, M., Yovits, M.C. (eds.) Advances in Computers, vol. 15, pp. 65–118. Academic Press (1976)
41. Robbins, H.: Some aspects of the sequential design of experiments. Bull. Amer. Math. Soc. **58**, 527–535 (1952)
42. Sayag, T., Fine, S., Mansour, Y.: Combining multiple heuristics. In: STACS, pp. 242–253 (2006)
43. Schaul, T., Schmidhuber, J.: Metalearning. Scholarpedia **5**(6), 4650 (2010)
44. Smith-Miles, K.A.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. **41**(1), 1–25 (2008)
45. Streeter, M.: Using Online Algorithms to Solve NP-hard Problems more Efficiently in Practice. PhD thesis, Carnegie Mellon University (2007)
46. Streeter, M., Smith, S.F.: New techniques for algorithm portfolio design. In: McAllester, D.A., Myllymäki, P. (eds.) 24th Conference on Uncertainty in Artificial Intelligence—UAI, pp. 519–527. AUAI Press (2008)
47. Streeter, M.J., Golovin, D., Smith, S.F.: Combining multiple heuristics online. In: Holte, R.C., Howe, A. (eds.) Twenty-second AAAI Conference on Artificial Intelligence, pp. 1197–1203. AAAI Press (2007)
48. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla-07: the design and analysis of an algorithm portfolio for SAT. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming—CP. LNCS, vol. 4741, pp. 712–727. Springer (2007)
49. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)