Online Dynamic Algorithm Portfolios

Minimizing the computational cost of problem solving

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Matteo Gagliolo

under the supervision of
Prof. Jürgen Schmidhuber

March 2010

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Fernando Pedone	Università della Svizzera Italiana, Switzerland
Dr. Mauro Birattari	IRIDIA, ULB, Belgium
Prof. Carla Gomes	Cornell University, USA
Dr. Faustino Gomez	IDSIA, Switzerland

Dissertation accepted on 24 March 2010

Prof. Jürgen Schmidhuber
Research Advisor
IDSIA, Switzerland

Michele Lanza
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Matteo Gagliolo
Lugano, 24 March 2010

“Sul violino ci sono due esse.
Vuol dire: studia sempre.”

Nonno Fedele,
on the necessity of life-long learning.

Abstract

This thesis presents methods for minimizing the computational effort of problem solving. Rather than looking at a particular algorithm, we consider the issue of computational complexity at a higher level, and propose techniques that, given a set of candidate algorithms, of unknown performance, learn to use these algorithms *while* solving a sequence of problem instances, with the aim of solving all instances in a minimum time. An analogous *meta-level* approach to problem solving has been adopted in many different fields, with different aims and terminology. A widely accepted term to describe it is *algorithm selection*. Algorithm *portfolios* represent a more general framework, in which computation time is allocated to a set of algorithms running on one or more processors.

Automating algorithm selection is an old dream of the AI community, which has been brought closer to reality in the last decade. Most available selection techniques are based on a model of algorithm performance, assumed to be available, or learned during a separate *offline* training sequence, which is often prohibitively expensive. The model is used to perform a *static* allocation of resources, with no feedback from the actual execution of the algorithms. There is a trade-off between the performance of model-based selection, and the cost of learning the model. In this thesis, we formulate this trade-off as a bandit problem.

We propose GAMBLETA, a fully dynamic and online algorithm portfolio selection technique, with no separate training phase: all candidate algorithms are run in parallel, while a model incrementally learns their runtime distributions. A redundant set of *time allocators* uses the partially trained model to optimize machine time shares for the algorithms, in order to minimize runtime. A bandit problem solver picks the allocator to use on each instance, gradually increasing the impact of the best time allocators as the model improves. A similar approach is adopted for learning restart strategies online (GAMBLER). In both cases, the runtime distributions are modeled using survival analysis techniques; unsuccessful runs are correctly considered as censored runtime observations, allowing to save further computation time.

The methods proposed are validated with several experiments, mostly based on data from solver competitions, displaying a robust performance in a variety of settings, and showing that rough performance models already allow to allocate resources efficiently, reducing the risk of wasting computation time.

Acknowledgements

The research presented in this thesis was carried out at the "Dalle Molle" Institute for Artificial Intelligence Studies (IDSIA), Lugano, under the attentive yet liberal supervision of Jürgen Schmidhuber, to whom I would like to express my gratitude first. Funding was provided by the Swiss National Science Foundation (SNF), under grants "*General methods for search and reinforcement learning*" (200020-100259), "*General methods for search and Kolmogorov complexity*" (200020-107590/1), "*Survival Analysis Methods for Optimisation Algorithm Selection*" (PBTI22-118573); and by the Hasler Foundation, with the grant "*Distributed algorithm portfolios*" (2244).

Working at IDSIA was a thoroughly enriching experience. Many fellow "*idsiani*" kindly helped me along the way, discussing my ideas and providing useful advice: at the risk of forgetting some, I should mention at least Christoph Ambuehl, Bram Bakker, Leonora Bianchi, Norman Casagrande, Alexey Chernov, Gianni Di Caro, Frederick Ducatelle, Douglas Eck, Alex Graves, David Huber, Shane Legg, Nikos Mutsanas, Jan Poland, Paola Rancoita, Daniil Ryabko, Tom Schaul, Ola Svensson, Daan Wierstra, Viktor Zhumatiy. Faustino Gomez was a constant reference, and did his best to improve my writing skills. Our secretary Cinzia Daldini solved all kinds of practical problem instances. I would also like to thank the fellow students, researchers and personnel of the Faculty of Informatics of the *Universitá della Svizzera Italiana* (USI).

I had several occasions of exchanging comments with other students and researchers working on related topics, as Roberto Battiti, Tom Carchrae, Marco Chiarandini, Carla Gomes, Youssef Hamadi, Holger Hoos, Frank Hutter, Yannet Interian, Marek Petrik, Laura Spierdijk. Nicoló Cesa-Bianchi communicated an unpublished proof which was essential for my work. Kevin Leyton-Brown kindly assisted with the WDP data. Matt Streeter provided the formatted data from solver competitions used in his thesis, and a number of useful insights on his work. I especially acknowledge the patient and careful work of the members of the Committee, and all anonymous reviewers of my papers.

Before the end of my PhD, thanks to an SNF grant, I spent one year as a visiting researcher at IRIDIA (*Université Libre de Bruxelles*), invited by Marco Dorigo; and could cooperate with the Statistics Institute of the *Université catholique de Louvain* (UCL), invited by Ingrid Van Keilegom. At IRIDIA, I found an equally warm and enriching environment, and could interact with, among others, Prasanna Balaprakash, Mauro Birattari, Eliseo Ferrante, Marco Montes de Oca, Thomas Stützle. Catherine Legrand (UCL) proved a precious and reliable advisor for survival analysis.

I take this opportunity to thank again my Master thesis supervisor Davide Anguita (DIBE, *Università degli studi di Genova*), who had a role in my decision of pursuing a PhD, and all those who contributed to my earlier education, in particular Sandro Ridella and Rodolfo Zunino (DIBE); and, earlier on, Nino Podestà and Rosanna Rinolfi (*Liceo Scientifico "G. Bruno"*, Albenga).

To conclude, I would like to thank my parents, and all the friends and relatives who have been there all these years, near or far, you all know who you are.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

The idea of time allocation

The idea of automating the use of a set of algorithms is quite old, and has been proposed in many areas, with various aims and different approaches: typical examples include the selection of one of the algorithms, or the combined execution of many, sequentially or in parallel. The general aim is to obtain a composite problem solving method which improves over the performance of each algorithm in the set. A common aspect of all such approaches is that, when implemented in practice, they involve some form of allocation of computation time to the component algorithms: henceforth, we will refer to the general idea with the term *time allocation*.

In this introductory chapter we motivate research on time allocation (Section 1.1), summarizing the current state of the art (Section 1.2), and underlining its potential with a practical example (Section 1.3). Section 1.4 describes our research goals. Section 1.5 summarizes our contributions, and Section 1.6 outlines the contents of the thesis.

1.1 Motivation

Many important practical problems can be represented mathematically, and solved automatically. Algorithms for solving such problems have been object of study for a long time, but the advent of computing machines has brought about an explosion of research devoted to the design of more efficient solvers. Moreover, important problems are being formalized in the fields of life sciences, medicine, chemistry, economics, etc., with the aim of automating their solution.

The computational power of processors keeps growing at an exponential rate, according to the "self-fulfilling prophecy" of Moore's Law [Mollick, 2006]: unfortunately, many problems of practical importance display a combinatorial explosion of computational complexity [Garey and Johnson, 1990; Papadimitriou and Steiglitz, 1998]. Roughly speaking, this means that if a given machine takes one hour to solve a problem of size n, a twice as powerful machine will use the same time to solve a problem of size $n + 1$ only. Therefore, the computational cost of such problems will *always* represent an obstacle, no matter how powerful the machines of tomorrow will be. This issue constitutes the core motivation of a large corpus of research, producing several novel algorithms every year, with the aim of improving performance on hard problems.

It is often observed in practice that there is no single "best" algorithm: instead, different algorithms perform better or worse on different problem instances. This fact has even received a theoretical confirmation, albeit in a worst-case setting, in the "no free lunch" theorems of Wolpert and Macready [1997, 1995].

Such performance variation represents both a challenge and an opportunity. The challenge is to find the "right" algorithm for solving a given problem instance, and the opportunity is offered by the possibility of combining the execution of several algorithms, improving over the performance of each single one. In practice, such combination consists of allocating computation time to the different algorithms. In the following we will therefore use the general term *time allocator* to refer to any method which solves problems using a set of algorithms.

Given a set of algorithms, prior knowledge of the mapping between each problem instance and the corresponding best algorithm would allow for the best possible performance. *Learning* and performing such mapping automatically is the object of *algorithm selection*, now a thirty-year-old branch of artificial intelligence [Rice, 1976]. The algorithm selection problem can be formulated as follows: given a particular problem instance, and a set of alternative solvers, which solver should we use to get the best result? This question immediately raises a further one: what does "best" mean? The answer depends on the problem being considered.

In the broad class of *decision*, or *search* problems, the aim of problem solving is just to *find* a solution, and different algorithms may simply be compared based on the time they spend in doing so, so in this context "best" means "fastest". In the more general class of *optimization* problems, each solution is also characterized by a measure of its quality: the notion of algorithm performance may simply coincide with this measure, or involve some more complex trade-off between solution quality and time. If a target quality value is given, an optimization problem can be reduced to a search problem: finding a solution whose quality reaches the target. In certain cases it is also possible to prove that a given solution is the *global optimum*, i.e. it cannot be further improved. In both cases, two algorithms may be compared based on the time spent in finding a solution with the desired quality level. Also in this context, then, "best" may simply mean "fastest". Indeed, much of the research on time allocation is focused on the task of *minimizing the time* spent in solving problems. We will therefore consider time allocation to *Las Vegas* algorithms (LVA) [Babai, 1979], i.e. algorithms whose performance on a single instance coincides with their runtime, which is in general a random variable. More precisely we will consider *generalized* LVAs [Hoos and Stützle, 2004], whose runtime can be infinite.

1.2 State of the art

Algorithm selection can be performed once for a whole set of problem instances (*per set* selection [Hutter and Hamadi, 2005]), or repeated independently for each instance (*per instance* selection). The latter alternative 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 simple idea was proposed already by Rice [1976], and was later adopted by several authors. Practical implementations are 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 on this sample, a predictive model of performance is learned mapping (*instance*,*algorithm*) pairs to the expected performance: in practice, this can be implemented 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 single algorithm expected to obtain the best performance is selected, and used to solve the instance. For Las Vegas algorithms, a particularly successful implementation of this idea was proposed by Leyton-Brown et al. [2002], and later improved by Nudelman

et al. [2004]; Xu et al. [2008]. In this case, the runtime is predicted, and minimized.

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* [Huberman et al., 1997; Gomes and Selman, 2001]. 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 parallel, as in a portfolio, or periodically restarting the algorithm with a different random seed, with a *restart strategy* [Luby et al., 1993]. 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 continually receives a constant share of computation time [Huberman et al., 1997; Gomes and Selman, 2001]; or dynamic, e.g. according to a *task switching* schedule, specifying which algorithm is active at a given time Finkelstein et al. [2003]. The problem of estimating the RTDs is not tackled, so this line of work remains at a theoretical level.

Recently, some authors proposed practical methods to evaluate portfolios on a per set basis, based directly on a runtime sample, without explicitly modeling the RTDs [Petrik and Zilberstein, 2006; Sayag et al., 2006; Streeter et al., 2007]. Also in this case the runtime sample is collected offline, with the exception of Streeter et al. [2007] who also consider online versions of their methods.

1.3 An example scenario

Both classes of decision and optimization problems contain examples of computationally complex problems, with important practical applications in various fields, such as industry, logistics, medicine, etc. A considerable research effort has therefore been put in devising more competitive algorithms. In recent years, public competitions among solvers have been held, where the contestants compete on one or more sets of problem instances. In most cases, none of the competing algorithms dominates the others in performance: different algorithms attain the best performance on different instances, and it is rarely observed that a single algorithm solves all instances within the maximum time allowed. The results of such competitions constitute therefore an experimental confirmation of the potential benefit of combining different algorithms, and can be used as time allocation benchmarks [Petrik and Zilberstein, 2006; Streeter et al., 2007].

As an example, in Figure 1.1 we report results from one of these competitions (SAT 2007, Random category), held among solvers of the satisfiability problem [Gent and Walsh, 1999], which consists in finding a bitstring of length n satisfying a given set of conditions. This problem has applications in many important fields, such as software and hardware verification and design, logistics, planning, etc.

In this competition, 14 algorithms had to solve 511 randomly generated problem instances. Each algorithm was allowed a maximum of 5000 seconds on each instance: 411 of these were solved by at least one algorithm within the time limit. In terms of the number of instances solved, the winner in this category was `March KS`, which could solve 257 instances within the time limit, with an average runtime of 2305 seconds[1]. Figure 1.1 reports its runtime (vertical

[1]This measure is used here to compare with the results as reported by Streeter [2007]. The scoring used in the actual competition was more complex, as it took also into account the number of algorithms which could solve each instance. According to these criteria, SATzilla won this competition, even though it solved 248 instances. See

axis) versus the minimum runtime observed on each instance (horizontal axis), limited to the 411 instances which could be solved by at least one algorithm. This algorithm was the best on the set of instances, as it solved the most instances in the shortest time, but on 327 instances it was outperformed by some other contestant. The crosses represent the 257 instances solved by the winner: the crosses along the diagonal correspond to the 84 instances on which it was also the fastest, while the circles corresponds to 154 instances which it could not solve before the timeout. Looking at points far from the diagonal, which represent instances on which the winner was outperformed, you can notice a disadvantage of up to five orders of magnitude.

An "oracle" algorithm selection method, which predicts and runs, independently for each instance, the algorithm which will solve it in the shortest time, would have solved 411 instances within the time limit: its performance would be represented in the plot by points along the diagonal.

The black line above the diagonal reports instead the performance that would have been attained executing all 14 algorithms in parallel, on a single processor, with equal priorities. This simple "uniform" portfolio solves each instance as soon as the fastest algorithm solves it: as the algorithms are sharing computation time equally, its runtime on each instance is exactly 14 times the runtime of the oracle. This trivial time allocator would have outperformed the winner on 224 instances, corresponding to the crosses and circles above the black line. Moreover, it would have solved a larger number of instances, 302, with an average runtime of 1775 seconds, even though it would have timed out on some of the instances solved by `March KS`, those in correspondence of the horizontal portion of the line.

While this situation is rather extreme, it is not at all an exception. At the same competition, the uniform portfolio would have ranked second in another category (Hand-crafted instances), and third in the remaining two. In Sec 9.3 we report results of 43 solver competitions, on different problems, including satisfiability, constraint programming, automated theorem proving, software verification, etc. In no case the winner was also the fastest on each instance: only in three cases it had an overhead of less than 10% over the oracle. In 32 competitions, the overhead was larger than 100%, and the winner timed out on instances which other contestants could solve. In 12 of these, the winner would have been outperformed by a uniform portfolio of all contestants running in parallel.

If a uniform portfolio can already score well in a competition, a more efficient time allocator may be able to further improve the performance, learning the correct mapping among problem instances and the corresponding fastest algorithm. Our main contribution consists of an example of such an allocator, outlined in the next sections.

`http://www.satcompetition.org/2007/` for the full results.

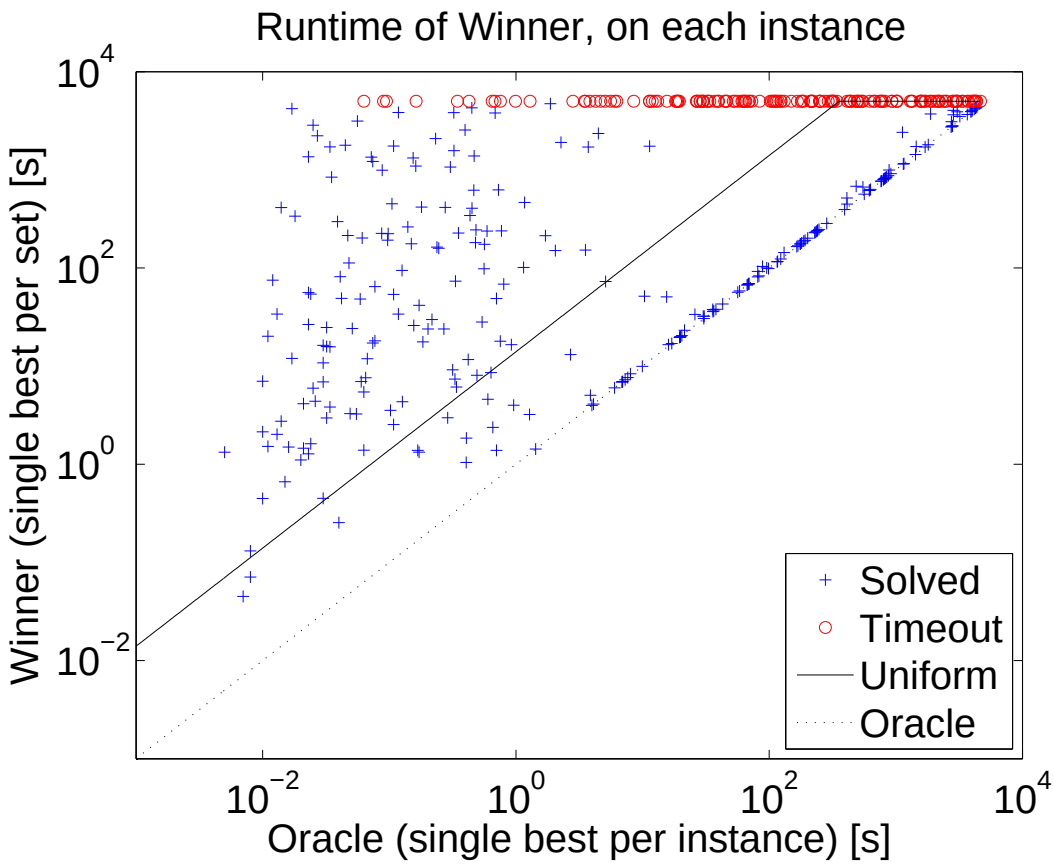


Figure 1.1. Results from the SAT 2007 competition, Random category. The runtime of the winner (vertical axis) is plotted against the best runtime registered on each instance (horizontal axis). Each point in the graph corresponds to a problem instance: blue plus signs correspond to instances solved by the winner before timeout; Red circles correspond to instances where the winner timed out, some of which are easy for a different algorithm. The best possible time allocation would be attained by an ORACLE with foresight of runtimes, represented by points along the diagonal. A simple UNIFORM portfolio of all contestants would correspond to points along the black line: it would timeout for the hardest instances (horizontal portion of the line), but would solve more instances than the winner.

1.4 Research goals

The main aim of our research has been to develop a general method for allocating computation time to a given set of algorithms, in order to solve a given set of problem instances, using one or more processors. For simplicity, we limited our research to problems with a well defined solution criterion, such as decision problems, with the aim of minimizing the total computation time. This definition includes many computationally expensive problems of utmost practical importance.

Hypotheses were kept to a minimum. The algorithms at disposal are assumed to be generalized Las Vegas [Hoos and Stützle, 2004], meaning that we do not require each instance to be solvable by each algorithm. This allows for the combination of complete and incomplete solvers, which can be more efficient in practice. We adopted a "black box" approach towards the algorithms, avoiding any problem-specific assumptions.

The issues of collecting performance data, and predicting future performance, were explicitly taken into account. Rather than assuming ideal models, or arbitrary amounts of data to be available, we require the system to start from scratch, without any prior knowledge about the performance of the different algorithms, or the difficulty of the various instances. If there are some regularities in algorithm performance, these should be learned and exploited to reduce computation time. The worst case performance should be comparable to the performance of the uniform portfolio, which in turn is a constant factor worse than the best possible performance. In the average case, the system is expected to sensibly improve over the uniform portfolio. The fact of starting from scratch allows to explicitly take into account the computational cost of learning, searching for a balance with its benefit in terms of performance.

1.5 Summary of contributions

This thesis develops and validates two core ideas. The first is that a very rough model of algorithm performance may already allow to perform time allocation efficiently. This suggests that model-based allocation can be made more efficient and practical, by reducing the time needed to sample algorithm performance: for example, adopting an *online* approach, where the model is iteratively updated based on the runtime spent for solving an instance, and used to speed up the solution of the following one.

The other idea is that a *dynamic* time allocation, i.e. an allocation which is allowed to change during the solution of a single instance, can offer an advantage over a *static* allocation, where the priorities of the algorithms are set before starting them, and kept constant until the instance is solved.

For ease of exposition we will start from the latter idea. Our time allocators are algorithm portfolios: the algorithms are executed in parallel, without interacting, and their execution is halted as soon as one of them solves the current instance. In this case, time allocation consists in assigning a portion of computation time to each algorithm. These portions are evaluated based on the runtime distributions of the algorithms for the instance being solved, so the evaluation is repeated for each instance. This approach has the advantage of being more general than single algorithm selection, as the allocation evaluated may assign computation time to more than one algorithm, when desirable, and it is based on the whole distribution of runtime, instead of the mere expected value. It can also deal with a set of generalized Las Vegas algorithms, for which single algorithm selection would be impractical, as not all algorithms are guaranteed to find a solution.

The evaluation of the allocated portions of computation time is formulated as an optimization problem, according to different criteria, e.g. minimizing the expected runtime of the portfolio. This static allocation can be made dynamic by simply updating it periodically. The advantage of dynamic allocation can be shown experimentally, and has been proved formally by Sayag et al. [2006]. The extension to multiple processors consists in correcting the quantity being optimized, but this has the disadvantage of increasing the search space of possible allocations. For two of the criteria, we prove that it is possible to reduce the size of the search space where the optimal allocation resides.

Implementing the time allocators in practice requires predicting the runtime distributions of the algorithms. Modeling random events in time is the subject of *survival analysis* [Klein and Moeschberger, 2003], a branch of statistics with applications in several fields, ranging from medicine to engineering. The same models used to predict the duration of a human life, a light-bulb, or a mechanic part subject to wearing, can also be used to predict the time an algorithm will take to solve a given instance. In our case the data being sampled is the runtime of the algorithms on different instances. Such data can be used to predict the RTDs of the algorithms on a new instance by conditioning a *regression* RTD model on features of the instance.

A common issue in survival analysis is the incompleteness of the data: for example, if several light-bulbs are observed until the last one fails, their lifetime distribution can be evaluated using classical statistics. If, as more commonly happens, the failures are recorded only over a limited period of time, such that not all light-bulbs fail, only a lower bound will be available for the failure times of the surviving bulbs. Survival analysis methods can correctly take into account such *censored* data. When an algorithm portfolio solves an instance, the runtime is known exactly only for one of the algorithms: the runtimes of the remaining ones can be considered censored observations, and used to update their RTD models. This allows to reduce the computational cost of sampling runtime distributions, avoiding to solve the same problem instance more than once. The same algorithm portfolio can therefore be used to speed up problem solution, and to collect runtime data in an efficient manner.

Time allocation is usually based on a performance model. Given a set of M instances to be solved, we could then solve the first M_0 with a uniform portfolio, or exhaustively with all algorithms, in order to collect a runtime sample, and learn a model, based on which we could then allocate time for the remaining $M - M_0$ instances. This *offline* approach requires setting M_0. Intuitively, this poses an *exploration* vs. *exploitation* dilemma: collecting more data is computationally expensive, but can improve the model, and make the resulting allocation more efficient. Moreover, it requires assuming that the first M_0 instances are representative of the remaining ones. We decided to adopt an *online* approach instead: each solved instance contributes to updating the model. Independently for each instance we decide whether to solve it with a model-based time allocator, or with the uniform portfolio. The decision is taken probabilistically, according to a solver for the *multi-armed bandit* problem [Auer et al., 2002], a well known framework for online learning, in which a discrete set of alternatives is explored during a sequence of trials. This general approach, which we named GAMBLETA, can be applied to any time allocator which is based on a runtime sample, not necessarily via a model, and also allows to select among several alternative allocators, which can be seen as additional arms. In particular, we propose to use a bandit problem solver which can provide a bound on the regret with respect to the best arm, such that, as the number of instances increases, the total runtime spent will converge to the one of the best alternative. In this case, we expect the model-based allocator to eventually outperform the uniform portfolio; if this does not happen, the overall performance will nonetheless be close to that of the uniform portfolio.

We can therefore bound the performance of GAMBLETA with respect to the best time allocator, which in the worst case is the uniform portfolio. In several challenging experiments, the actual performance was found to be sensibly better than predicted by the bound.

1.6 Outline of the thesis

In writing this dissertation, we strove to ease casual reading, making each chapter as self-contained as possible, at the cost of some redundancy. With the exception of this and the last one, each chapter is introduced by a few paragraphs, outlining its contents, and concluded by a discussion section, which summarizes it and relates to the following one. The reader should be able to follow the stream of ideas just by reading these two portions, and use the internal references to selectively delve into the contents.

The thesis is divided into five parts. The first part coincides with this introductory chapter. The second part contains the foundations, subdivided into three chapters. Chapter 2 briefly discusses the problem of computational complexity, motivating research on time allocation, and describes related work in this area, including algorithm selection, algorithm portfolios and restart strategies. Chapter 3 presents some notions of survival analysis, aimed at sampling and modeling runtime distributions. Chapter 4 describes the Multi-Armed Bandit problem, discussing its application to time allocation, and presents related work where this paradigm has been used in this sense.

The third part contains our own contributions to the field, and is also subdivided into three chapters, with a rough one-to-one correspondence with the chapters of Part II. Chapter 5 defines the time allocation problem we intend to address, and presents several ideal portfolios based on runtime distributions, assuming them to be available. The same chapter also describes a general method for turning any static time allocator into a dynamic one, and addresses the problem of allocating multiple processors. The two chapters that follow are aimed at the practical implementation of the proposed time allocators. Chapter 6 addresses the issues of sampling and modeling runtime distributions efficiently, exploiting survival analysis techniques, and discusses their impact on the correctness and efficacy of time allocation. Chapter 7 proposes a general method for performing online time allocation, based on the Multi-Armed Bandit framework, and presents two instantiations of the idea: an online restart strategy (GAMBLER) and an online, dynamic algorithm portfolio (GAMBLETA). In these three chapters, a related work section precedes the closing discussion.

Part IV contains two chapters, reporting results of experiments with GAMBLER and GAMBLETA, respectively. Part V consists of a single chapter, which concludes the thesis summarizing our achievements and discussing directions for future research.

Part II

Foundations

Chapter 2

Time allocation

In this chapter we present related work on time allocation, focusing in particular on the task of minimizing the time cost of problem solving. We begin by informally discussing, in Section 2.1, the issue of computational complexity, pointing out the practical importance of speeding up problem solution. In Section 2.2 we refine the concept of time allocation, and define some terms which will be useful for describing related work, which is subdivided as follows: Section 2.3 describes work where a single algorithm is selected, independently for each problem instance, based on performance models. Section 2.4, describes foundational work on restart strategies and algorithm portfolios, where multiple runs of one or more algorithms are combined in order to reduce the risk of wasting computation time. Section 2.5 describes more recent work where the optimal allocation for a *set* of problem instances is evaluated based on previous experimental results. Section 2.6 describes some heuristic approaches which are not based on performance models. Section 2.7 draws some conclusions and motivates the following chapters.

2.1 A motivation: computational complexity

In this section we recall some basic notions of computational complexity, based on [Hoos and Stützle, 2004; Birattari, 2009], with the sole aim of giving a rough idea of the issue. Many exhaustive texts are available on these topic, e.g. [Garey and Johnson, 1990; Papadimitriou and Steiglitz, 1998].

A problem can be defined as a collection of *instances*, corresponding to functions over elements of a *search space*. In a *search problem*, such functions are binary: each element of the search space is either a solution, or not. A search problem consists in finding one solution, or proving that there is none. A *decision problem* can be defined as a question with a "yes" or "no" answer. For every search problem, an associated decision problem can be formulated: given an instance, say whether a solution exists. For this reason, the two terms are often used indifferently. A simple example of a decision problem is primality testing: given a natural number n, say whether n is prime. The associated search problem is: find a nontrivial factor of n.

In an *optimization* problem, each instance can be represented as an *objective function*, mapping each element of the search space to its *objective value*, a scalar representing solution quality. Such value is to be *optimized*, i.e. either minimized or maximized, depending on the problem. An additional binary function may first distinguish among *feasible* and *unfeasible* solutions. The *search variant* of an optimization problem consists in finding the best possible feasible solution,

i.e. the feasible solution which optimizes the objective. The *evaluation variant* consists in finding the best objective value: for hard problems, this requires solving the search variant. In real-world applications, often the size of the search space makes it impossible to find the optimal solution. In such cases, a stopping criterion has to be specified. Given a threshold on the objective value, a corresponding decision problem can be stated: e.g., for maximization, say whether there exist a feasible solution whose value is larger than the threshold. The corresponding search problem is: find such a solution, or prove that there is none. An example optimization problem is the following: given two natural numbers m, n, find the greatest common divisor. In this case, the set of feasible solutions is the set of all common divisors of m and n; the objective function is simply the value of of a number, which has to be maximized over the set of feasible solutions.

Decision problems can be classified according to their inherent difficulty, independent from the particular algorithm used for solving them. The class of *NP* problems includes all problems for which, given an arbitrary element of the search space, it is possible to determine whether such element is a solution in *polynomial time*, i.e. a time which depends on the size of the problem according to a polynomial formula. This definition alone does not tell anything about the difficulty of finding a solution, which is instead related to two subclasses of NP: class *P*, which contains problems whose solution can also be found in polynomial time, and class *NP-complete*, which contains those NP problems which are at least as hard as any other NP problem, in the sense that an arbitrary NP problem instance can be mapped to an NP-complete problem instance whose solution, once found, can be mapped back to a solution of the original instance, and such *reduction* can be performed in polynomial time. In practice this means that an algorithm for solving one particular NP-complete problem can be efficiently applied to any other NP problem, including other NP-complete problems. For problems in this subclass, no polynomial time solver is known, i.e., the amount of computation required to find a solution is exponential in the dimensionality of the search space.

NP-hard problems are not necessarily contained in NP, but are at least as hard as NP-complete problems, i.e. NP-complete problems can be reduced to NP-hard problems in polynomial time. In *combinatorial* problems, the candidate solutions are combinations of elements of a finite set: for example, orderings, subsets, or assignment of discrete values to a finite number of variables. In these problems, the size of the search space (i.e. the number of possible solutions) grows exponentially with the size of the problem (e.g. the number of variables to be assigned). Many important combinatorial optimization problems are NP-hard: a well known example is the Traveling Salesman Problem (TSP) [Applegate, 2006], where, given a set of cities and the distances among them, the objective function is the cost of a path visiting all cities once, which has to be minimized.

2.1.1 An example: the satisfiability problem

The *satisfiability* (SAT) problem [Gent and Walsh, 1999] is a well known example of combinatorial decision problem, which consists in assigning values to a set of variables in order to verify a Boolean formula. A conjunctive normal form CNF(k,n,m) SAT problem consists in saying whether there exists an instantiation of a set of n Boolean variables that simultaneously satisfies a set of m clauses, each being the logical OR of k literals, chosen from the set of variables and their negations. An instance is termed *satisfiable* (SAT) if there exists at least one of such instantiations, otherwise it is *unsatisfiable* (UNSAT). Given a bitstring of length n, representing an instantiation of the n variables, the value of all clauses of an instance of the SAT problem can be evaluated in a time $O(km)$, so the problem is NP. With $k = 3$ or greater, it is NP-complete. Satis-

fiability of an instance depends in probability on the clauses to variables ratio: for $k = 3$, a *phase transition* [Mitchell et al., 1992] can be observed at $m/n \approx 4.3$, at which an instance is satisfiable with probability 0.5. This probability quickly goes towards 0 for m/n above the threshold, and towards 1 below. The optimization equivalent of SAT is Max-SAT, where the number of satisfied clauses is maximized: by definition, this problem is NP-hard.

Many other important combinatorial decision problems are NP-complete, for example constraint satisfaction, and graph coloring. Given the definition of this class, each of them can be represented as SAT problem with $k \geq 3$, and solved accordingly. This fact, and the simplicity of the SAT representation, determined a huge effort by the research community to analyze the SAT problem, and devise more efficient solvers for it. In principle, a solution of a SAT problem of size n can be found by simply enumerating and testing all 2^n candidate solutions: it is clear that this approach does not scale well with n, so it cannot be used on real-world problems. Still, many efficient solvers can in principle explore or exclude all possible combinations of n bits. Such *complete* solvers perform backtracking search in order to reduce the amount of exploration: the search space is organized as a binary tree, where each of the two possible assignments of the i-th variable represents a branch, and the 2^n leaf nodes are the candidate solutions. The tree is traversed depth-first, until the assignment of a value to the i-th variable contradicts the formula: at this point, a whole sub-tree can be "pruned", as none of its leaves will verify the formula. If both values have been tested at node i, the search then starts again assigning a different value to the $(i-1)$-th variable. When all the search space has been pruned unsuccessfully, the instance is proved to be unsatisfiable, otherwise the algorithm halts when the first solution is found. While the basic method is deterministic [Davis et al., 1962], many recent versions involve some random aspects, e.g. in the order with which variables are assigned.

When an instance is satisfiable, it can often be solved faster by stochastic local search (SLS) methods [Hoos and Stützle, 2004]. Rather than searching the solution space exhaustively, these methods start from a random candidate solution: after testing it, they search for a neighboring candidate which increases the number of satisfied clauses, When a better neighbor is found, local search is repeated from there, until a solution is reached. When the search arrives at a point where no better neighbor is found, the process is repeated, starting from another random binary string. These solvers are *incomplete*, as they do not perform an exhaustive search in solution space, so they cannot prove unsatisfiability.

2.1.2 Algorithm performance criteria

Time allocation is generally aimed at improving problem solving performance. From a computer scientist's point of view, an algorithm is nothing but a piece of software, being executed on a particular machine. In this sense, its performance may be related to the use of various resources, as CPU time, memory, bandwidth, etc.

In general, the notion of performance will depend on the problem at hand. For algorithms solving decision or search problems. the most commonly considered performance indicator is the CPU time spent, or *runtime*, especially because of its worst-case exponential relationship with the size of the problem. The criteria for time allocation in this case are simple: minimize the time required to solve an instance, or maximize the number of instances solved in a given time.

For optimization problem solvers, the notion of performance is less trivial. We have seen that the search variant of an optimization problem consists of finding an optimal solution, and proving its optimality. Also in this case, two algorithms which both solve an instance can only be

compared based on their runtimes. Proving optimality in NP-hard optimization problems often requires an exhaustive search in (a subset of) the solution space: when solving large instances of combinatorial problems, this cannot be done in a practical time. The same consideration holds for global optimization of continuous functions in high dimensional spaces. In these cases, the runtime may or not be an important issue, relative to the quality of solution found, depending on the application[1]. When a target quality can be set in advance, such problems may be reduced to decision problems, where runtime becomes again the only performance criterion.

Indeed, the computational complexity of both NP-complete decision problems, and NP-hard optimization problems poses a serious practical issue. Much of the research on time allocation is therefore focused on the task of *minimizing the time* spent in solving problems. Before presenting this work, in the next section we refine the concept of time allocation, focusing on the different design decisions it involves.

2.2 The time allocation problem

The time allocation problem has been addressed by different communities, from different perspectives, and using different terminologies. Often, the same term has been used with different meanings, and *vice versa*; before continuing, it will then be useful to precise some of what, in our view, are the key concepts. Rather than proposing a formal taxonomy of time allocation, the sole aim of this section it to better specify a set of "tags" which will ease the task of describing related work, as well as our own contributions. Whenever possible, we tried to use the most widely accepted term, citing its source, or other papers where the term is used in the same sense.

In general terms, a *time allocator* 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 can only use 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 $\mathcal{B} = \{b_1, \ldots, b_M\}$, which have to be solved. To this aim, the algorithms are executed on one or more processors, according to a *schedule* $\mathbf{s} \in \mathcal{S}$ [Finkelstein et al., 2002]. In this context, a *time allocator* (TA) can be defined as any function generating a schedule $\mathbf{s}$, which can be used to solve $\mathcal{B}$ using $\mathcal{A}$.

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.

Existing TAs may differ in the way the schedule $\mathbf{s}$ is represented, as well as in its allowed values $\mathcal{S}$. We do not pose 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. 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*).

In the following, we restrict the discussion to methods where the algorithms are not modified, and cannot interact. Examples of time allocation which we do not consider here include continuous parameter *tuning* and *control* [Battiti et al., 2008]; and sequential composition of algo-

[1] For example, when optimizing the structure of a microprocessor which will be produced in millions of exemplars, it will be worth to invest more time in designing a more efficient and cheaper product; when optimizing the actions of a robot in a time-critical application, a rough solution obtained in a short time may be much more useful than an optimal solution which requires more computation.

rithms, as *search in program space* [Gagliolo, 2007], and *anytime algorithm* scheduling [Horvitz and Zilberstein, 2001].

A first distinction can be made among (single) algorithm selection, and more general forms of allocation, involving the execution of multiple algorithms (as in algorithm *portfolios*).

In both cases, the allocation can be performed once, and kept for the entire set of problem instances $\mathcal{B}$; or repeated independently for each instance $b_m \in \mathcal{B}$. Following Hutter and Hamadi [2005] we term these two approaches *per set* and *per instance* allocation, respectively.

Another independent classification can be made among *static* and *dynamic* schedules [Petrik, 2005a]. Static schedules are stationary, and are set before starting any $a_n \in \mathcal{A}$. Dynamic schedules can be a function of time $\mathbf{s} = \mathbf{s}(t)$, i.e. they can change *while* the algorithms are being executed.

Further dimensions can be introduced based on the information used by a TA in order to produce a schedule. In *oblivious*, or *low-knowledge* techniques [Beck and Freuder, 2004], time allocation is performed independently, "from scratch" for each problem instance b_m: in these methods, the schedule is set according to some heuristics, which may involve interactions with the algorithms. In *non-oblivious* techniques, there is some *knowledge transfer* across subsequent problem instances $b_1, b_2, \ldots$

Non-oblivious methods may be described specifying what information is transferred across instances, and how it is collected. In *model-based* methods, a predictive model of performance for each a_n is learned, based on a sample of its performance; these models are used to allocate time to $\mathcal{A}$. In *model-free* methods, time allocation is directly based on the collected performance data.

Regarding *how* such data is collected, many non-oblivious methods simply assume its availability, or even the availability of a correct model. When the issue of learning is considered explicitly, it can be performed *offline* or *online*. In offline learning techniques, a sample of algorithm performance is collected during a preliminary training phase, which usually consists in solving each instance from a different *training* set $\mathcal{B}_0$, with each available algorithm. In online techniques, the data is updated every time an instance b_m is solved, and used to evaluate the allocation for the following b_{m+1}.

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 2.3 reports several examples of this approach.

Another set of ideas address the sequential allocation of multiple runs of the same algorithm, or of multiple algorithms in parallel, based on the RTDs of the algorithms on the current instance, which are assumed to be available. This line of research includes the foundational works on restart strategies and algorithm portfolios (Sec. 2.4).

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 (Sec. 2.5).

Some interesting oblivious methods, where each instance is addressed separately based on a dynamic interaction with the algorithms solving it, are described in Section 2.6.

Further references which are related to the multi-armed bandit problem will be given in the corresponding chapter, in Section 4.5. Section 2.7 concludes the chapter discussing the state of the art, and outlining the research directions which will be pursued in the rest of the thesis.

2.3 Model based selection, per instance

The potential difference in performance among the best per set and the best per instance selection can be easily understood by comparing the two approaches *a posteriori*. Consider a set of N algorithms $\mathcal{A} = \{a_n\}$, solving a set of M instances $\mathcal{B} = \{b_m\}$. Suppose the runtimes $\{t_n(m)\}$ of the n-th algorithm on the m-th instance are known in advance. The best per set selection of a single algorithm solves all the instances in a time

$$t^*_{set} = \min_n \{\sum_{m=1}^{M} t_n(m)\}, \tag{2.1}$$

while the best per instance selection achieves a performance

$$t^*_{inst} = \sum_{m=1}^{M} \min_n \{t_n(m)\}. \tag{2.2}$$

It is easy to see that $t^*_{inst} \leq t^*_{set}$, with equality holding only if the same algorithm a_n is the fastest on all instances, which is often not the case. Incidentally, note that (2.1) is the best possible performance that can be achieved on the set of instances $\mathcal{B}$, using the set of algorithms $\mathcal{A}$ on a single processor.

In practice, the $t_n(m)$ are not available, and it may be easier to predict which a_n will achieve the best per set performance, than to predict, independently for each b_m, which a_n will solve it fastest. Static per instance selection has to be based on some information related to the each instance. In order to do so, discrete or continuous *features* $\mathbf{x}$ are extracted from the instance, and selection is based on these features, usually conditioning a performance model. Intuitively, the selection will be efficient only if the features allow to discriminate well among instances: i.e. if for any two instances b_1, b_2 having the same features $\mathbf{x}_1 = \mathbf{x}_2$, also $\arg\min_n\{t_i(1)\} = \arg\min_n\{t_i(2)\}$. Nonetheless, many existing algorithm selection methods adopt this approach, sometimes with impressive results: in this section we give some examples.

2.3.1 Origins of the idea

Rice [1976] provides the first rigorous discussion of the algorithm selection problem, casting it in the framework of *approximation theory*. Based on practical examples, such as the selection among numerical algorithms estimating the integral of a function, or among different schedulers in an operating system, several formulations of the problem are proposed. In the most general one, the algorithm selection problem consists in finding a *selection mapping* from the elements of a *problem space* to elements of an *algorithm space*.

A *performance function* maps (*algorithm*, *problem*) pairs to values in a n-dimensional performance measure space, and the aim of selection is to optimize a scalar function of the performance: e.g., in the integration case, a linear combination of speed and accuracy, which can be evaluated integrating a particular function (the problem) using any of the available algorithms. The selection function does not operate directly on the problem space, but rather on its image in a m-dimensional *feature* space, to which problems are mapped via a *feature extraction* function. Rice distinguishes several criteria for choosing a selection mapping, such as maximizing the performance independently for each problem instance; or minimizing the degradation of performance on a given subclass of problem instances, compared to the optimal performance of the per-instance best algorithm. He then goes on to consider several extensions, such as the

selection of the feature space (now called *feature selection*), or the dependence of selection on the performance function (e.g., allowing the user to set the desired trade-off among speed and accuracy).

The analogy with approximation theory comes from the fact that the performance function can be seen as a norm in problem space. Rice concludes noting that approximation theory is a subset of optimization theory, implicitly describing algorithm selection as a form of optimization. According to the "tags" described above, Rice considers offline learning of static single-algorithm selection, both per set and per instance, for decision and optimization problems. As we will see, much of the subsequent work in this field can be formulated within his framework.

Boisvert [2000] provides an interesting history of the impact of Rice's ideas in the field of mathematical software. Many high performance libraries for numerical computation include some form of algorithm selection, for example ATLAS [Whaley et al., 2001]. Several papers on high performance computing describe algorithm selection methods, e.g. [McCracken et al., 2003; Yu et al., 2004].

2.3.2 Algorithm selection as meta-learning

Similar concepts have been proposed in the machine learning community, as a particular case of *Meta-Learning* [Vilalta et al., 2005], using terms as algorithm *recommendation*, *ranking*, or *model selection* [Keller and Giraud-Carrier, 2000; Fürnkranz, 2001; Vilalta and Drissi, 2002; Giraud-Carrier et al., 2004]. Work in this area is mostly applied to the optimization problems addressed in machine learning, as classification or regression [Cherkassky and Mulier, 1998; Bishop, 2006]. For example, Soares et al. [2004] evaluate different values for the kernel parameter of a Support Vector Machine [Vapnik, 1995], on different training data sets. Each data set is described by a set of features. For an unseen data set, the features are first evaluated, and a *ranking* of the kernel parameter values is induced, using a k-nearest-neighbor estimate of performance, based on the distance in feature space between the new data set and the ones used for training. Other examples of *model selection* are presented by MacKay [1992]; Cristianini et al. [1999]; Seeger [2000]. In *landmarking* techniques [Pfahringer et al., 2000; Fürnkranz et al., 2002] the performances of fast base-learners, not included in the algorithm set, are used as instance features, in order to obtain a better discrimination of task difficulty.

2.3.3 Empirical hardness models

Leyton-Brown et al. [2003]; Nudelman et al. [2003] focus on obtaining accurate models of the expected runtime of complete decision problem solvers, conditioned on numerous features of the problem instances, in order to select the per instance best algorithm[2]. For each available algorithm, an *empirical hardness model* [Leyton-Brown et al., 2002] is learned offline, based on the runtimes on several training problem instances: after the initial learning phase, the expected performance of each algorithm on an unseen problem instance is predicted based on instance features, and the best algorithm is selected accordingly. Nudelman et al. [2004] consider several features of SAT instances, some of which are related to the performance of simple solvers, executed for a short time on the instance. A successful example application of this approach is SATzilla [Xu, Hutter, Hoos and Leyton-Brown, 2007; Xu et al., 2008; Nudelman et al., 2004],

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

which won several medals at the last SAT competitions[3]. SATZILLA is a full-fledged algorithm selection method for SAT solvers, in which some of the design decisions are also automated, as the composition of the set of algorithms, as well as the choice of which instance features to use. One of the practical issues of selecting among complete solvers for combinatorial decision problems is that their runtimes may vary across several orders of magnitude: to address this issue, the logarithm of the expected runtime is modeled.

Xu, Hoos and Leyton-Brown [2007] model the runtime of complete solvers on SAT problems at the threshold, using a hierarchical mixture of experts [Bishop, 2006] which estimates the probability that an instance is satisfiable, based on instance features, and uses this probability to mix the predictions of two different models, separately trained on SAT and UNSAT instances respectively. On four different benchmarks, these models correctly predict the satisfiability of an instance between 73% and 98% of the times, which is an impressive accuracy, given that the problem is NP-complete. Unfortunately, an application of these models to algorithm selection has not been presented yet: the authors argue that the cost of misclassification, in terms of runtime, would be too high, as a wrong prediction would cause the selection of the wrong algorithm, and the difference in performance with the actual best algorithm may be huge.

A parameter tuning method, where local search in parameter space is guided by a hardness model, is proposed in [Hutter et al., 2006, 2007; Hutter, 2009]; online selection is advocated in [Hutter and Hamadi, 2005].

The above articles focus on decision problem solvers: more general performance functions are considered in [Xu et al., 2008].

In this section we have seen successful examples of single algorithm selection, on a per instance basis, in both decision and optimization problem domains. While the sophistication of the details involved changes, all these approaches can be described against the algorithm selection framework proposed by Rice [1976]: models of each algorithm's performance, conditioned on features of the problem, are learned offline, based on the results of each of the algorithms in $\mathcal{A}$ on a set of training problems. The models are then used to perform algorithm selection on a per instance basis: for each instance b, the performance of each a_n is predicted, conditioning the model on instance features, and the predicted best a_n is selected to solve b.

In the next section, we will see situations in which the selection of a single algorithm is too risky, and more general forms of allocation have to be adopted.

[3] See `http://www.satcompetition.org`.

2.4 Model based allocation, per instance

In the previous section we have seen methods where the result of time allocation is a single run of a single algorithm. Ideally, a method which could select always the fastest algorithm would achieve the best possible performance (2.2). In practice, selection is never perfect, and the selection of a single algorithm always involves a risk. Let us look again at (2.2). On a single instance b_m, the best possible performance is $t_{n^*}(m) = \min_n t_n(m)$. Selecting the "wrong" a_n, $n \neq n^*$, would cost an overhead of $t_n(m) - t_{n^*}(m)$. In practice, the risk of single algorithm selection depends on how big this overhead can be. In some cases, it may be of several orders of magnitudes. In Section 1.3 we have seen already an example, where this issue makes single algorithm selection impractical: more will be presented in Section 9.3. Similar considerations are reported by Xu, Hoos and Leyton-Brown [2007] (see Sect 2.3.3).

Moreover, when the algorithms involved are randomized, the optimal n^* may change for different runs. Indeed, in some important problem domains there are particular combinations of algorithms and instances for which the variations among runtimes measured for the same algorithm, on the same instance, varies of several orders of magnitude across different runs. In such situations, combining multiple runs of the same algorithm, whether sequentially or in parallel, may lead to the solution faster. Practitioners have exploited this possibility for a long time; the conditions under which such speedup may be observed have been formalized in the last two decades. Some of this work also considers the possibility of exploiting such speedups, performing some sort of time allocation in order to reduce the aforementioned risk. In these papers, per instance allocation is considered, based on the *runtime distributions* (RTDs) of the algorithms on the current instance, which are assumed to be available. Time allocation is again formulated as an optimization problem: the RTD resulting from an allocation is evaluated based on the RTD(s) of the algorithm(s), and the resulting formula is used to optimize the allocation, e.g. minimizing the expected runtime. Before going into the details of the different methods, in the next subsection we introduce the basic functional representations of runtime distributions, and see how they have been used to describe algorithm performance. In the two following subsections, we will see how RTDs can be used to evaluate optimal sequential restarts, and parallel portfolios, respectively.

2.4.1 Runtime distributions

In this section we formalize the notion of runtime distribution, and present some of the work in which it has been used to describe algorithm performance.

Let $T \in [0, \infty)$ be a random variable, representing the *runtime* of an algorithm a, defined as the interval between its starting time and its halting time. T can be fully described in terms of its distribution, usually termed the *runtime distribution* (RTD). As any other distribution, a runtime distribution can be described by its *cumulative distribution* function (CDF),

$$F(t) = \Pr\{T \leq t\}, \quad F : [0, \infty) \to [0, 1], \tag{2.3}$$

which is an increasing function of time, representing the probability that the algorithm halts within a time t. Its derivative $f(t) = dF(t)/dt$ is termed *probability density* function (pdf).

Another representation of the RTD, which will ease some of the math in the following, is the *survival* function

$$S(t) = \Pr\{T > t\} = 1 - F(t), \tag{2.4}$$

which owes its name to the application of statistics to the field of medicine, and represents, in our case, the probability that the algorithm will be still running after a time t. As common practice in the literature on algorithm performance modeling, in the following we will often abuse the term RTD to refer to any of its functional representations, as $F(t)$, $S(t)$, $f(t)$, or others which will be introduced later: note that, given any of these functions, the remaining ones can be obtained analytically.

The expected value of runtime, or *expected runtime*, can be evaluated as

$$E\{T\} = \int_0^\infty t f(t) dt = \int_0^1 t dF(t) = \int_0^\infty S(t) dt. \tag{2.5}$$

A *quantile* $t(\alpha)$ of the RTD is defined as the time at which the probability $F(t)$ of halting reaches a value $\alpha \in [0,1]$, and can be seen as an upper bound on its runtime which holds with probability α. It is evaluated solving the equation:

$$t(\alpha) = F^{-1}(\alpha); \tag{2.6}$$

the quantile for $\alpha = 0.5$ is the *median* runtime.

An important distinction has to be made between the RTD of a randomized algorithm on a given problem instance, which can be sampled by solving the instance repeatedly, each time initializing the algorithm with a different random seed; and the RTD of a randomized (or deterministic) algorithm on a set of instances, which can be sampled by running the algorithm repeatedly, each time solving a randomly chosen instance. In the first case, the random aspects are inherent to the algorithm. In the second, an additional random aspect is involved when drawing the instance. We will refer to these two distributions as the *RTD on the instance* and the *RTD on the set*, respectively; when no distinction is made, the concepts described are valid in both cases.

Given the RTDs $\{F_m(t)\}$ of an algorithm a on each of the M instances b_m of a set $\mathcal{B}$, the RTD on the set $F_{\mathcal{B}}(t)$ can be evaluated analytically, as the distribution of the runtime of a on instance b_m, chosen with probability $1/M$. $F_{\mathcal{B}}(t)$ is then simply the average of the $\{F_m(t)\}$:

$$F_{\mathcal{B}}(t) = \frac{\sum_{m=1}^{M} F_m(t)}{M}. \tag{2.7}$$

When only $F_{\mathcal{B}}(t)$ is available, the $\{F_m(t)\}$ cannot be recovered. A related setting which is often considered in literature is that of instances being drawn from a given distribution on problem instance space: in this case one may study the *RTD* of the algorithm on such instances, i.e. the *RTD on the distribution*.

So far we have implicitly assumed that the solver is Las Vegas in the strict sense, i.e. it will solve the problem and halt in a finite time. A generalized LVA, which is not guaranteed to halt in a finite time, can be characterized by an *improper* RTD, with values in $[0,\infty]$, where $F(\infty)$ is smaller than unity, and equals the integral of the pdf over $[0,\infty)$. For a run which halts, the pdf is $f(t \mid T < \infty) = f(t)/F(\infty)$. Such RTD can characterize the behavior of an algorithm either on a single instance, if not all the runs halt, or on a set of instances, if the algorithm does not always halt on at least one of them. The expected runtime (2.5) is infinite[4], while quantiles

[4] As

$$\begin{aligned} E\{T\} &= \Pr\{T<\infty\}E\{T \mid T<\infty\} + \Pr\{T=\infty\}E\{T \mid T=\infty\} \\ &= F(\infty)\int_0^\infty t f(t \mid T<\infty) dt + [1-F(\infty)]\infty = \infty, \end{aligned}$$

(2.6) are finite if $\alpha < F(\infty)$, infinite otherwise.

Literature on RTD modeling aimed at analyzing algorithm performance is relatively recent: most earlier work were limited to studying simple statistics of a runtime sample, as the mean, variance, median, or other quantiles. In the context of performance analysis of complete solvers for constraint satisfaction, an interest has been recently growing around the existence, in some structured domains, of a second phase transition, in the under-constrained region, where, for some problem instances, the RTD of complete solvers exhibits "heavy tails" [Hogg and Williams, 1994; Gomes et al., 2000]. The CDF of a heavy-tailed RTD is characterized by a Pareto tail:

$$F(t) \rightarrow_{t\rightarrow\infty} 1 - Ct^{-\alpha}. \tag{2.8}$$

In practice, this means that most runs are relatively short, but the remaining few can take a very long time. Depending on C, α, the mean of a heavy-tailed distribution can be finite or not, while higher moments are always infinite. Gomes et al. [2005] explain that the length of a single run depends on the order with which randomized backtracking assigns values to the variables. In some runs, backtracking has to search very deep branches in the tree of possible solutions before finding a contradiction. The same instance may be very easy if solved with a different random reordering of the variables. This is an example phenomenon which is difficult to study based on simple statistics, as mean and variance.

Fig. 2.1(a) displays a plot of the CDF of a complete SAT solver (`Satz-Rand`, [Gomes et al., 2000]) on three instances, obtained encoding structured graph coloring problems (from [Gent et al., 1999], described in Sec. 8.1). The three instances differ in the amount of structure in the problem. A practical method to detect the presence of an heavy tail consists in plotting the survival function on a log-log scale, where, according to (2.8), a heavy tail appears as a straight line with slope $-\alpha$. Fig. 2.1(b) displays such plots, for the same three instances.

Frost et al. [1997] also studies the behavior of complete SAT solvers, but on instances near phase transition, showing that the shape of their RTD is different on solvable and unsolvable instances, and can be modeled using the Weibull and the log-normal distributions, respectively:

$$f(t;\lambda,\rho) = \lambda^{\rho}\rho t^{\rho-1}e^{-(\lambda t)^{\rho}},\ \lambda,\rho > 0, \tag{2.9}$$

$$f(t;\nu,\tau) = \frac{e^{\frac{\log t-\nu}{2\tau^2}}}{t\tau\sqrt{2\pi}}, \tag{2.10}$$

The parameter ρ determines the shape of the Weibull distribution (2.9), which reduces to the exponential distribution for $\rho = 1$: its tail decreases faster than exponentially with $\rho > 1$, and slower with $\rho < 1$. The log-normal distribution (2.10) is a normal distribution on the logarithm of t.

The RTD of local search SAT solvers does not present heavy tails, and is often modeled using exponential distributions [Hoos and Stützle, 1998a; Hoos and Stutzle, 1998b; Hoos and Stützle, 1999]. Hoos [2002] consider a mixture of exponential distributions

$$f(t;\mathbf{w},\boldsymbol{\lambda}) = \sum_i w_i\lambda_i e^{-\lambda t},\ \lambda_i, w_i > 0, \sum_i w_i = 1. \tag{2.11}$$

Chapter 4 of [Hoos and Stützle, 2004] is a valuable source of further discussion and references. In the following, we will see that the issue of heavy tails can indeed be exploited to improve runtime performance, combining multiple independent runs of the same algorithm.

2.4.2 Restart strategies

A restart strategy consists in executing a sequence of runs of a randomized algorithm, in order to solve a single problem instance, stopping the r-th run after a time $\tau(r)$ if no solution is found, and restarting the algorithm with a different random seed; it can be operationally defined by a function $\tau : \mathbb{N} \rightarrow \mathbb{R}^+$ producing the sequence of thresholds $\tau(r)$ employed.

While the origin of the idea is often attributed to the area of global optimization, due to the well known heuristic of restarting search on a rugged functional surface from multiple random points, the earliest analytical study which we are aware of was carried out in the field of communication networks, by Fayolle et al. [1978], who derive the optimal timeout for a simple "send and wait" communication protocol, maximizing the transmission rate. In this context, the process being restarted is the wait for an acknowledgment, and the restart consists in re-sending a packet, assuming that the previous one was lost.

In the field of global optimization, Kolen [1988] applies restarts to learning algorithms for artificial neural networks [Bishop, 1995]. Muselli and Rabbia [1991] perform a theoretical analysis of the conditions under which restarts, or parallel execution, are advantageous, keeping the total computation time fixed. They prove restarts to be beneficial under two conditions: if the survival function[5] decreases less fast than an exponential, and if the RTD is improper. They analyze such conditions on three different parametric RTDs, one of which is an alternative representation of the Weibull distribution (2.9). For this distribution, restarts are beneficial if the shape parameter is $\rho < 1$, detrimental for $\rho > 1$, and indifferent for $\rho = 1$, which corresponds to an exponential distribution. Shonkwiler and van Vleck [1994] propose an alternative analysis of restarts in global optimization, based on eigenvalue theory.

Restart strategies are often used to improve the time performance of complete solvers for decision problems. Luby et al. [1993] prove that the optimal restart strategy is *uniform*, i.e., one in which a constant $\tau(r) = \tau$ is used to bound each run. They show that the expected value of the total run-time T_τ of a uniform strategy τ can be evaluated as

$$E\{T_\tau\} = \frac{\tau - \int_0^\tau F(t)dt}{F(\tau)} \tag{2.12}$$

where $F(t)$ is the CDF of the RTD of the algorithm. When such distribution is known, an optimal cutoff time τ^* can be evaluated minimizing (2.12). Otherwise, the authors suggest a universal, non-uniform restart strategy, whose cutoff sequence is composed of powers of 2: when 2^{j-1} is used twice, 2^j is the next[6]. This results in the sequence $\{1, 1, 2, 1, 1, 2, 4, 1, ...\}$, whose performance t_U is bounded with high probability with respect to the *expected* run-time $E\{T_{\tau^*}\}$ of the optimal uniform strategy, as

$$t_U <= 192E\{T_{\tau^*}\}(logE\{T_{\tau^*}\} + 5) \tag{2.13}$$

Alt et al. [1996]proposes an alternative method, minimizing the tail probability. Gomes and Selman [2001] point out that the number of runs required to find a solution using a uniform strategy follows a geometric distribution, whose exponential decay effectively eliminates the heavy tails of the RTD for the single run, if present. Kautz et al. [2002] assume that the RTD on the instance is not known in advance, but belongs to a known finite set of distributions, from which the correct one can be discriminated based on dynamic features, as described by Horvitz et al. [2001].

[5] Designed as "the complement of F" in the paper.

[6] More precisely, $r = 1, 2, ...$, $\tau(r) := 2^{j-1}$ if $r = 2^j - 1$; $\tau(r) := \tau(r - 2^{j-1} + 1)$ if $2^{j-1} \leq r < 2^j - 1$.

van Moorsel and Wolter [2004a] analyze restarts in the context of communication networks, deriving the CDF of a uniform strategy, along with higher moments of its completion time T_τ, and an optimal threshold for a finite number of restarts. They also present a necessary and sufficient condition for a restart at time τ to be beneficial,

$$E\{T\} < E\{T - \tau | T > \tau\}, \tag{2.14}$$

showing that it holds for hyper-exponential distributions (i.e., mixtures of exponentials (2.11)), but not for hypo-exponential ones (sums of random variables with different exponential distributions). A strategy minimizing higher moments of T_τ is proposed in van Moorsel and Wolter [2004b].

To summarize, a restart strategy is effective when the tail of the survival function decreases less than exponentially: this condition is met for improper, heavy-tailed and hyper-exponential distributions, as well as for the Weibull distribution with $\rho < 1$.

The effect of a restart strategy can be better understood with a graphical explanation. In Figure 2.2 we report a simple experiment, where `Satz-Rand` is used on a sat-encoded instance of the graph coloring problem, on which its RTD is heavy tailed (Set 2, instance 0, corresponding to the red line in Fig. 2.1). Figure 2.2(a) reports the cost of a uniform restart strategy, given the threshold τ. The cost was evaluated based on (2.12), the function being optimized by an optimal restart strategy. There is a single optimum τ^*, at 1.85×10^6. Figure 2.2(b) plots the heavy tailed CDF of the algorithm (red line) compared with the one obtained restarting the algorithm with the optimal restart threshold (green line). The black vertical lines indicate the restarts. It can be seen that τ^* lies exactly at the "knee" where the heavy tail starts, and that the restarts effectively "cut" the tail: only the initial portion of RTD, between 0 and τ^*, is repeated for each run, allowing the CDF to quickly arrive very close to 1, effectively removing the heavy tail. Fig. 2.3 reports the log-log plot of the survival function (a), comparing with the effect of a parallel portfolio (b, see next section).

The universal strategy of Luby is an oblivious, per set time allocator. The optimal uniform strategy, obtained minimizing (2.12), is instead per instance, non-oblivious, model based, but the model is assumed to be available. In the next subsection we will present analogous time allocators for more general portfolios of one or more algorithms.

2.4.3 Algorithm portfolios

It is well known that parallelization of computer programs can speed up computation, in terms of the "wall-clock" time, but in the field of parallel computing it is generally assumed that the total amount of computation does not change, and it can only be subdivided among different processors. This is not the case for search algorithms: different parallel runs of an algorithm, solving the same instance, will arrive at the solution at different times; but, once the fastest one solves the instance, the remaining ones can be stopped, and their residual computation does not need to be carried out. In this sense, parallelizing search may not only reduce wall-clock time, but even the total amount of computation. This phenomenon was observed early on for backtracking search (e.g., by Janakiram et al. [1988]).

Intuitively, a *homogeneous* portfolio, composed of multiple copies of the same algorithm, initialized with different random seeds, offers a similar advantage as a restart strategy: both approaches can help reduce runtime, for example when the RTD of the algorithm displays heavy tails. The performance of an arbitrary portfolio depends on the RTDs of the algorithms in $\mathcal{A}$, and on the schedule $\mathbf{s}$. In the work presented in this subsection, the RTDs on the problem instance

are assumed to be available. Allocation is performed per instance, evaluating the RTD of the portfolio as a function of $\mathbf{s}$, and optimizing $\mathbf{s}$ accordingly.

The evaluation of the RTD of parallel algorithm portfolios has been carried out independently in previous work, in order to allocate time to the algorithms, or simply to analyze performance. In the following, we present a derivation based on the survival function (2.4), which has the merit of simplifying the formulas.

Consider a portfolio of N algorithms $\mathcal{A} = \{a_1, \ldots, a_N\}$, solving the same problem instance. The a_i can be different algorithms, different parametrizations of the same algorithm, multiple copies of the same randomized algorithm differing in the random seed, or mixtures thereof. They are executed in parallel, and share the computational resources of a single machine[7] according to a *resource sharing* schedule, or *share*, $\mathbf{s} = \{s_1, .., s_N\}$, $s_i \geq 0$, $\sum_{i=1}^{N} s_i = 1$; i.e., for any amount δt of machine time, a portion $\delta t_i = s_i(t)\delta t$ is allocated to a_i. With this notation, single algorithm selection can be represented as a share with a single $s_i = 1$. A *uniform* algorithm portfolio is executed according to a share $\mathbf{s}_{\mathcal{U}} = (1/N, ..., 1/N)$. This simple allocation solves any problem instance in a time $N \min_i\{t_i\}$, so it is always a factor N worse than the best possible performance (2.2).

The runtime of each a_i on the current problem instance is a random variable $T_i \in [0, \infty)$. We assume that the T_i are independent (e.g., the algorithms cannot interact), and that at least one of them is finite (i.e., at least one of the algorithms will solve the instance). The instance is solved as soon as one of the algorithms finds a solution: the runtime of the portfolio $T_{\mathcal{A}}$ depends on the T_i and on the share $\mathbf{s}$ as

$$T_{\mathcal{A}}(\mathbf{s}) = \min_i \frac{T_i}{s_i}, \tag{2.15}$$

so it is also a random variable. The evaluation of its distribution is more intuitive if we reason in terms of the survival function: at a given time t, each a_i has used a share of computation time $s_i t$. The probability $S_{\mathcal{A}}(t; \mathbf{s})$ of *not* having obtained a solution is equal to the joint probability that no single algorithm has obtained a solution within its time share. As we assume solution events to be independent for each a_i, this joint probability can be evaluated as the product of the individual survival functions $S_i(s_i t)$

$$S_{\mathcal{A}}(t; \mathbf{s}) = \prod_{i=1}^{N} S_i(s_i t), \tag{2.16}$$

which, in CDF form, corresponds to

$$F_{\mathcal{A}}(t; \mathbf{s}) = 1 - \prod_{i=1}^{N} [1 - F_i(s_i t)]. \tag{2.17}$$

Note that the assumption of independence of the T_i, which allows to express (2.16) as a product, is justified by the use of the *actual* RTDs of the a_i *on the current instance*, and the fact that the a_i do not interact: we will return on the implications of this assumption in Section 6.1.

If the N algorithms are executed on N different processors, the above formula can be simplified setting each s_i to 1: for example, (2.17) becomes

$$F_{\mathcal{A},N}(t) = 1 - [1 - F_i(t)]^N. \tag{2.18}$$

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

This alternative way of combining multiple runs of the same randomized algorithm can also reduce heavy tails, as the area under the tail decreases exponentially with the number of processors N: in Figure 2.3(b) we display this effect on the RTD of `Satz-Rand`.

More generally, one can consider the allocation of Z processors, according to a discrete share $\mathbf{z} = \{z_1, .., z_N\}$, where $z_k \in \{0, 1, \ldots, Z\}$ indicates the number of processors allocated to copies of a_k, so $\sum_{k=1}^{N} z_k = Z$. In this case (2.16,2.17) become respectively

$$S_{\mathcal{A},Z}(t;\mathbf{z}) = \prod_{k=1}^{N} S_k^{z_k}(t), \tag{2.19}$$

$$F_{\mathcal{A},Z}(t;\mathbf{z}) = 1 - \prod_{k=1}^{N} [1 - F_k(t)]^{z_k}. \tag{2.20}$$

To our knowledge, it is again Muselli and Rabbia [1991] who first propose a similar analysis, evaluating the RTD of N parallel searches of the global minimum of a function in a form equivalent to (2.18).

Huberman et al. [1997] study the combination of complete solvers, introducing the use of the term *portfolio*, borrowed from finance, to point out that the method can reduce the "risk" of wasting computation time. They analyze two examples, each with 2 identical algorithms, obtained generating runtimes from a bimodal distribution, and from the RTD of a graph coloring heuristic, sampled on a single problem instance. In both cases they plot the expected value vs. the *risk* (standard deviation) of solution time, varying s_1 between 0 and 1, and identify the *efficient frontier* of the curve, i.e. the set of points (and the corresponding s_1 values) which dominates all other points in terms of both risk and expected value. The RTD of the portfolio is evaluated using a discrete expression of its pdf, which results in an expression equivalent to (2.17). The paper goes on to discuss a networking application of restart strategies, presents a further experiment where communication among the algorithms reduces both expected runtime and risk, and conclude suggesting the possibility of estimating the RTD online, and adapting the values of the s_i accordingly[8]

Gomes and Selman [2001] provide a theoretical and empirical validation of the portfolio approach, pointing out that its advantage depends essentially on the RTD of the algorithm(s) at hand, and suggesting that practical approaches to algorithm portfolio design should feature mechanisms for estimating such distributions. Three approaches are considered: parallel execution on multiple processors, parallel execution on the same processor, and restart strategies. The analysis is again carried out evaluating the discrete pdf of the portfolio RTD. All three scenarios are illustrated with experiments with constraints satisfaction and mixed integer programming solvers, plotting the expected time vs. risk, and proposing to select portfolios from the efficient frontier. The paper concludes suggesting that the portfolio approach can encourage the development of more risk-prone algorithms, e.g. based on depth-first search, as it can make these more usable, effectively removing the heavy tails.

The portfolios described so far are based on static shares, also termed *resource sharing* schedules [Sayag et al., 2006]: the actual task switching among algorithms on a single serial processor is left to the scheduler of the operating system. Other works on algorithm portfolios consider a different machine model, in which a single algorithm is active at a given time, and allocation consists in selecting a dynamic schedule, termed *task-switching* [Sayag et al., 2006; Streeter et al., 2007] or *suspend-resume* schedule [Finkelstein et al., 2003], according to which the execution of

[8] Referencing a work in preparation, which unfortunately we could not find.

the different algorithms is interleaved. Such schedules can be described by a sequence of pairs (i, τ_i), indicating the index i of the algorithm, and the corresponding computation time value τ_i. Often, restrictions on the possible schedules are considered, for example allowing a finite set of possible time values τ_i, and schedule space is represented as a tree. One such approach based on runtime distributions is proposed by Finkelstein et al. [2002, 2003], 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. Examples of allocation to two algorithms are presented with artificially generated runtimes, and a real Latin square solver. The computational complexity of the approach is exponential in the number of algorithms, due to the tree search.

The time allocators described in this section rely on the knowledge of the actual RTDs on the problem instance at hand, which allow to evaluate (2.12) and (2.17) correctly, and set an optimal share accordingly. In principle, they could be used as a basis for implementing practical methods, based on *estimates* of these RTDs. This is the approach taken in our contribution: we will return on the issue of evaluating (2.16) using estimates of the instance RTDs in Chapter 5.

Depending on the RTDs, the optimal allocation may be equivalent to single algorithm selection, i.e., the optimal share may have a single $s_i = 1$: in other words, the methods described here generalize single algorithm selection, producing optimal allocation even when running a single algorithm is not the best choice.

Analogous to single algorithm selection, also here the problem of setting the optimal share is represented as an optimization problem. For the uniform restart strategy of Luby et al., the share is represented as a scalar $\tau \in [0, \infty)$: in this case the optimal value can be found easily. In the static portfolios of Huberman et al. and Gomes and Selman, the share is a vector $\mathbf{s} \in \Delta^N$, where $\Delta^N \in [0, 1]^N$ is the N-dimensional *simplex*, so the search space is again continuous, with dimensionality $N - 1$, as the s_i have to sum to 1. In the dynamic task-switching portfolios of Finkelstein et al., the schedule is a sequence of pairs (i, τ_i), and the search space can be written as $\cup_{k=0}^{\infty}[\{1, \ldots, N\} \times \mathbb{R}_{>0}]^k$. In both these latter representations, the size of the search space is exponential in the number of algorithms N, so is the worst case computational cost of finding the optimal share. In the next section we will see that this intuition is formally confirmed in research on per set allocation.

2.5 Model free allocation, per set

Recently, some authors investigated the possibility of evaluating an optimal per set schedule for a portfolio, based directly on a runtime sample. In work presented in this section, the runtime $t_n(m)$ of each of the N algorithms $a_n \in \mathcal{A}$ is collected on each of M training problem instances $b_m \in \mathcal{B}_0$. For an arbitrary schedule space $\mathcal{S}$, the optimal per set share can be found a posteriori, minimizing the runtime of the portfolio on the M training instances

$$t^*_{\mathcal{A}}(\mathcal{B}) = \min_{\mathbf{s} \in \mathcal{S}} \{ \sum_{m=1}^{M} t_{\mathcal{A}}(m; \mathbf{s}) \}. \tag{2.21}$$

The corresponding optimal share can then be used to solve further problem instances, generated by the same probability distribution which generated the training instances: based on this assumption, probabilistic bounds on the performance on further instances are presented. The

problem of optimizing the share is proved to be NP-hard [Petrik and Zilberstein, 2006; Sayag et al., 2006]. Streeter et al. [2007] proposed a 4-optimal approximation which can be evaluated in polynomial time.

Petrik and Zilberstein [2006] evaluate a per set static resource sharing schedule, for both decision and optimization problem solvers. For optimization problems, 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 performance on unseen instances, sampled from the same distribution, are given based on the PAC learning framework [Mitchell, 1997]. Their method finds a locally optimum share in an iterative manner: the authors propose to restart the optimization of the share, with different random initializations, in order to improve its performance, but also describe a global optimization approach, which explores the local optima exhaustively, with a computational complexity of $O(MN(M+1)^{\binom{N}{2}})$. The problem of optimizing the share is found to be NP-hard.

Dynamic resource sharing schedules are considered in [Petrik, 2005b,a]. 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) (Puterman [1994]), and a variation of dynamic programming is used to select the per-set optimal dynamic schedule, with a computational complexity of $O(MN^2\binom{b}{N!})$

Sayag et al. [2006] consider both resource sharing and task switching schedules of deterministic algorithms, minimizing the time to solve a set of problem instances with a static, per set schedule. They prove that for any given resource sharing schedule, it is possible to find a task switching schedule whose performance is equal or better. More precisely, if t^*_{inst} is the performance (2.1) of an "oracle", running only the fastest algorithm for each instance, and t^*_{rs} and t^*_{ts} are the runtimes (2.21) of the best per set resource sharing and task switching schedules, respectively, then the following inequalities hold:

$$t^*_{ts} \leq t^*_{rs} \leq N t^*_{inst}. \tag{2.22}$$

Notice that the last inequality is trivial, as the right hand term corresponds to the performance of a uniform portfolio. After showing that finding the per-set-optimal resource sharing schedule, or even a ϵ approximation thereof[9], is NP-hard for some $\epsilon > 0$, 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 allocator: during an initial learning phase, the performance of each algorithm is measured on each of the problem instances, 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, Streeter et al. [2007] propose a polynomial time method for evaluating a "greedy" task switching schedule, whose time cost is a factor of 4 worse than the optimal, proving that finding a better approximation is NP-hard. 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 $\mathbf{s}$, i.e. $G(t; \mathbf{s}) = \text{card}(\{b_m :$

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

$t_{\mathcal{A}}(m;\mathbf{s}) \leq t\}$). 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

$$(n_{k+1},\tau_{k+1}) = \arg\max_{(n,\tau)} \frac{G(\tau+\mathrm{T}_k;\{\mathbf{s}_k,(n,\tau)\}) - G(\mathrm{T}_k;\mathbf{s}_k)}{\tau}, \tag{2.23}$$

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 $\{t_n(m)\}$; if available, the set RTD of the portfolio with share $\mathbf{s}$ can be used instead. Both this paper and the following [Streeter and Smith, 2008] present also offline and online per set allocators. In the offline case, several runs of each algorithm are made for each problem instance, and the expected runtime of the portfolio for an arbitrary share is estimated based on the collected sample, using an ad-hoc technique. The online methods will be described in Section 4.5, after we introduce the multi-armed bandit problem, on which such allocators are based.

2.6 Low-knowledge approaches

Oblivious time allocators are characterized by the absence of any knowledge transfer across problem instances. Some oblivious methods are simple per set heuristics, which are repeated in exactly the same way on each instance: one example is the universal strategy of Luby et al. [1993] (Sec. 2.4.2). Another example is the uniform portfolio $\mathbf{s}_{\mathcal{U}}$. The papers we describe in this section are instead characterized by a per instance approach, in which the schedule produced is dynamic, and is evaluated based on the interaction with the a_i at runtime, based on simple performance indicators which are used to somehow predict the performance of each algorithm at the end of the run.

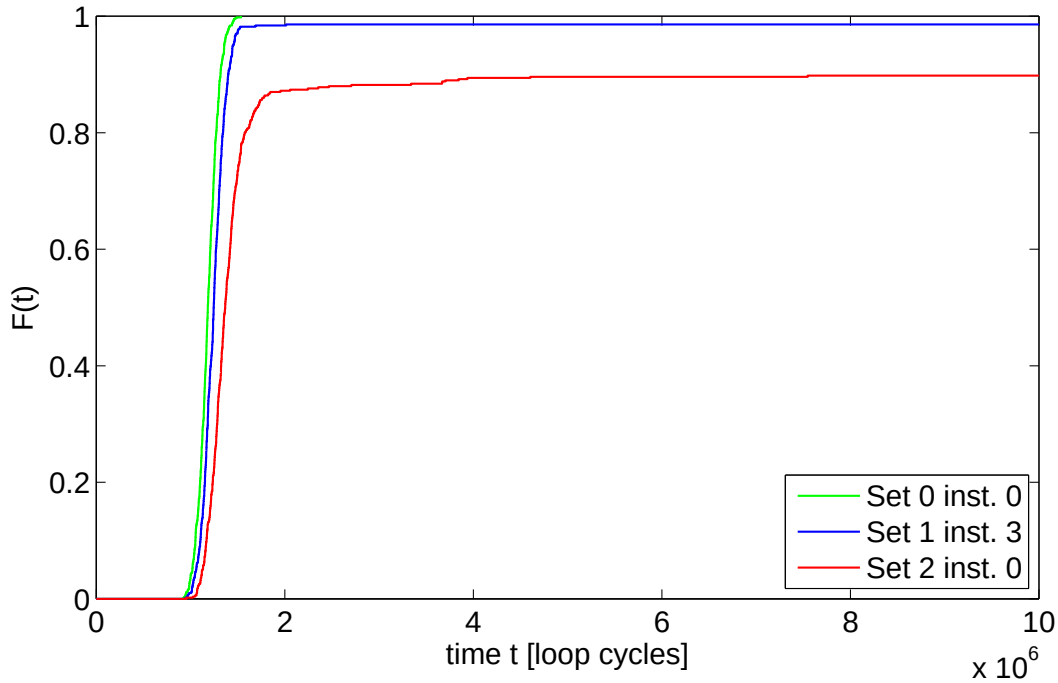
The "parameter-less GA" [Harick and Lobo, 1999] may be viewed as a specialized heuristic for dynamic selection of population size in genetic algorithms (GA, Holland [1975]). It consists of a sequence of simple generational GAs, with no mutation, differing only in population size, generated and executed according to a fixed interleaving schedule that assigns more runtime to smaller populations. Each time the i-th population, of size 2^i, is updated for 4 generations, the next population, of size 2^{i+1}, is updated once, such that each population performs twice the number of fitness function evaluations of the next one[10]. Once a population achieves the highest average fitness, all smaller populations are discarded. As there is no mutation, when a population converges it can also be discarded, as it will not evolve further. This heuristic is based on the simple intuition that if a population reaches the average fitness of a smaller one, while being run at most half of the time, then the smaller population is probably drifting too slowly, so one may safely stop updating it, as it is unlikely to produce a better solution.

"Low-knowledge" approaches can be found in [Beck and Freuder, 2004; Carchrae and Beck, 2005], where various simple indicators of current solution improvement are used for algorithm selection, in order to achieve the best solution quality within a given time contract. In [Beck and Freuder, 2004], all available algorithms are run for a fraction of the contract, and a performance predictor is then used to select a single one for the remaining time. In [Carchrae and Beck,

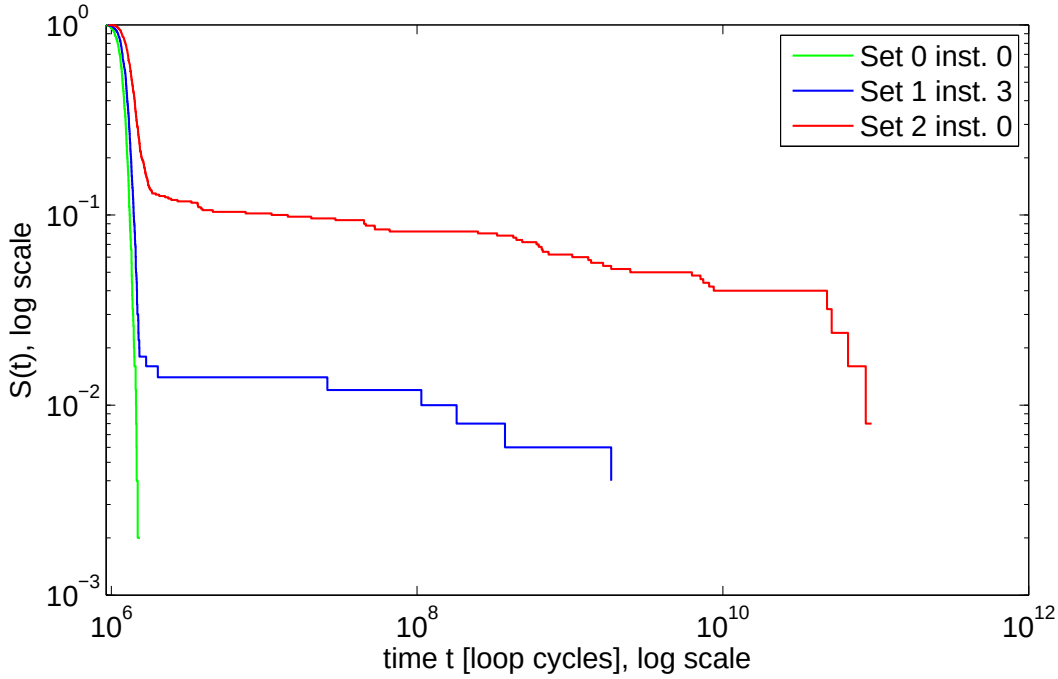
[10] It is interesting to compare the order with which the populations are updated, and the pattern of the sequence of restart thresholds in the universal strategy of Luby et al. [1993] (see Sec. 2.4.2): in both cases, a doubling trick is used to reach the optimal value for a parameter which is directly related to the computational cost, but while in the universal restart the same amount of computation is spent on each threshold value 2^i, in the parameter-less GA the i-th population consumes a portion 2^{-i} of the total computation.

2005], the selection process is iterated: machine time shares are based on a recency-weighted average of performance improvements.

We adopt a similar approach in [Gagliolo et al., 2004], where we consider optimization algorithms whose objective has to reach a target value. The time to solution is guessed based on a shifting-window linear extrapolation of the learning curves; the algorithms are then ranked based on these guesses, and the r-th expected fastest solver gets a share $s_i = 2^{-r}$. This simple general purpose heuristic is competitive with the parameterless GA, which is heavily based on domain knowledge.

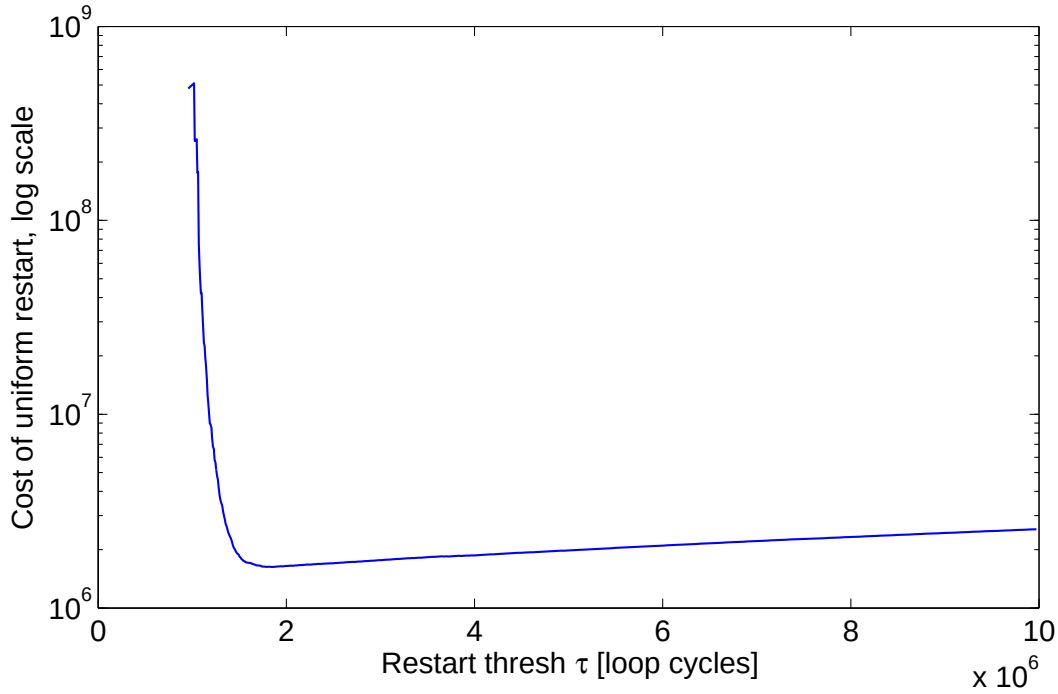


(a)

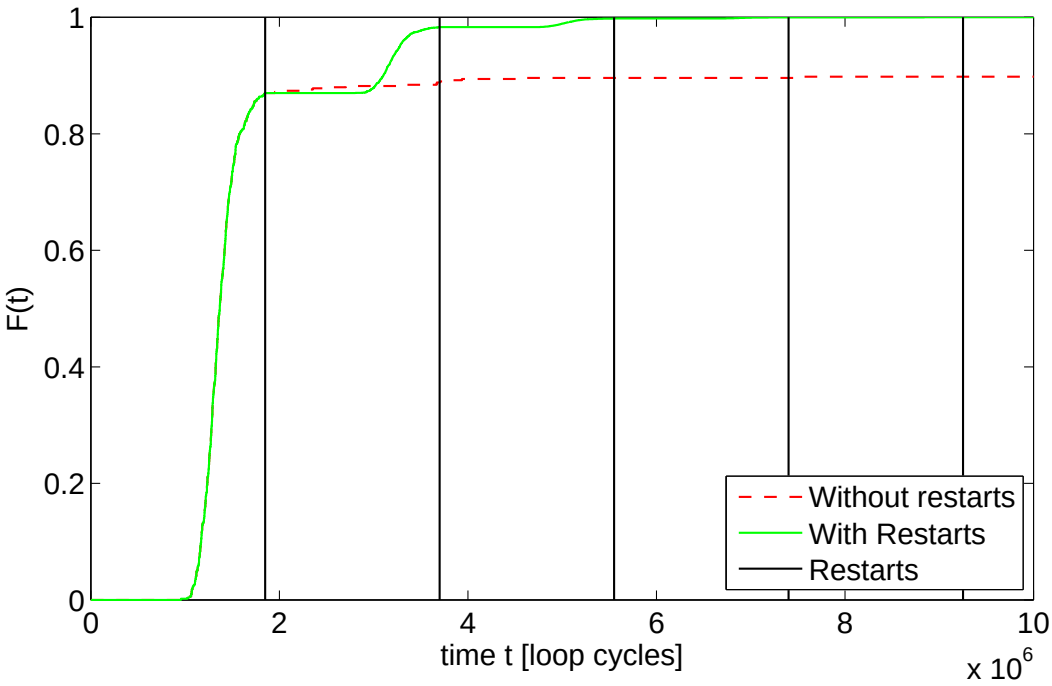


(b)

Figure 2.1. (a) CDF of the RTD of `Satz-Rand` on three instances, differing in the amount of structure. (b) Log-log plot of the survival function for the same three instances. Two are heavy-tailed, as the tail can be approximated by a straight line.

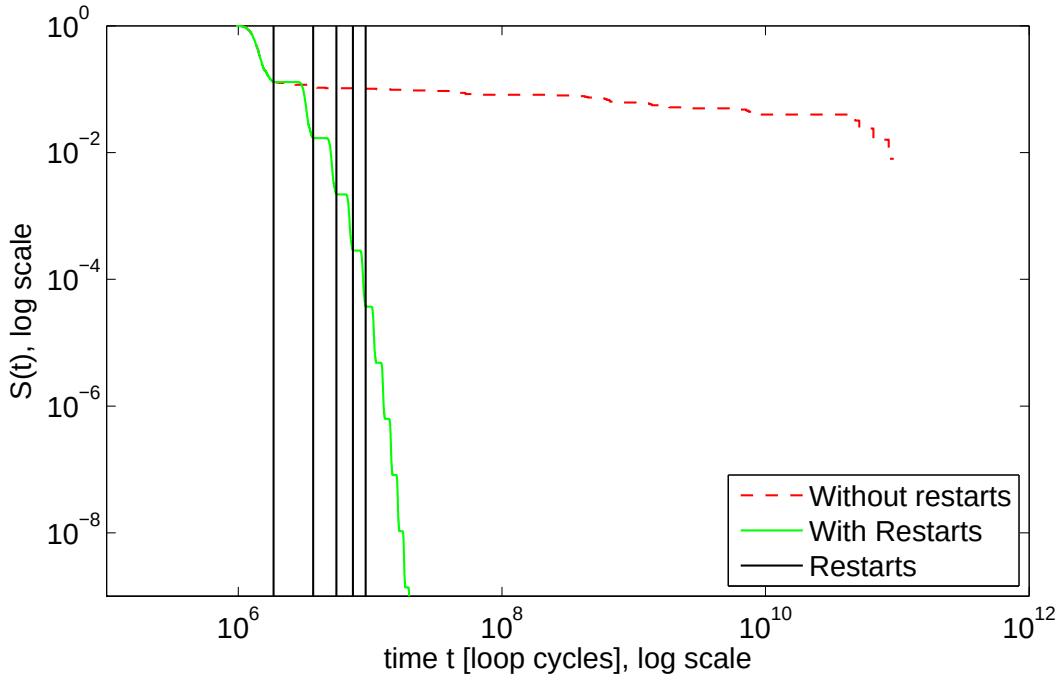


(a)

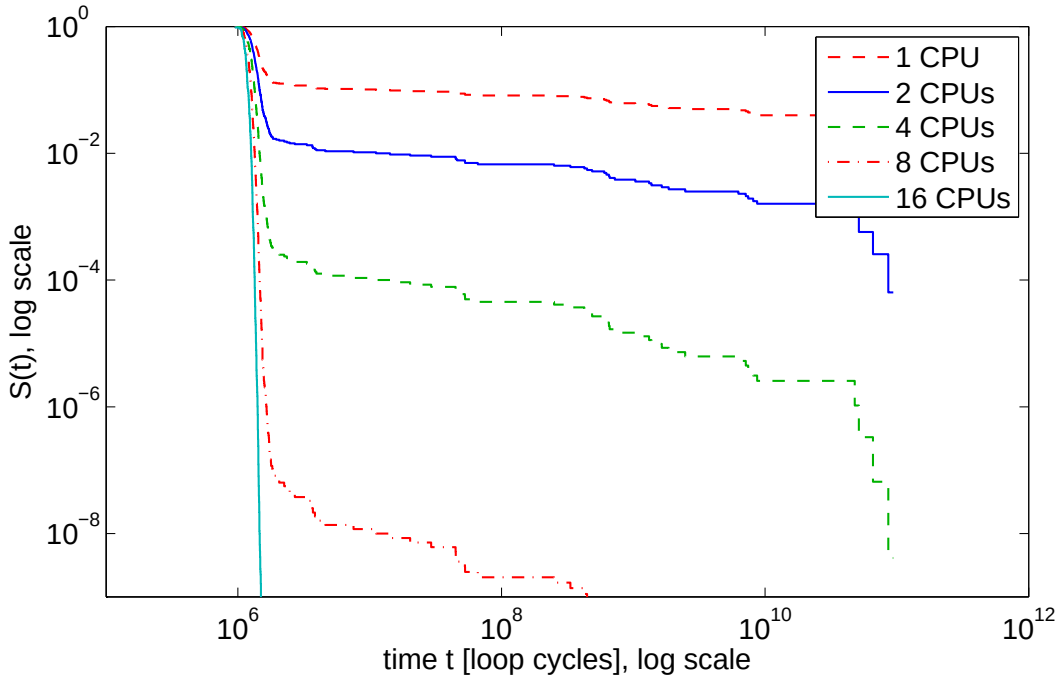


(b)

Figure 2.2. `Satz-Rand` on instance 0 from set 2. (a) Cost of uniform restart (2.12) for `Satz-Rand` on a SAT instance: the minimum is the optimal restart threshold $\tau^* = 1.85 \times 10^6$. (b) Original CDF (red) compared with the CDF of the algorithm with restarts (green). Each restart replicates the same portion of the original CDF below τ^*. Black vertical lines indicate restarts.



(a)



(b)

Figure 2.3. `Satz-Rand` on instance 0 from set 2. (a) Tails of the survival function of the original RTD (red) compared with that of the algorithm with restarts (green). Black vertical lines indicate restarts: only the first 5 are represented to avoid cluttering the graph. Each restart replicates the same portion of the original survival function, before the restart threshold. (b) Effect of running the algorithm without restarts, on multiple CPUs. Also in this case the combination of multiple runs reduces the heavy tail: adding CPUs "pushes" the tail down.

2.7 Discussion

In this chapter we presented related work on time allocation. We first discussed the opportunity of performing time allocation, along with its possible aims, focusing on the practically relevant problem of reducing the computational cost of problem solving (Sec. 2.1). We then precised some of the terminology used in this thesis, and defined the time allocation problem in a slightly more precise way (Sec. 2.2).

We have seen that per instance allocation, performed independently for each instance, is in principle more efficient than per set allocation, performed once for a whole set of instances (Sec. 2.3): the literature is rich in examples in which this simple principle is exploited, with impressively good results. Per instance allocation has to be based on features of the problem instance. In existing work on single algorithm selection (Sec. 2.3.3), instance features are used to condition regression models of the expected runtime.

In Section 2.4 we have described situations in which the selection of a single algorithm is too risky. A more general approach is that of algorithm portfolios, where several algorithms run on one or more processors, solving the same instance: once one algorithm solves it, the whole portfolio halts. The most general model of runtime performance is the runtime distribution (RTD): having access to the RTD of the algorithms on the current instance allows to allocate time in an optimal way. The optimal allocation may share computation time among multiple algorithms, or run just one, depending on the shape of the RTDs.

In both cases, allocation is per instance. The quality of the allocation will depend on the information available. In single algorithm selection, it is implicitly assumed that the right features are used, and a reliable model is available, and that this combination allows us to discriminate among instances precisely enough. In portfolios, the issue of modeling is not addressed explicitly, even though it has been suggested [Huberman et al., 1997; Gomes and Selman, 2001].

These two ideas can be combined: instead of models of expected runtime, aimed at performing single algorithm selection, one could estimate a model of the whole runtime distribution, and use it to optimize an algorithm portfolio. To allow per instance allocation, such models have to be conditioned on instance features: this means that we need to learn regression models of the RTDs of the algorithm.

In more recent work on per set optimal portfolios (Sec. 2.5), parallel execution is again aimed at reducing the risk, but on a set of instances. Intuitively, this could be done basing the allocation on the RTDs on the set of instances; unfortunately, we have seen in Sec. 2.4 that this approach would violate the independence hypothesis in evaluating the RTD of the portfolio (2.16). The per set optimal share is therefore evaluated minimizing expected time on a set of training instances, on which the runtimes of the algorithms have been already evaluated, offline. Assuming that subsequent instances will be similar to the training instances, bounds on future performance can be derived. This work also proves that finding an optimal share is hard, with a complexity which is exponential in the number of algorithms N.

In the more general dynamic portfolios, the allocation of resources is allowed to change over time. Sayag et al. [2006] proves that the best dynamic allocation cannot be outperformed by the best static allocation: the proof can be easily extended to per instance allocation. In Sec. 2.6 we have seen some interesting examples of oblivious dynamic allocation. In these papers, good results are obtained without any knowledge transfer, on a single instance, just by adapting the allocation dynamically based on information obtained from the algorithms during their execution. This line of work can be seen as a confirmation of the practical efficiency of dynamic allocation.

An advantage of oblivious methods is that they do not have to pay the huge computational cost of the training phase of non-oblivious methods. Apart from the exceptions of Streeter et al. [2007]; Streeter and Smith [2008], all other non-oblivious methods we have seen in this chapter follow an offline approach. A sample of algorithm performance is obtained solving each of a set of training problems with each of the available algorithms. The sample is then used to learn models of performance, in per instance algorithm selection (Sec. 2.3); or to directly evaluate an optimal per set allocation (Sec. 2.5).

Intuitively, there must be a trade-off between the cost of the training phase, and the performance on future instances. Increasing the number of training instances has an obvious impact on the cost of the training phase, but it also allows to collect more information about each algorithm's behavior, and should improve allocation on future instances. In most work on model-based algorithm selection (Sec. 2.3), this trade-off is ignored, and the number of training instances is chosen heuristically. In some of the work on per-set allocation, in Section 2.5, the required number of training instances is evaluated in a sound way, in order to obtain the required performance on future instances with a given probability. Even in this case, the training phase essentially consists in solving the same instances over and over again. In Streeter et al. [2007]; Streeter and Smith [2008], the issue is avoided, adopting an online learning method: the results reported are comparable to the ones of much more costly offline methods.

To summarize, we have identified two orthogonal directions of research in time allocation for solving search problems, both promising in terms of the potential reduction in computational complexity, and nearly unexplored in existing research. One consists in modeling the RTDs of the algorithms, in order to allocate time in a more general way than single algorithm selection. To approximate a per instance optimal allocation, these models should allow conditioning on instance features. To perform dynamic allocation, it should also be possible to update the predictions of the models during the execution of the algorithms. Modeling the distribution of random events in time is the subject of *survival analysis*. In the next chapter we will look at some of the available modeling methods in this area, which include regression models, and see how these models can be updated dynamically.

Another dimension to be explored is related to the cost of learning such models, and its trade-off relationship with performance. An extensively studied paradigm of exploration-exploitation trade-offs in the online setting is the *multi-armed bandit problem*. In Chapter 4 we will describe the most recent work on this topic. After that we will be ready to propose our contributions, in Part III.

Chapter 3

Algorithm survival analysis

Many time allocators, including the ones presented in this thesis, are based on models of algorithm performance. When the sole aim of allocation is to minimize runtime, the most general model is the runtime distribution. The branch of statistics devoted to modeling the random occurrence of events in time is *survival analysis* [Klein and Moeschberger, 2003; Collet, 2003; Machin et al., 2006]. Researchers in medicine are typically interested in time to death (hence the name), but the analysis of *time-to-event* data occurs in other fields, and different terms are used to label it. Engineers modeling the duration of a device speak of *failure analysis*, or *reliability theory* [Nelson, 1982]. Actuaries setting premiums for insurance companies use the term *actuarial science*.

In this chapter, we review the basic concepts of survival analysis, and discuss their application to algorithm performance modeling. Section 3.1 introduces the notion of *hazard*. When sampling the runtime of an algorithm, one often has to deal with runs which exceed a predefined timeout value: the statistically correct way of interpreting such *censored* information is presented in Section 3.2. Section 3.3 goes on to describe estimation from censored data, distinguishing among parametric and non-parametric methods. Regression models of the RTD, which can be conditioned on instance features, are considered in Section 3.4. The topics covered in the last two sections are more specific to our work: Section 3.5 proposes a simple way of updating predictions during algorithm execution, which we will use to implement a *dynamic* algorithm portfolio. Section 3.6 discusses the bias induced when using the runtime of fast algorithms as a timeout for the runtime of slower algorithms, as it happens in a portfolio. Section 3.7 summarizes the chapter.

3.1 The hazard function

The basic functional representations of an RTD were introduced in Section 2.4.1: the CDF $F(t)$ (2.3); its complement, or survival function $S(t)$ (2.4); and its derivative, the pdf $f(t)$. Another important concept in survival analysis, on which many estimators are based, is the *hazard* function $h(t)$, quantifying the instantaneous probability of the event of interest occurring at time t, given that it was not observed earlier:

$$h(t) = \lim_{\Delta t \to 0} \frac{\Pr\{t \leq T < t + \Delta t \mid T \geq t\}}{\Delta t} = \lim_{\Delta t \to 0} \frac{\Pr\{t \leq T < t + \Delta t\}}{\Delta t \Pr\{T \geq t\}} = f(t \mid T \geq t), \qquad (3.1)$$

where $f(t \mid T \geq t) = f(t)/S(t)$ if $S(t)$ is continuous at t. The integral of (3.1) is the *cumulative* hazard $H(t)$. This quantity has the following relationship with the survival function:

$$H(t) = \int_0^t h(\tau)d\tau = \int_0^t \frac{dF(\tau)}{S(\tau)} = -\ln S(t), \tag{3.2}$$

or, conversely, $S(t) = \exp(-H(t))$. This function allows to derive an elegant representation of the RTD of a portfolio (2.16) (see Sec. 2.4)

$$H_{\mathcal{A},\mathbf{s}}(t) = -\ln(S_{\mathcal{A},\mathbf{s}}(t)) = \sum_{i=1}^{N} -\ln(S_i(s_i t)) = \sum_{i=1}^{N} H_i(s_i t). \tag{3.3}$$

Apart form the terms s_i, (3.3) is the method used by engineers to evaluate the failure distribution of a *series* system, which stops working as soon as one of the components fail, based on the failure distribution for each single component (Nelson [1982]): this is indeed analogous to a portfolio, which finds a solution as soon as one of its algorithms does.

In *discrete* distributions, the probabilities are defined on a discrete set of points t_i. These distributions are important in practice as most nonparametric estimator, including the ones we will use, are represented on a discrete set. In this case the pdf and the hazard are discrete, defined only for t_i, while the CDF, survival function, and cumulative hazard are step-wise functions, defined as sums. The CDF and survival function are defined as

$$F(t) = \Pr\{T < t\} = \sum_{t \leq t_i} f(t_i), \tag{3.4}$$

$$S(t) = \Pr\{T \geq t\} = \sum_{t > t_i} f(t_i), \tag{3.5}$$

while the hazard becomes

$$h(t_i) = \frac{\Pr\{T == t\}}{\Pr\{T \geq t\}} = \frac{f(t_i)}{S(t_{i-1})}. \tag{3.6}$$

3.2 Censoring

The issue which distinguishes survival analysis from classical statistical analysis is data *censoring*, which refers to a situation in which the exact time of the event of interest is not known, but is known to lie in some (possibly open) interval. For example, in a medical research analyzing survival of patients with a particular disease, some patient will hopefully still be alive at the time of performing the analysis of the data. The only information available about one of these subjects is that she has survived up to a certain time, which constitutes a lower bound on her survival time[1].

Censoring is often the result of the experimenter's decisions, aimed at reducing the duration of an experiment. For example, in estimating a duration model for a newly produced light bulb, an engineer could leave a large number of prototypes turned on for a predetermined period of

[1] More precisely, one can distinguish between *right* censoring, when, as in the above example, the interval is open on the right; *left* censoring, when the interval is open on the left, i.e., we only know that the event occurred before a certain time; and *interval* censoring, when the interval is closed, i.e., we know that the event occurred in between two time points. Right censoring is certainly the most common, and is the only one that is relevant to algorithm performance modeling, so in the following we will limit to this case.

time: in this case the number of observed failures is a random variable, related to the lifetime distribution of the bulbs. This practice is termed *type I* censored sampling. As an alternative, the experiment could end as soon as a predetermined number of bulbs has failed (*type II* censored sampling): in this case, the duration of the experiment is a random variable.

In both cases, only a lower bound on failure time is available for the surviving bulbs. Unless the engineer is willing to wait for years, (or the new product is quite cheap), this incomplete data will constitute a large portion of the collected sample, and will clearly have an impact on the precision of the model. In other words, there is a *trade-off* between the duration of a sampling experiment, and the precision of the obtained model. In this sense, the advantage of type II censoring is that the portion of sample which will be censored is set in advance.

In algorithm performance modeling, type I censoring is typically performed, imposing a threshold on runtime. Also in this case, censoring can be viewed as a tool to limit the duration of a runtime sampling experiment, at the cost of decreasing model precision.

Discarding incomplete data can result in an biased model, even when such data constitutes only a small portion of the sample. Appropriate techniques have therefore been developed to take censored observations into account, as we will see in the next section.

3.3 Estimation in survival analysis

In this section we consider the problem of estimating the survival distribution of an event, based on experimental data. The object of the estimation is one of the functions describing the distribution. In *parametric* methods, this amounts to estimating the parameters of a function; in *non-parametric* methods, the function itself is expressed based on the available data. The functions typically estimated in the two cases are the pdf and the hazard, respectively.

A sample of censored survival data is usually represented as a set of realizations $D = \{(t_1, c_1), \ldots, (t_n, c_n)\}$ of a pair of random variables (T, C), generated from another pair of *latent* random variables, the time to the actual event T_e, and the time to censoring T_c. For each realization in the sample, only the minimum of these two variables $T = \min\{T_e, T_c\}$ is observed, along with the *event indicator* $C = I(T_e \leq T_c)$, which is 1 if the event of interest was observed, 0 if it was censored. Most estimation techniques require the time to event T_e and the time to censoring T_c to be statistically independent (*uninformative* censoring).

The following two subsections describe parametric and non-parametric estimation from censored data. In Section 3.4 we discuss regression models, where each observation is related to a vector of covariates.

3.3.1 Parametric methods

In this class of methods, a function $f(t; \boldsymbol{\theta})$ is assumed to represent the pdf of the time-to-event t. The choice of a particular distribution is a crucial one, and should be based on empirical evidence: both graphical and statistical tests are available to evaluate a parametric model [Klein and Moeschberger, 2003, Ch. 12]. A well-known example is the χ^2 statistics, which can be measured dividing the *uncensored* data into m bins, and comparing the number of samples o_i in each bin to the one e_i predicted by the fitted distribution, according to the formula

$$\chi^2 = \sum_i [(o_i - e_i)/e_i]^2. \tag{3.7}$$

A high value indicates a poor fit: the model passes the test with confidence α if χ^2 is lower than the $1-\alpha$ quantile of the χ^2 distribution with $m-k$ degrees of freedom, k being the number of parameters in the model. Frost et al. [1997] apply this test to parametric models of runtime distributions, dividing the data into equally sized bins on a logarithmic scale.

The parameter $\boldsymbol{\theta}$ is estimated based on the data. Various approaches can be followed in this sense, usually based on the concept of the *likelihood* of the parameter in light of the data, that is, $L(\boldsymbol{\theta}; D) = \Pr\{D \mid \boldsymbol{\theta}\}$. In a *frequentist* context, *maximum likelihood* methods estimate the parameter to be the one that maximizes the likelihood. In *Bayesian* approaches [Bishop, 1995; MacKay, 2003], the parameter value is assumed to be drawn from a *prior* distribution $\Pr\{\boldsymbol{\theta}\}$, and inference is based on the *posterior* probability $\Pr\{\boldsymbol{\theta} \mid D\} \propto \Pr\{D \mid \boldsymbol{\theta}\}\Pr\{\boldsymbol{\theta}\}$, either considering its mode as point estimate of $\boldsymbol{\theta}$, or evaluating *credible intervals* based on the posterior. As both approaches require computing the likelihood, in the following we illustrate how this can be done in the case of censored survival data [Klein and Moeschberger, 2003].

If $g(.)$ and $G(.)$ are respectively the pdf and the CDF of the censoring times, the contribution of a non-censored observation $(t_i, c_i = 1)$ to the likelihood is

$$L(\boldsymbol{\theta}; t_i, c_i = 1) = (1 - G(t_i))f(t_i; \boldsymbol{\theta}), \tag{3.8}$$

while the contribution of a censored observation $(t_i, c_i = 0)$ is

$$L(\boldsymbol{\theta}; t_i, c_i = 0) = (1 - F(t_i; \boldsymbol{\theta}))g(t_i). \tag{3.9}$$

A sample of size n is a combination of censored and non-censored observations, and, assuming independence among the n realizations of (T, C), the likelihood will be given by

$$L(\boldsymbol{\theta}; D) = \prod_{i=1}^{n} [(1 - G(t_i))f(t_i; \boldsymbol{\theta})]^{c_i} [(1 - F(t_i; \boldsymbol{\theta}))g(t_i)]^{1-c_i}. \tag{3.10}$$

If we assume *independent* censoring [Liang et al., 1995; Fleming and Harrington, 1991], i.e. if we assume that the distribution of the censoring times does not depend on the parameters influencing the survival times distribution, the factors $(1 - G(t_i))$ and $g(t_i)$ are not informative for the inference on the parameters and can be removed from the likelihood:

$$L(\boldsymbol{\theta}; D) \propto \prod_{i=1}^{n} f(t_i; \boldsymbol{\theta})^{c_i} (1 - F(t_i; \boldsymbol{\theta}))^{1-c_i}. \tag{3.11}$$

To illustrate these concepts, Figure 3.1 displays the likelihood of two events, one censored and one uncensored, plotted against the pdf.

We have already seen examples of parametric distributions which can be used to model time-to-event data (Sec. 2.4.1) as the Weibull (2.9) or the lognormal distribution (2.10). None of these can be used to model heavy-tailed distributions (Sec. 2.4.1). Such distributions have important applications in various other fields, as economics. Reed and Jorgensen [2004] introduces parametric formulas for modeling them. The double Pareto-lognormal distribution (DPLN), which will be used in one of our experiments, has pdf

$$\begin{aligned} f(t|\alpha, \beta, \nu, \tau) &= \frac{\alpha\beta}{\alpha+\beta}[A(\alpha, \nu, \tau)t^{-\alpha-1}\Phi(\frac{\log t - \nu}{\tau} - \alpha\tau) + \\ &+ A(-\beta, \nu, \tau)t^{\beta-1}\Phi^c(\frac{\log t - \nu}{\tau} + \beta\tau)] \end{aligned} \tag{3.12}$$

where $A(\gamma, \nu, \tau) = \exp(\gamma\nu + \gamma^2\tau^2/2)$, $\Phi(t)$ is the CDF of the standard normal distribution with mean 0 and variance 1, and $\Phi^c(t) = 1 - \Phi(t)$ is the corresponding complementary CDF. The DPLN describes the distribution of the product of two independent random variables, one with lognormal distribution (2.10), one with Double Pareto distribution, whose pdf can be written as

$$f(t; \alpha, \beta) = \begin{cases} \gamma t^{\alpha-1}, & t \geq 1, \\ \gamma t^{\beta-1}, & 0 \leq t < 1, \end{cases} \tag{3.13}$$

with $\gamma = \alpha\beta/(\alpha+\beta)$. The DPLN can fit both lognormal and heavy tailed distributions, depending on its parameters.

3.3.2 Non-parametric methods

In non-parametric methods, no assumption is made on the distribution of the survival times, and estimates are based solely on the data observed. If there is no censored data, a simple non-parametric estimate of the survival function can be obtained based on the empirical estimate of the CDF

$$\hat{F}(t) = \sum_{i:t_i \leq t} \frac{1}{n} \tag{3.14}$$

where $t_1 < t_2 < ... < t_n$ are the ordered survival times observed.

In the presence of censoring, the most commonly used non-parametric estimator of the survival function is the *product-limit* estimate [Kaplan and Meyer, 1958]. This method is based on the idea of estimating the hazard at each time t_i in the sample as the portion of patients still alive, or "at risk" (in our case: the algorithms still running), experiencing an event at this time point:

$$\hat{h}(t_i) = \frac{\sum_{i:t_j=t_i, c_j=1} 1}{\sum_{i:t_j \geq t_i} 1} = \frac{d_i}{n_i}, \tag{3.15}$$

where c_i is the event indicator, d_i is then the number of events observed at time t_i, and n_i is the number of observations still at risk at time t_i. An estimate of the survival function based on (3.15) is given by the *product limit* estimator, also known as Kaplan-Meier (KM) estimator [Kaplan and Meyer, 1958]:

$$\hat{S}(t) = \prod_{i:t_i \leq t} (1 - \hat{h}(t_i)) = \prod_{i:t_i \leq t} (1 - \frac{d_i}{n_i}), \tag{3.16}$$

from which one can estimate the CDF as $\hat{F}(t) = 1 - \hat{S}(t)$. The cumulative hazard can be obtained from (3.16) using the relation $\hat{H}(t) = -\log(\hat{S}(t))$, or based directly on (3.15), as proposed by Nelson [1972]; Aalen [1978]: $\hat{H}(t) = \sum_{i:t_i<t} d_i/n_i$. The two estimators are similar for large sample sizes.

The functions estimated by these and many other nonparametric methods are discrete, as in (3.4,3.5,3.6): $\hat{F}(t)$, $\hat{S}(t)$ are "stepwise" functions, whose values change only at uncensored observations $\{t_i \mid c_i = 1\}$, and are defined until the largest one t_u; if there is a larger *censored* observation t_c, the estimated $\hat{F}(t_u)$ will be < 1. For our purpose, we will consider it improper, with $\hat{F}(\infty) = \hat{F}(t_u)$, even though the actual distribution being estimated may be proper. The hazard estimate $\hat{h}(t)$ (3.15) is discrete, defined only at the event times $\{t_i\}$ in the sample D. In order to obtain meaningful predictions also for other values of t, hazard estimates can be

smoothed [Wang, 2005]. Plots presented in the previous chapter, as all other RTD plots in this thesis, are obtained from product-limit estimates (3.16), which are correctly displayed as stepwise functions, constant among uncensored observations.

It is well known that, when the proposed parametric distribution for runtimes is appropriate, parametric inference is more efficient than non-parametric [Pintilie, 2006], meaning that the resulting model will converge faster to the underlying distribution as the sample increases. Conversely, the use of a parametric function which does not fit the data well can produce an inefficient model.

Recently, interesting research on nonparametric estimation for censored survival data has been carried out in a Bayesian framework: a review can be found in [Ibrahim et al., 2001].

3.4 Regression models

It is often of interest to investigate the relationship between a time to event distribution and some *covariates* $\mathbf{x} \in \mathbb{R}^d$ linked to the observations, estimating a conditional model, or *regression* model, of one of the functions representing the RTD, usually the hazard $h(t \mid \mathbf{x})$. In this case, each observation in the sample is a triple $(\mathbf{x}_i, t_i, c_i)$. For example, in a medical study, each patient may be characterized by different discrete of or continuous values, such as its sex, age, weight, etc..

In our case, consider again the RTD of a set of instances, and the RTDs of each instance in the set, which will in general be different from the RTD of the set. If the features of an instance are related to the runtime of the algorithm, we can use them as covariates, to obtain estimates that are closer to the actual RTD of each instance. Note that this is essential to perform per instance allocation. In another situation, one might want to study the impact of the various parameters of an algorithm on its runtime distribution: in this case a sample could be formed using different values of the parameters, and the parameters could be used as covariates.

A parametric regression model can be based on a pdf $f(t \mid \boldsymbol{\theta})$ in which the parameter is itself a parametric function of the covariate $\boldsymbol{\theta}(\mathbf{x}; \boldsymbol{\beta})$ [see, e.g., Bishop, 1995, par 6.4]: for example, an exponential distribution $f(t; \theta) = \theta \exp(-\theta t)$ can be cascaded with a linear model $\theta(\mathbf{x}; \boldsymbol{\beta}) = \boldsymbol{\beta} \cdot \mathbf{x}$, obtaining a parametric conditional model:

$$f(t \mid x; \boldsymbol{\beta}) = \theta(\mathbf{x}; \boldsymbol{\beta}) \exp(-\theta(\mathbf{x}; \boldsymbol{\beta}) t) = (\boldsymbol{\beta} \cdot \mathbf{x}) \exp(-\boldsymbol{\beta} \cdot \mathbf{x}). \tag{3.17}$$

This approach was used in an earlier version of our time allocator [Gagliolo and Schmidhuber, 2005, 2006a], but was later abandoned as we did not find a parametric form $f(t; \boldsymbol{\theta})$ which could fit the data well.

Many regression methods are *semi-parametric*, as they involve both parametric and non-parametric terms: the most popular one is the *proportional hazards* model, or Cox model [Cox, 1972]:

$$h(t \mid \mathbf{x}) = h_0(t) \exp(\boldsymbol{\beta} \cdot \mathbf{x}). \tag{3.18}$$

Also in this case, the parameter of the model is itself a parametric function of the covariates, whose weights $\boldsymbol{\beta}$ are optimized based on the sample. The covariates act multiplicatively on a baseline non-parametric hazard $h_0(t)$, which can be estimated as well, or left unspecified. Other examples include the additive risk model [Klein and Moeschberger, 2003, Chapter 10]; [Martinussen and Scheike, 2006, Chapter 5], in which the covariates act additively on the hazard

function; and the *accelerated failure time* model [Cox and Oakes, 1984], in which the covariates are related to the logarithm of survival times.

Only limited work has been done on fully non-parametric regression models [see reviews by Van Keilegom et al., 2001; Spierdijk, 2005]. Beran [1981] and later works [e.g., Akritas, 1994] are based on the idea of estimating a conditional hazard function, as in (3.15), but with 1 replaced by a *kernel* function $K(\mathbf{x}, \mathbf{y}) : \mathbb{R}^d \times \mathbb{R}^d \to [0, \infty)$, representing a similarity measure in covariate space: a prediction of the hazard at time t for an individual with covariate $\mathbf{x}$ is given by

$$\hat{h}(t \mid \mathbf{x}) = \frac{\sum_{i:t_i=t,c_i=1} K(\mathbf{x}, \mathbf{x}_i)}{\sum_{j:t_j \geq t} K(\mathbf{x}, \mathbf{x}_j)}. \tag{3.19}$$

In short, (3.19) performs a nearest neighbor estimate of the hazard. An example of a non-parametric model of this form, which will be used in our experiments, is proposed by Wichert and Wilke [2005]. In this model, the kernel does not measure similarity in covariate space, but in the cumulative distribution of the covariates. For scalar covariates ($d = 1$), an empirical CDF (3.14) $F_x(x)$ of the x_i in the sample is first evaluated. In order to predict the hazard function for an unseen value x of the covariate, its CDF value $F_x(x)$, is estimated, by evaluating its rank in the sorted list of x_i, and compared to the $F_x(x_i)$ of each observation, via a scalar kernel function K_1, with bandwidth parameter b_n. The value of $K_1((F_x(x) - F_x(x_i))/b_n)$ is then used to weight the event t_i. The resulting estimate of the hazard for an individual with scalar covariate x is

$$\hat{h}(t|x) = \frac{\sum_{t_i=t,v_i=1} K_1(\frac{F(x)-F(x_i)}{b_n})}{\sum_{t_i \geq t} K_1(\frac{F(x)-F(x_i)}{b_n})}. \tag{3.20}$$

If the covariates are multidimensional, $\mathbf{x} = (x^{(1)}, \ldots, x^{(d)})$, $d > 1$, a *product kernel* is used: the kernel distances are measured for each dimension, and their product is used as weight:

$$K(\mathbf{x}, \mathbf{x}_i) = \prod_{j=1}^{d} K_1(\frac{F(x^{(j)}) - F(x_i^{(j)})}{b_n}). \tag{3.21}$$

An advantage of this kernel is that it measures a distance on the *distribution* of covariate values, so it is not sensitive to scaling. The kernel function K_1 is required to be symmetric around 0, and integrate to 1. A uniform kernel (0.5 on $[-1, 1]$, and 0 elsewhere) is a common choice in non-parametric statistics. The convergence proof for the estimator requires the bandwidth parameter b_n to be set based on the size n of the sample, as $b_n \in [n^{-1/2}, n^{-1/4}]$. This method, as others based on a product kernel, suffers from a curse of dimensionality: when d is high, the kernel tends to vanish at all $\mathbf{x}$ which are not present in the sample.

In general, the precision of any regression model will depend on the number of covariates d. This dependence is more intuitive if we look at (3.19), but similar considerations hold for (semi-)parametric models. Consider a runtime sample collected for an algorithm a, using a to solve each b_m in set of instances $\mathcal{B}_0 = \{b_m\}$ many times, with different random seeds. Using no covariates at all corresponds to considering all events as being sampled from the same distribution, so it would only allow to estimate the RTD of a on the set $\mathcal{B}_0$. Using instance features $\mathbf{x}_m \in \mathbb{R}^d$, one can learn a regression model (3.19), and in principle use it to recover the RTD on the single instances: if d is large enough, and $\mathbf{x}$ varies such that $\mathbf{x}_j$ is quite far from $\mathbf{x}_k$ for two different instances b_j, b_k, then $K(\mathbf{x}_j, \mathbf{x}_k)$ will be 0 for all $j \neq k$, and (3.19) conditioned on $\mathbf{x}_j$ will

effectively select only samples which were obtained solving b_j. The problem with such a precise model arises when using it to predict the runtime on an unseen instance b_u, with features $\mathbf{x}_u$: likely, also $K(\mathbf{x}_u, \mathbf{x}_k)$ will be 0 for all k, so the model's prediction would be that a will *never* solve b_u. In machine learning terminology, this is an example of *overfitting*, where the model cannot generalize.

3.5 Time-varying covariates

In the conditional models described above, the covariates are constant over time. In some situations, it might be interesting to model the effect of covariates which vary over time. In a medical setting, measures such as the amount of cholesterol in the blood may be taken on each individual at regular intervals, and related to the lifetime. In our case, *dynamic* information about the algorithm could be available, for example the state of some internal variables. One possibility is to consider it as a *time-varying* covariate [Li and Doss, 1995; Nielsen and Linton, 1995]; in order to do so, the time-dependent distribution of the covariate has to be modeled using *longitudinal data* analysis [Fitzmaurice et al., 2008], which can also be used to describe the performance of optimization algorithms, modeling the joint distribution of runtime and solution quality [Gagliolo et al., 2009].

When the aim of modeling is time allocation, a potential advantage of conditioning on time-varying covariates is that it allows to update the allocation dynamically, improving over static allocation [Sayag et al., 2006]. A potential disadvantage is related to prediction: adding time-varying covariates may result in the same kind of overfitting described at the end of the previous section.

Indeed, the simplest time-varying covariate is time itself: if an algorithm is still running at a time y, the CDF F of the RTD for the rest of the run can be evaluated by simply shifting and scaling the original F

$$F(t|T > y) = \frac{F(t) - F(y)}{1 - F(y)} = \frac{S(y) - S(t)}{S(y)}, \tag{3.22}$$

defined only for $t > y$. Given the definition (3.1) the hazard function for $t > y$ does not change; in the non-parametric setting, it can be evaluated by simply discarding hazard values $h(t_i)$ with $t_i \leq y$.

3.6 Algorithms as competing risks

In failure analysis, type II censored sampling is often used to reduce the duration of an experiment, ending it when a certain number of uncensored values have been collected. The same approach can in principle be applied when sampling algorithm performance, running the algorithms in parallel in a uniform portfolio (Sect 2.4.3), and using the runtime of faster algorithms to censor the runtime of slower ones: instead of solving an instance b with all N algorithms, one could wait until the first $N_C < N$ solve it, and abort the remaining $N - N_C$, recording their runtimes as censored. This could allow to reduce the computational cost of learning performance models, an issue which is common to all non-oblivious allocators (see Sec. 2.7, and Sect 6.2).

Unfortunately, while the lifetimes of different light-bulbs are not correlated, runtimes of algorithms solving the *same* problem instance may well be. All modeling methods we have seen

in Section 3.3 are based on the assumption that censoring time and event time are independent: collecting a sample in this way may then induce a bias in the obtained model [Tsiatis, 1975; Putter et al., 2006]. In practice, the survival function of the slower algorithms will be overestimated.

A similar situation arises in survival analysis when patients under study for a certain condition die from a *different* condition. For example, in a medical setting, one might want to study the distribution of time to death T_A, due to a particular disease or condition A, and gather a sample of individuals affected by it: at the end of the study, some patients will have died of condition A, and their lifetime recorded as uncensored, while others will be still alive, and their time to death censored. But some of the patients might have died from a different condition, B. If we want to estimate the distribution of time to death of A, we may view death of B as a censoring event, as it prevented to know what T_A *would* have been if the patient did not die of B at a time $T_B < T_A$. Death from B is termed a *competing risk* [Pintilie, 2006; Putter et al., 2006].

As death of B is an event that we are not interested in modeling, one possibility is to consider it as an additional censoring mechanism. This may pose a problem when the distribution of the event of interest and the one of the competing risk are not independent, for example if both conditions A and B become more likely with old age. This would obviously result in a censoring mechanism that is not independent from the distribution of time to events, thus not satisfying assumption on which most estimators rely. More details on this topic are reported in Appendix B; a complete textbook on the topic is [Pintilie, 2006].

3.7 Discussion

In this chapter we considered the problem of estimating the runtime distribution (RTD) of a generalized Las Vegas algorithm (gLVA), such as a complete or incomplete decision problem solver, or a solver for the search formulation of an optimization problem. The RTD is the most general model for this class of algorithms, as it can be used to evaluate all relevant performance statistics: expected runtime, higher moments, quantiles, tail probabilities, etc. It can also be used to describe the performance of incomplete solvers, whose probability of solving an instance is less than unity. Some important phenomena, as the heavy tailed behavior of backtracking search solvers on certain instances, can only be captured studying the RTD.

To this aim, we reviewed some basic concepts of survival analysis, a branch of statistics which studies the distribution of time-to-event data, and can therefore be applied to study the RTD of gLVAs (Sec. 3.1). As discussed in Section 2.7, our main goal is to estimate the instance RTD of an algorithm, in order to be able to implement portfolios and restarts. In existing work on single algorithm selection (see Sec. 2.3), the runtime on a given instance is predicted conditioning a scalar regression model on instance features. In Section 3.4 we learned that estimation of time-to-event distributions can be conditioned on a vector of covariates, associated to each observation. Using instance features as covariates would therefore allow to estimate the instance RTD of an algorithm, based on the results on a set of experiments. Moreover, such estimate can be updated at runtime, conditioning on time-varying covariates, or simply on the time spent so far (Sec. 3.5): this could be exploited to implement a dynamic time allocator.

One of the requirements for our modeling method is that it should have a low computational cost, as our overall goal is to minimize runtime. An advantage of survival analysis in this sense is that it allows to take into account the information conveyed by a partial run of an algorithm, which was aborted before converging to a solution. The use of such *censored*

information (Sec. 3.2) is indeed the distinguishing feature of this branch of statistics, and was described in Section 3.3. Traditional scalar regression cannot deal with censored data: model based algorithm selection therefore requires to solve each training instance with each available algorithm, in order to obtain the exact runtimes. Estimating the whole RTD allows instead to correctly account for incomplete runs, therefore saving some time. Intuitively, there is a trade-off among sampling time and model precision: censoring an experiment earlier allows to reduce its duration, but this can result in poorer predictions.

In algorithm portfolios, as soon as one of the algorithms solves the current instance, the remaining ones are also stopped. Their runtimes can be considered as censored observations. In this case, the fastest algorithm is acting as a censoring mechanism. In survival analysis terminology, the algorithms act as competing risks (Sec. 3.6). Sampling runtime data in this way has the disadvantage of inducing a bias in the resulting RTD estimators.

In Chapter 6 we will apply the methods presented here to sample and estimate RTDs efficiently. One issue that remains open is the aforementioned trade-off among sampling time and model precision. In the next chapter, we will describe a framework for online learning which can be used to solve this trade-off.

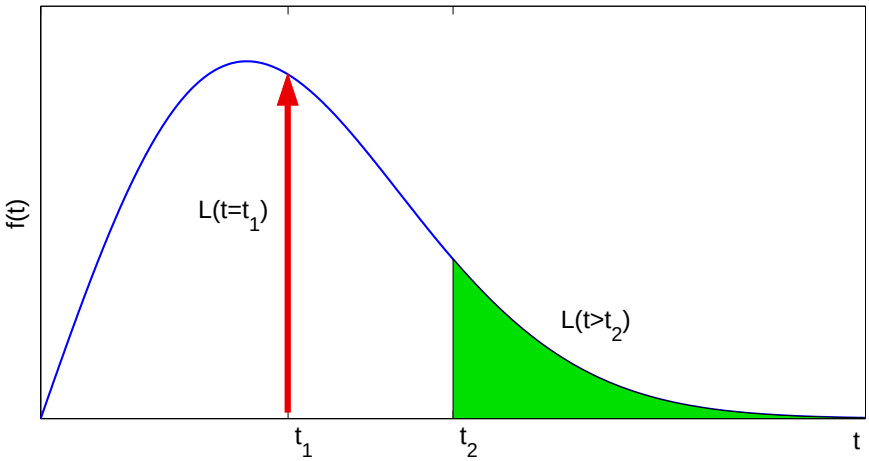


Figure 3.1. The contribution of a censored and an uncensored event to the likelihood of the parameters of a Rayleigh distribution. The function displayed is the pdf $f(t;\boldsymbol{\theta})$: the likelihood of an uncensored event at t_1 is the corresponding value of $f(t_1)$, while that of a censored event at t_2 is the integral of the remaining portion of the pdf, corresponding to $S(t_2)$

Chapter 4

Bandit problems

The problem of repeated decision making has been extensively studied in the *online* setting [see e.g. Sutton and Barto, 1998; Vovk et al., 2005; Cesa-Bianchi and Lugosi, 2006]. In the *multi-armed bandit problem*, a discrete set of K alternatives has to be explored, during a sequence of trials. A each trial, one of the alternatives is chosen, and the corresponding outcome is observed. The problem consists in optimizing the total value of the outcomes observed during the whole sequence. This well known paradigm was originally studied in the context of sequential experiment design [Robbins, 1952]: a typical example application is in the medical domain, where a sequence of patients can be assigned to one of a set of alternative therapies, based on results observed so far. Intuitively, the algorithm selection problem can be formulated as a bandit problem in an analogous manner, viewing each instance of a problem as a "patient", and each algorithm as a "therapy". When selection is aimed at minimizing runtime, this value may be considered as the outcome of each "therapy", to be minimized.

Several different variations of the bandit problem have been addressed, characterized by different hypothesis on the processes generating the outcomes, on their observability, etc. Research on this topic is usually not supported experimentally: the performance of bandit problem solvers is instead assessed based on probabilistic or exact *bounds* on their performance, relative to the performance of each arm, which make experimental comparison less relevant. In this chapter, we present the basic version of the problem (Sect. 4.1), along with a more recent version where worst-case hypothesis are made on the process generating the outcomes (Sect. 4.2). Two solvers are described in more detail, one for each section, as they will be used in our experiments. Outcomes of a trial are usually assumed to lay in a closed interval. This may pose a problem when viewing runtimes as outcomes, as we have seen that their values may vary across several orders of magnitude (see Sect. 2.4.1). When no prior information is available on the range of the outcomes, it is still possible to derive bounds on performance based on a simple doubling trick (Section 4.3). The last two sections discuss applications in the time allocation domain. In Section 4.4, we discuss the "naive" formulation of per set algorithm selection sketched above. In Section 4.5 we describe related work where time allocation involves solving a bandit problem of some sort. Section 4.6 discusses the chapter.

4.1 Stochastic bandits

In its basic version, the *multi-armed bandit* problem is faced by a gambler, playing a sequence of trials on a K-armed slot machine[1]. At the i-th trial, the gambler chooses one of the available arms, $k \in \{1, \ldots, K\}$, and incurs in the corresponding loss $l_k(i)$. The aim of the game is to minimize the cumulative loss: an alternative formulation can be made, where positive *payoffs* are received at each trial, whose sum has to be maximized.

This simple setting is also called *partial information* setting: its difficulty resides in the fact that only the loss corresponding to the chosen arm can be observed. In the *full information* setting, the loss of each arm is revealed at each trial, and the problem becomes easier.

A game of M trials can be described by a sequence of pairs $\{(k(i), l_{k(i)}(i))\}$, $i = 1, \ldots, M$. A bandit problem solver (BPS) can be represented as a mapping from the game up to the $(i-1)$-th trial, to a probability distribution $\mathbf{p}(i) = (p_1(i), ..., p_K(i))$, over the arms, based on which the choice for the i-th trial will be made.

As the actual losses $l_k(i)$ are only available for rounds at which k was pulled, comparing the cumulative loss of the gambler to the cumulative loss of each arm does not tell much about what the loss *could* have been. For this reason, the performance is described in terms of the *regret* R, defined as the difference between the cumulative loss L of the gambler, and the one L^* of the best arm: after M trials, R amounts to

$$R(M) = L(M) - L^*(M) = \sum_{i=1}^{M} l_{k(i)}(i) - \min_k \sum_{i=1}^{M} l_k(i). \tag{4.1}$$

Note that the regret (4.1) cannot be computed in the partial information setting. What is normally evaluated is the expected value of (4.1), or some upper bound which holds with a desired probability, where the expectation is w.r.t. the process picking one of the arms with probabilities $\mathbf{p}$. A solver is termed *Hannan consistent* if $E\{R(M)\}/M$ goes to 0 in the limit when the number of trials M goes to infinity.

A simple example of BPS is ϵ-GREEDY (Alg. 1) [see, e.g., Sutton and Barto, 1998, Ch. 2]. This solver keeps an estimate of the average loss of each arm $\hat{l}_k$. At every trial, with probability ϵ, the solver pulls an arm at random, according to a uniform distribution; with probability $(1-\epsilon)$ it pulls the arm with the lowest estimated $\hat{l}_k$. In other words, it explores with probability ϵ, and greedily exploits the estimated best arm otherwise[2], hence the name.

Algorithm 1 ϵ-GREEDY (ϵ), with parameter $\epsilon \in (0,1)$. k^* is the current greedy action, $\mathbb{I}(k = k^*)$ is 1 for $k = k^*$, 0 otherwise.

initialize $\hat{l}_k := 0$, $L_k := 0$, $n_k := 0$, for $k = 1, ..., K$;
for each trial i **do**
  $k^* := \arg\min_k \hat{l}_k$
  pick arm k with probability $p_k := (1-\epsilon)\mathbb{I}(k = k^*) + \frac{\epsilon}{K}$
  observe loss $l_k(i)$
  update $L_k := L_k + l_k(i)$, $n_k := n_k + 1$, $\hat{l}_k := \frac{L_k}{n_k}$
end for

[1] The name comes from use of the slang term "one-armed bandit" to refer to slot machines.
[2] If more arms have the same $\hat{l}_k$, one is picked at random among them.

If the l_k are sampled independently from K *stationary* distributions, ϵ-GREEDY will eventually try all arms enough times to have good estimates $\hat{l_k}$, and start playing the best arm with probability $(1-\epsilon)$: decreasing ϵ with i allows to achieve Hannan consistency. Analogous considerations hold if the distributions are initially *non-stationary*, but gradually stabilize, becoming stationary. If instead the distributions of l_k remain non-stationary, the estimated losses $\hat{l_k}$ should be updated using a decay factor, in order to "forget" about old losses.

Rather than looking at improved versions of ϵ-GREEDY, in the next section we will consider a game in a much more pessimistic setting, and describe a solver which achieves not only Hannan consistency, but also bounds on the expected regret, which hold for an arbitrary number of trials.

4.2 Non-stochastic bandits

In the *non-stochastic* bandit problem [Baños, 1968], no statistical assumptions are made about the process generating the losses, which are allowed to be an arbitrary function of the entire history of the game. This includes the *non-oblivious* adversarial setting, where the losses are set by an adversary whose aim is to deceive the gambler. The only obvious restriction is that the losses have to be set for each trial before the gambler makes his choice. Additionally, nearly all solvers for this worst-case setting assume losses to be bounded, $l_k \in [0,1]$.

Auer et al. [1995, 2002] propose several algorithms (for the *payoff* version of the game), where the expected regret on a finite number of trials M is bounded by a term $O(\sqrt{MK\log K})$. Cesa-Bianchi and Lugosi [2006] describe a version for the loss game, whose bound is $O(\sqrt[3]{(MK)^2\log K})$. These and other solvers from the same authors are modified versions of the *weighted majority* algorithm [Littlestone and Warmuth, 1994], a solver for the full information game. In the partial information setting, the cumulative losses for each arm are obtained through an *unbiased estimate*, whose aim is to correct for the missing information: for a given trial, and a given arm with loss l, whose probability of being pulled is 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: $E\{\tilde{l}\} = pl/p + (1-p)0 = l$.

EXP3LIGHT [Cesa-Bianchi, Mansour and Stoltz, 2005, Sec. 4] is an example of a solver based on unbiased estimates. This solver subdivides the game 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:

$$\eta_r = \sqrt{\frac{2(\log K + K\log M)}{(K4^r)}} \tag{4.2}$$

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

Algorithm 2 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) indicates the value of a quantity used or observed at trial $i \in \{1,\ldots,M\}$; k indicate 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 [Cesa-Bianchi, Mansour and Stoltz, 2005] proves the following bound on the expected regret of EXP3LIGHT, which holds if η_r is updated according to (4.2):

$$\begin{aligned} & E\{L_E(M)\} - L^*(M) \leq \\ & 2\sqrt{2(\log K + K\log M)K(1+3L^*(M))} \\ + \quad & (2K+1)(1+\log_4(3M+1)). \end{aligned} \tag{4.3}$$

Algorithm 2 EXP3LIGHT (K, M).

K arms, M trials;
losses $l_k(i) \in [0,1]\ \forall\ i = 1,...,M$, $k = 1,\dots,K$;
initialize epoch $r := 0$, $L_E(0) := 0$, $\tilde{L}_k(0) := 0$ for $k = 1,\dots,K$;
initialize η_r according to (4.2).
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 $k = 1,\dots,K$,
  $\tilde{L}^*(i) := min_k\{\tilde{L}_k(i)\}$;
  if $(\tilde{L}^*(i)) > 4^r$ **then**
    start next epoch $r := \lceil \log_4(\tilde{L}^*(i)) \rceil$;
    update η_r according to (4.2).
  end if
end for

4.3 Unbounded losses

EXP3LIGHT and similar solvers require the losses to be in $[0,1]$. In practice, these bounds may be scaled to $[0,\mathcal{L}]$, to account for an arbitrary maximum loss $\mathcal{L}$, provided that such maximum is known in advance. When working with algorithm runtimes, it is not easy to guess a maximum runtime, as we have seen (Sect. 2.4.1). In principle, one could use an arbitrarily large $\mathcal{L}$, e.g. one year. This would not in general affect the bound on the regret, but it would unfortunately impact the practical performance of the solver, as the observed losses would look too small to allow to quickly discriminate the best arm. An alternative would be to choose a small $\mathcal{L}$ and "censor" larger losses, but this would obviously invalidate any bounds on the actual runtime of the method.

Some interesting results regarding games with *unbounded* losses have recently been obtained. Cesa-Bianchi, Mansour and Stoltz [2005]; Cesa-Bianchi et al. [2007] consider a full information game, and provide two algorithms which can adapt to unknown bounds on signed payoffs. The algorithms are based on a simple doubling trick, similar to the one used in Alg. 2 to estimate the optimal loss: the game is divided in a sequence of rounds $u = 1,2,\dots$ with $\mathcal{L} = 4^u$, and each time a loss larger than 4^u is observed, the BPS is reinitialized, and a new round starts.

Based on this work, Allenberg et al. [2006] provide a Hannan consistent algorithm for losses whose bound grows in the number of trials i with a known rate i^{ν}, $\nu < 1/2$.

In Section 7.5, we will modify Alg. 2 based on the doubling trick described above, in order to deal with unbounded losses. In the next section, we propose a simple framework for per set algorithm selection, based on a BPS, discussing its limitations.

4.4 Algorithm selection as a bandit problem

Let us now see how to represent selection among Las Vegas algorithms as a bandit problem. Consider a sequence $\mathcal{B} = \{b_1, \ldots, b_M\}$ of M instances of a problem, for which we want to minimize solution time, and a set of N Las Vegas algorithms $\mathcal{A} = \{a_1, \ldots, a_N\}$, such that by definition each b_m can be solved in a finite time by *each* a_k. In the following, BPS (K, M) will indicate a bandit problem solver whose parameters have been initialized to minimize regret on a game of M trials, playing on a bandit with K arms.

It is straightforward to describe static algorithm selection in a multi-armed bandit setting, where "pick arm k" means "run algorithm a_k on next problem instance" (GAMBLEAS, Algorithm 3). 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* (see Sect. 4.2).

Algorithm 3 GAMBLEAS $(\mathcal{A}, BPS)$ Gambling Algorithm Selection.

Algorithm set $\mathcal{A}$ with N algorithms;
A bandit problem solver BPS
M problem instances.

initialize BPS (N, M)
for each problem $b_m, m = 1, \ldots, M$ **do**
  pick algorithm $I(m) = k$ with probability $p_k(m)$ from BPS.
  solve problem b_m using algorithm $a_{I(m)}$
  incur loss $l_{I(m)} = t_{I(m)}(m)$
  update BPS
end for

As a BPS minimizes the regret with respect to the best single arm, based on the whole set of trials, this approach would allow to implement *per set* selection, of the overall best algorithm. Unfortunately, as we have seen, per set selection is only profitable if one of the algorithms dominates the others on all problem instances (Sect. 2.3), which is often not the case.

In the next section, we will see other related work where a BPS, or similar methods, are used for performing time allocation.

4.5 Applications to algorithm selection

The *Max K*-armed bandit problem [Cicirello and Smith, 2005] is a variant of the game where the aim is to maximize, rather than the sum, the *maximum* of the payoffs achieved on all rounds. Solvers for this game are used in [Cicirello and Smith, 2005; Streeter and Smith, 2006] to implement oblivious per instance selection from a set of randomized optimization algorithms: multiple runs of the available solvers are allocated, to maximize solution quality on a single problem instance. In this case, each problem instance is solved multiple times, keeping only the best solution.

Carchrae and Beck [2005] propose the oblivious, dynamic, per-instance selection of a resource sharing portfolio, where machine time shares are based on a recency-weighted average of performance improvements. This latter technique is actually a simple solver for time-varying bandit problems [see, e.g., Sutton and Barto, 1998, Ch. 2]: it can be seen as a modification of ϵ-GREEDY (Alg. 1) for non-stationary problems, in which the average loss estimates are updated with a decay, in order to gradually forget old losses.

Maturana et al. [2009] use a BPS at each generation of a Genetic Algorithm, to select among alternative operators, implementing Adaptive Operator Selection. The fitness increment is considered as a reward.

In *racing* [Moore and Lee, 1994; Birattari et al., 2002; Birattari, 2004], the algorithm set contains different parametrizations of a given supervised algorithm, solving an optimization problem. Each alternative is repeatedly run on a sequence of increasingly large leave-one-out training sets, which can be seen as a sequence of related problem instances; after an instance is solved, badly performing algorithms are discarded if statistically sufficient evidence is gathered against them, such that machine time is shared among fewer algorithms on next instance. While the algorithms are running in parallel, the aim of racing is to select a single per set best algorithm, to be used on subsequent problem instances. In this case, per set selection is implicitly modeled as a stationary bandit problem with full information, where each alternative parameter setting corresponds to an arm, and each arm is tested at each trial. The distinguishing feature of this approach is that the number of arms is progressively reduced, based on bounds on the expected payoff, allowed by the stationarity assumption.

In Section 2.5 we described per set, offline time allocators, where the optimal schedule is based on a runtime sample, obtained solving a set of training instances. Streeter et al. [2007] propose a method where a "greedy" 4-optimal schedule is evaluated. 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 probabilistically, independently for each instance, based on a *label efficient forecaster* [Cesa-Bianchi, Lugosi and Stoltz, 2005], which allows to bound the regret compared to the offline greedy strategy, and whose complexity is also exponential in N. 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).

Streeter and Smith [2008] propose a different online method, where a per set optimal task switching schedule is learned while solving a sequence of instances. 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 by Auer et al. [2002]) picks one of the N algorithms, receiving a reward if the algorithm solves the instance: each trial ends when an instance is solved, or when the timeout value B is reached. 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. Streeter and Smith [2008] 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: in Section 9 we will compare our method on the same benchmarks.

4.6 Discussion

In this chapter, we described the multi-armed bandit problem, in the basic stationary setting (Section 4.1), and in the more general and pessimistic non-stochastic setting (Section 4.2). For each setting, we described one solver in more detail. With or without provable bounds on its (expected) regret, a bandit problem solver (BPS) performs online learning of the single best alternative, out of a discrete set of alternatives (the K arms), based on the results observed on a set of trials. In principle, such a learning device can be used to implement a per set method for single algorithm selection, where each trial consists in solving an instance with one of the algorithms, representing the arms (Section 4.4). Indeed, we are not aware of work on algorithm selection where a similar approach is proposed: recently, it has been considered as a comparison term by Streeter [2007].

In previous work where some technique for bandit problems has been applied to time allocation (Section 4.5), the result has either been an oblivious per instance, or non-oblivious per-set method. In both cases, the BPS learns to select the single best alternative performing a sequence of trials. In oblivious per instance methods, all trials refer to a single instance. In non-oblivious per set methods, each trial corresponds to an instance, and the resulting method is per set.

Recall the motivation behind our interest in bandit problem solvers (Sect. 2.7): we wanted to have the benefits of a non-oblivious time allocation method, using past experience in problem solving to reduce the cost of future problem solving, while avoiding a costly offline training phase. We looked at the task of devising an online method instead, where time allocation for each problem instance is based on results on previously solved instances. In Sect. 2.4 we have seen that models of the RTDs of the algorithms allow to allocate time in an optimal way; in Chapter 3 we have seen how such models can be learned, based on previous problem solving experiments. In principle, such models can be learned online, based on a runtime sample which is updated every time a problem is solved. From statistics, we know that the reliability of a model depends on the size of the sample used to estimate it. A model which is learn online will therefore be completely unreliable in the beginning, and gradually improve as the sample size increases. In this scenario, the online time allocation problem can be reduced to the following question: when can I start using the model? We can represent this question as a 2-armed bandit problem: in this case one of the arms would mean "use the model", the other one would mean "do not use the model". Each trial would correspond to a problem instance, as in GAMBLEAS (Section 4.4). The problem would be at least non-stationary, or even adversarial (Section 4.2),

depending on the hypotheses on the sequence of instances.

In the next part, we will present our contributions: the above idea of online time allocation as a bandit problem is developed in Chapter 7.

Part III

Contributions

Chapter 5

Dynamic algorithm portfolios

This part of the thesis contains our main contributions, organized in three chapters. This chapter is focused on static and dynamic allocation of one or more processors, based on the RTDs of the algorithms. Issues related to learning models of such RTDs, and using them to perform allocation, are discussed in the next chapter. The remaining one introduces a method for updating arbitrary time allocators online, and combines all contributions presenting two practical time allocators, a restart strategy (GAMBLER, Sec. 7.2), and an algorithm portfolio (GAMBLETA, Sec. 7.3).

Before proceeding, it will be useful to recall our remarks on related work, made at the end of Chapter 2, in the light of what we have learned about survival analysis in Chapter 3. Regarding single algorithm selection, we have seen that per instance selection is in principle more efficient than per set selection (Sec. 2.3). This consideration can be generalized to time allocation: for an arbitrary space of schedules, analog of (2.1) and (2.2) can be written, showing that the best per instance schedule can only improve over the best per set schedule.

Dynamic allocation is more general than static allocation: as proved by Sayag et al. [2006], it is also more efficient (Sec. 2.5). Model based allocation may be made dynamic if model predictions can be updated during algorithm execution: for RTD models, such update can be based on dynamic information about the current run, or simply on the time spent so far (Sec. 3.5). This means that the same RTD models can also be used to implement dynamic portfolios.

This chapter is organized as follows. In Section 5.1 we precise the time allocation problem considered in this thesis: given a set of algorithms, of unknown performance, use them to solve a set of instances, of unknown difficulty, with the aim of minimizing the computational effort. In the three following sections, we focus on allocation on a single instance, assuming the availability of RTD models. As seen in Section 2.4.3, given the RTD of each algorithm on the instance at hand, the RTD of the portfolio as a whole can be derived analytically, for an arbitrary allocation. This allows to formulate time allocation as an optimization problem, searching for the allocation which optimizes some quantity related to the RTD of the resulting portfolio, for example minimizing the expected time to solution. Section 5.2 introduces two additional criteria for static time allocation. Section 5.3 considers the allocation of multiple CPUs. Section 5.4 presents a simple way of turning a static allocator into a dynamic one, discussing its application to our RTD-based allocators. Section 5.5 compares our work with the related literature. Section 5.6 summarizes and discusses the chapter.

5.1 The time allocation problem

The time allocation scenario considered in this thesis consists in solving a sequence of instances of a problem, using a set of algorithms which do not interact.

The problem may be a decision, or search problem, but also a global optimization problem, in which the aim is to find the global optimum, or the decision version of an optimization problem, where a solution of a given quality has to be found. In general, we will consider all those problems for which the notion of algorithm runtime is well defined, and it makes sense to compare the performance of different algorithms on a same instance based only on their runtimes.

The meaning of "solution" will depend on the problem considered. A decision problem is solved when the correct "yes" or "no" answer is produced, and its correctness is proved. A search problem is solved when a single solution is found, or unsolvability is proved. A global optimization problem is solved when a feasible solution is found, and it is proved that no other feasible solutions have a better objective. We will refer to all these situations simply saying that "the problem is solved", and define the runtime of an algorithm on an instance as the time the algorithm takes to solve the instance on a single processor.

We consider a set of generalized Las Vegas Algorithms, whose runtime is a random variable whose value may be infinite [Hoos and Stützle, 2004]. We assume no prior information about the performance of the algorithms, except for their correctness, i.e. they will not falsely claim that an instance is solved: as sole additional hypothesis, we require each problem instance to be solvable by *at least one* of the algorithms.

Consider then a sequence $\mathcal{B}$ of M instances of a problem, $\mathcal{B} = \{b_1, \ldots, b_M\}$. To solve these, we are given a discrete, finite set $\mathcal{A}$, consisting of N deterministic or randomized algorithms, $\mathcal{A} = \{a_1, \ldots, a_N\}$; and a number $Z \geq 1$ of processors. The runtime of a_n on b_m is $t_n(m) \in [0, \infty]$. If a_n is randomized, $t_n(m)$ is a realization of a random variable $T_n(m) \in [0, \infty]$, obtained initializing a_n with a particular random seed. The distribution of $T_n(m)$, i.e. the RTD of a_n, is described by its survival function $S_n(t \mid m)$, representing the probability that a_n will not be able to solve b_m in a time t. The algorithms do not interact, so the $T_n(m)$ are independent. The $a_n \in \mathcal{A}$ can be replicated an arbitrary number of times, each copy solving the same instance b_m, but initialized with a different random seed; and executed on the Z processors according to an arbitrary schedule $\mathbf{s} \in \mathcal{S}$. Such parallel execution is termed a *portfolio* $\mathcal{A}_z(m; \mathbf{s})$: its runtime $t_{\mathcal{A},z}(m; \mathbf{s})$ is an instantiation of the random variable $T_{\mathcal{A},z}(m; \mathbf{s})$, whose distribution is $S_{\mathcal{A},z}(t \mid m; \mathbf{s})$

The only additional hypothesis on $\mathcal{A}$ and $\mathcal{B}$ is that each instance can be solved by at least one of the algorithms:

Hypothesis 1. $\forall m \in \{1, \ldots, M\}\ \exists n \in \{1, \ldots, N\}$ *such that* $t_n(m) < \infty$*, for any random seed.*

Note that n does not have to be the same for each instance: in other words, all this hypothesis requires is that $\mathcal{A}$ can solve $\mathcal{B}$ *at all*. An corollary of this hypothesis is that all instances in the sequence can be solved by a portfolio obtained executing all algorithms in $\mathcal{A}$ in parallel, with a share $\mathbf{s}$ such that $s_n > 0\ \forall n$.

We make no assumptions about the sequence $\mathcal{B}$: in particular, about any relationship among the difficulty of instances, and the order with which they arrive. We assume that the instances have to be solved in the given order, and that the solution of b_{m+1} cannot start before b_m has been solved: to be able to exploit parallelization, if $Z > 1$ we relax this latter requirement, allowing more instances to be solved in parallel, but still in the given order.

We do not consider methods of pre-selecting which solvers to include in the set $\mathcal{A}$, even though the allocation evaluated may exclude some of the algorithms, setting the corresponding s_n to 0.

A time allocator is a mapping from algorithm sets and problem instances to schedules. We indicate allocation of z processors as $\mathbf{s}_z := \mathrm{TA}(\mathcal{A}, m; z)$. We will drop the index z when considering a single processor $z = 1$, and drop the index m, or replace it with b, when considering a single instance b.

5.2 Static time allocation

Consider a portfolio $\mathcal{A}(\mathbf{s})$, consisting of N algorithms, $\mathcal{A} = \{a_1, a_2, ..., a_N\}$, solving the *same* problem instance, and running in parallel on a single processor ($Z = 1$), according to a static *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}.

In this chapter, we assume that the RTDs of each a_n on each b_m are available, i.e. for each (n, m) we know the survival function $S_n(t \mid m)$ of the runtime $T_n(m)$ of a_n. In this and the following section we consider a single problem instance $b_m = b$ so for simplicity we will drop the index m, or replace it with b.

As seen in Section 2.4.3, the runtime of the portfolio $\mathcal{A}$ solving a given instance according to a share $\mathbf{s}$ is the random variable $T_{\mathcal{A}}(\mathbf{s})$ (2.15): given the RTDs $\{S_n(t)\}$ of the a_n on the instance, its distribution $S_{\mathcal{A}}(t; \mathbf{s})$ can be expressed as a function of the share $\mathbf{s}$, as in (2.16), and the problem of allocating machine time can be formulated as an optimization problem, in which some quantity derived from the RTD is optimized with respect to the share $\mathbf{s}$. Here we propose three alternative allocators:

Expected time allocator: TA_E. The expected runtime value $E\{T_{\mathcal{A},\mathbf{s}}\} = \int_0^\infty t f_{\mathcal{A},\mathbf{s}}(t) dt$ can be minimized with respect to $\mathbf{s}$:

$$\mathbf{s}_E = \arg\min_{\mathbf{s}} E\{T_{\mathcal{A}}(\mathbf{s})\}. \tag{5.1}$$

Higher moments $E\{[T_{\mathcal{A}}(\mathbf{s})]^n\}$ can also be optimized.

Contract allocator: $\mathrm{TA}_C(t_u)$. If an upper bound, or *contract*, t_u on runtime is imposed, one can instead use (2.17) to pick the $\mathbf{s}$ that maximizes the probability of solution within the contract $F_{\mathcal{A}}(t_u; \mathbf{s}) = \Pr\{T_{\mathcal{A}}(\mathbf{s}) \leq t_u\}$, or, equivalently, minimizes $S_{\mathcal{A}}(t_u; \mathbf{s})$:

$$\mathbf{s}_C(t_u) = \arg\min_{\mathbf{s}} S_{\mathcal{A}}(t_u; \mathbf{s}). \tag{5.2}$$

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 $t_{\mathcal{A}}(\alpha; \mathbf{s}) = F_{\mathcal{A}}^{-1}(\alpha; \mathbf{s})$ should be minimized:

$$\mathbf{s}_Q(\alpha) = \arg\min_{\mathbf{s}} t_{\mathcal{A}}(\alpha; \mathbf{s}). \tag{5.3}$$

All three allocators above require optimizing a function of $\mathbf{s} \in \mathbb{R}^N$, with the additional constraint that the s_n should sum to one, so the actual search space dimension is $N - 1$. 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 $\mathbf{s}$. In Sect 6.1.1 we will display examples of these surfaces for $N = 2$.

A choice among the three alternatives, as well as the choice of the relative parameters, might be imposed by the particular application, or left open as a design decision. In Chapter 7, we will propose an automated way of performing this choice. In the next section, we extend these allocators to deal with multiple processors.

5.3 Allocation of multiple processors

People working with computationally demanding problems usually have a *cluster* of machines at their disposal, and address their algorithm selection problems by submitting a large number of jobs to the cluster front-end, which allocates jobs to machines in a purely FIFO[1] order. After a few hours, or days, the user has to inspect the results, and decide which algorithm to use on future problem instances.

We envision a different scenario, in which the user has to solve a sequence of problems, and is given a set of candidate algorithms, of unknown performance. An intelligent cluster front-end *learns* the performance of the various algorithms on different problems, and allocates cluster nodes to jobs in order to minimize solution time. Eventually, the user will still be able to inspect the results, but he will also have obtained a model-based allocator, that can be used, and further refined, on subsequent problems.

The TAs described in the previous section allocate a single processor. When more processors are available, they can be used to solve more instances in parallel, or the same instance with more parallel copies of the portfolio. As seen in Section 2.4, this may or not be a good idea, depending on the RTDs at hand. In this section we consider the allocation of multiple processors.

Consider again our N algorithms $\mathcal{A} = \{a_1, a_2, ..., a_N\}$. This time we need to allocate time on Z processors, so the share will be represented as a $N \times Z$ matrix $\mathbf{\Theta} = \{s_{ij}\}$, with columns summing to 1, $s_{ij} \in [0,1], \sum_{i=1}^{N} s_{ij} = 1 \forall j \in \{0, 1, \ldots, Z\}$.

For a given share $\mathbf{\Theta}$, the survival function for the distributed portfolio can be evaluated as (compare with (2.16) from Sec. 2.4.3):

$$S_{\mathcal{A},Z}(t;\mathbf{\Theta}) = \prod_{j=1}^{Z}\prod_{i=1}^{N} S_i(s_{ij}t), \tag{5.4}$$

The model-based time allocators introduced in Section 5.2 for a single processor can easily be defined for $Z > 1$ processors, but while in that case the share $\mathbf{s}$ was found optimizing functions in a $(N-1)$ dimensional space, here the size of the search space grows also with the number of processors, as $Z(N-1)$. This is obviously undesirable, as the computational cost of finding the optimum share is exponential in the dimensionality of the search space.

A simple approach to reduce the number of dimensions is to limit the search to discrete shares $\mathbf{z}$, where each processor is allocated entirely to a single algorithm. In this case, each

[1] First In First Out.

$s_{ij} \in \{0,1\}$, and (5.4) reduces to (2.19), where $z_i = \sum_{j=1}^{Z} s_{ij}$. The best allocation of this kind is not necessarily optimal, though.

Fortunately, and rather unexpectedly, we can prove that the optimal share for the contract and quantile allocators is the same on each processor. It can therefore be found with a search in a $(N-1)$ dimensional space, regardless of the number of processors available.

We first define a share $\mathbf{\Theta}$ to be *homogeneous* if all its columns are equal. We say that two shares $\mathbf{s}_1, \mathbf{s}_2$ are *equivalent* with respect to one of the allocators of Section 5.2 if they correspond to the same objective: for example, the same quantile $t_{\mathcal{A}}(\alpha; \mathbf{s}_1) = t_{\mathcal{A}}(\alpha; \mathbf{s}_2)$. Note that this does not imply $\mathbf{s}_1 = \mathbf{s}_2$, as the objective $t_{\mathcal{A}}(\alpha; \mathbf{s})$ may be multimodal.

For contract and quantile optimal shares, the following theorems hold:

Theorem 1. *For each contract-optimal share $\mathbf{\Theta}^*(t_u)$ there is an homogeneous equivalent $\mathbf{\Theta}_h^*(t_u)$, such that $S_{\mathcal{A}}(t_u; \mathbf{\Theta}^*) = S_{\mathcal{A}}(t_u; \mathbf{\Theta}_h^*)$.*

Theorem 2. *For each quantile-optimal share $\mathbf{\Theta}^*(\alpha)$ there is an homogeneous equivalent $\mathbf{\Theta}_h^*(\alpha)$, such that $F_{\mathcal{A}}^{-1}(\alpha; \mathbf{\Theta}^*) = F_{\mathcal{A}}^{-1}(\alpha; \mathbf{\Theta}_h^*)$.*

A difference among the two cases is that $\mathbf{\Theta}_h^*(t_u)$ does not depend on the number of processors. For the expected time allocator, a similar theorem cannot be stated: the following can be proved by a simple counter-example.

Theorem 3. *Expected-value-optimal shares do not necessarily have an homogeneous equivalent.*

All proofs are given in the Appendix, Section A.1.

The allocators presented above consider a fixed number of processors Z. Available processors may be used to solve different problem instances, or allocated to the same one. Also this decision should depend on the shape of the runtime distributions at play. Regardless of whether the shares are homogeneous or not, one simple heuristic for selecting also the number of processors is to optimize the allocation for $1, 2, \ldots$ up to Z processors, and then use the number z that gives the best result in terms of *total* computation time, i.e., zt if the instance is solved in a time t using z processors.

A more precise description of this method is reported in Alg. 4, where $\mathcal{A}_z(b; \mathbf{\Theta})$ is the portfolio solving b on z processors, according to a $N \times z$ share $\mathbf{\Theta}$, and $\mathrm{TA}_Q(b; z)$ is the allocator optimizing the share for instance b, on z processors. The description is based on the quantile allocator, so $\mathbf{\Theta}_z = \arg\min_{\mathbf{\Theta}} t_{\mathcal{A},z}(\alpha; \mathbf{\Theta})$, and $q_z = \min_{\mathbf{\Theta}} t_{\mathcal{A},z}(\alpha; \mathbf{\Theta})$ is the corresponding optimal quantile. The expected time allocator can be used simply minimizing $zE_{\mathcal{A},\mathbf{\Theta}}\{t\}$, instead of $zt_{\mathcal{A}}(\alpha; \mathbf{\Theta})$. A different heuristic should be devised for the contract allocator.

This simple approach can also be used to optimize the number of parallel runs of a single algorithm a with survival probability $S_a(t)$: in this case the RTD of the portfolio (5.4) simply becomes

$$S_{\mathcal{A},z}(t) = [S_a(t)]^z. \tag{5.5}$$

and can be used to optimize the number of processors $z \in \{1, \ldots, Z\}$, as described in Alg. 4.

In the next section, we will see how to use these or other static allocators to perform dynamic allocation.

5.4 Dynamic time allocation

A dynamic schedule is more general than a static one: Sayag et al. [2006] also proved that, with respect to total runtime, the best per set task switching allocation cannot be worse than the best

Algorithm 4 Static Parallel Algorithm Portfolio.

Instance set $\mathcal{B} = \{b_1, \ldots, b_M\}$;
Algorithm set $\mathcal{A} = \{a_1, \ldots, a_N\}$;
$\mathcal{A}_z(b, \mathbf{\Theta})$ solves b on z processors, with share $\mathbf{\Theta}$
Quantile $\mathrm{TA}_Q(b_m; z)$ allocates z processors with share $\mathbf{\Theta}_z$,
  minimizing $q_z = t_{\mathcal{A},z}(\alpha; \mathbf{\Theta})$ to solve b_m
Z processors
$Z_F := Z$ free processors
Allocated processors are freed as soon as b_m is solved, and Z_F is increased accordingly.
for $m = 1, \ldots, M$, **in parallel**, as soon as $Z_F > 0$ **do**
  for $z = 1, \ldots, Z_F$ **do**
    evaluate $\mathbf{\Theta}_z := \mathrm{TA}_Q(b_m; z)$
    $q_z := t_{\mathcal{A},z}(\alpha; \mathbf{\Theta}_z)$
  end for
  choose $z_m := \arg\min_z \{z q_z\}$
  allocate z_m processors to $\mathcal{A}_{z_m}(b_m; \mathbf{\Theta}_{z_m})$
  $Z_F := Z_F - z_m$
end for

per set resource sharing allocation (see Sec. 2.5). 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, 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, as the shares sum to 1. 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 5.

For simplicity, in Alg. 5 we considered a resource sharing $\mathbf{s}$, but the same approach can be applied to an arbitrary time allocator producing arbitrary schedules, 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 Alg. 5, if the RTD models can be conditioned on time-varying covariates $\mathbf{x}_n$; otherwise, each $S_n(t)$ can simply be updated based on time spent y_n, as in (3.22): writing $t' = t - y$, the RTD of the portfolio after a time $y = \sum_n y_n$

Algorithm 5 Dynamic Algorithm Portfolio

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

is

$$S(t') = \prod_{n=1}^{N} \frac{S(y_n) - S(t' + y_n)}{S(y_n)}. \tag{5.6}$$

A concrete example of the effect of this dynamic update will be reported in the next chapter (Section 6.1.3).

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 case, 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 $\{\tau_1, 2\tau_1, \ldots, 2^i \tau_1, \ldots\}$.

Regarding the allocation of multiple processors, a *dynamic* version of Alg. 4 can be implemented as in Alg. 5, periodically updating the allocation, and re-evaluating the optimal share. For homogeneous shares, the number of CPUs allocated should only be decreased, as increasing it would require to start the algorithms from scratch, with the unconditioned model: in this situation the optimal share would not be homogeneous, and we would have to search a larger space. The resulting dynamic portfolio on multiple processors is described in Alg. 6.

Another allocator which can be evaluated dynamically, as in Alg. 5, is the greedy task-switching schedule (2.23) of Streeter et al. [2007]. The expected value of $G(t; \mathbf{s})$ can be estimated for an arbitrary $\mathbf{s} = \{(s_1, \tau_1), \ldots\}$ based on the RTDs on the set of instances $\{F_n(t)\}$, as

$$G(t; \mathbf{s}) = M \sum_{n=1}^{N} F_n(\sum_{n_j == n} \tau_j). \tag{5.7}$$

Given (3.22), the rate maximized in (2.23) 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:

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

Algorithm 6 Dynamic Parallel Algorithm Portfolio.

```
Instance set B = {b_1, ..., b_M}
Algorithm set A = {a_1, ..., a_N}, A_z(b, Θ) solves b on z processors, with share Θ
Quantile TA(b; z) allocates z processors with share Θ_z,
  minimizing q_z = t_{A,z}(α; Θ) on instance b
Z processors
Z_F := Z free processors
Allocated processors are freed as soon as b_m is solved, and Z_F is increased accordingly.
for m = 1, ..., M, in parallel, as soon as Z_F > 0 do
  for z = 1, ..., Z_F do
    evaluate Θ_z(m) := TA(b_m; z)
    q_z(m) := t_{A,z}(α; Θ_z(m))
  end for
  choose z_m := arg min_z {z q_z(m)}
  allocate z_m processors to A_{z_m}(b_m; Θ_{z_m})
  Z_F := Z_F − z_m
  initialize Δt(m)
  while b_m not solved do
    run A_{z_m}(b_m, Θ_{z_m}) for a maximum time Δt(m)
    for z = 1, ..., z_m do
      update Θ_z(m) := TA(b_m; z)
      q_z(m) := t_{A,z}(α; Θ_z(m))
      choose z'_m := arg min_z {z q_z}
      update Δt(m)
      free processors: Z_F = Z_F + z_m − z'_m
    end for
  end while
  free processors Z_F = Z_F + z_m
end for
```

$\mathbf{s}_k = \{(n_1, \tau_1), \ldots, (n_k, \tau_k)\}$. The next portion of the schedule is

$$(n_{k+1}, \tau_{k+1}) = \arg\max_{(n,\tau)} \frac{F_n(\tau + y_n^{(k)}) - F_n(y_n^{(k)})}{[1 - F_n(y_n^{(k)})]\tau}. \tag{5.8}$$

Based on[Streeter et al., 2007], it should be possible to prove that using the instance RTDs in (5.8) would generate a per instance 4-optimal task switching schedule. An advantage of (5.8) is that is can be evaluated in a time $O(N)$, as it requires N line searches to find the optimal τ for each algorithm.

5.5 Related work

The expected runtime of a portfolio is optimized by Finkelstein et al. [2002, 2003], for task-switching schedules. Huberman et al. [1997] and Gomes and Selman [2001] propose instead to jointly minimize expected value and variance of runtime, for a resource sharing portfolio of two algorithms, manually selecting a share on the efficient frontier. It is not clear how to automate the selection for larger portfolios, when the efficient frontier is no longer a curve, but a multidimensional surface. The contract allocator (Sec. 5.2) is too general to be original: we assume it has been used already to perform some sort of time allocation. The quantile allocator (Sec. 5.2) is instead an original contribution, first appeared in [Gagliolo and Schmidhuber, 2006c]. The use of RTDs was adopted in [Gagliolo and Schmidhuber, 2005], but there the allocation was heuristic: predictions of the expected runtime were mapped to to $\mathbf{s}$ values according to a "ranking" approach, where the r-th expected fastest solver got a share $s_n = 2^{-r}$, as in the previous [Gagliolo et al., 2004].

Only limited research has been performed on the allocation of multiple processors. Finkelstein et al. [2002] consider a task-switching portfolio where each algorithm is executed on a separate processor, which is left idle when the corresponding algorithm is not running. Their aim is to optimize an utility function, taking into account both wall-clock time and the total computation. Luby and Ertel [1994] propose the parallel execution of their universal restart strategy.

A number of interesting dynamic exceptions to the static selection paradigm have been proposed recently. In the context of restart strategies, Horvitz et al. [2001] propose performance models conditioned on the behavior of the candidate algorithms during a predefined amount of time, called the *observational horizon*. Each algorithm is run on each training problem, with a high enough censoring threshold, and features are extracted from the dynamic data recorded during this initial period. Runs are then distinguished as belonging to two classes of "short" and "long" experiments, using the median of all recorded runtimes as a decision threshold. A mapping is learned from the static and dynamic features to the correct classification labels. This approach is used by Kautz et al. [2002] to implement dynamic context-sensitive restart strategies for SAT solvers: the authors assume that the runtime distribution of their algorithm is not known in advance, but belongs to a known finite set of distributions, from which the correct one can be discriminated based on dynamic features. The approach is dynamic in a limited sense, as the strategy is updated only once.

Algorithmic chaining [Borrett et al., 1996] executes a predetermined sequence of Constraint Programming solvers, using an ad-hoc mechanism to decide when to switch to next algorithm, according to a prediction of "thrashing" behavior, given the current state. This can be viewed as

a dynamic portfolio, with a task-switching schedule, but all its components are fixed, designed based on a-priori expertise.

Oblivious time allocators (see Sec. 2.6) are inherently dynamic, as their allocation depends on the dynamic state of the algorithms.

Some work on sequential composition of algorithms involve dynamic decisions. In *anytime algorithm monitoring* Hansen and Zilberstein [2001], the *dynamic performance profile* of a planning technique is updated according to its performance, in order to stop the planning phase when further improvements in the planned action sequence are not worth the time spent evaluating them. In this case both the quality of a solution and its computational cost are taken into account, based on *performance profiles*, which are assumed to be available.

In the context of search in program space, Solomonoff [2003] suggests that probabilities assigned to programs are updated also during the solution of a single instance.

In a Reinforcement Learning setting, algorithm selection can be formulated as a Markov Decision Process: in [Lagoudakis and Littman, 2000], the algorithm set includes sequences of recursive algorithms, formed dynamically at run-time solving a sequential decision problem, and a variation of Q-learning is used to find a dynamic algorithm selection policy; Petrik [2005b,a] considers a set of deterministic algorithms, evaluating dynamic resource sharing schedules based on dynamic programming. The allocation is allowed to changed a finite number of times, set in advance: the last allocation is kept indefinitely. Success Story algorithms [Schmidhuber et al., 1997] are reinforcement learners which can undo policy modifications that did not improve the reward rate, even during a single epoch. Also parameter tuning is inherently dynamic: for example, Battiti and Protasi [1997] propose a reinforcement learning feedback mechanism to adapt the size of the prohibition list of a tabu-search algorithm.

Restricting to time allocation methods where the algorithms do not interact, our simple dynamic portfolio (Alg. 5), also introduced in [Gagliolo and Schmidhuber, 2005], is the only example we are aware of where the dynamic re-allocation can be repeated an arbitrary number of times, which does not have to be pre-defined, and thus depends only on the runtimes on the current instance. Moreover, a dynamic share, where $\mathbf{s}(t) \in [0,1]^N$, is more general than a task-switching schedule, where $\mathbf{s}(t) \in \{0,1\}^N$. A similar scheme was used already in [Gagliolo et al., 2004], where the intervals Δt had a constant duration, and the time allocation was performed according to an oblivious heuristic. Not also that Alg. 5 is general, in that it can be used to turn any static allocator into a dynamic one, regardless of the schedule used, provided that the allocation can change dynamically in a nontrivial way.

5.6 Discussion

In this chapter we presented the core of our time allocation methods. We first formulated the time allocation problem in a more precise way, focusing on minimization of the computational complexity of problem solving, under some very mild hypothesis (Sec. 5.1): given a sequence of instances, and a set of algorithms of unknown performance whose parallel execution can solve all instances, allocate computation time to the algorithms in order to speed up the solution of the whole sequence. In Section 2.4.3 we had seen that, given the instance RTDs of each algorithm, the RTD of a static resource sharing portfolio can be evaluated for an arbitrary share. This allows to optimize the share based on the resulting RTD. In Section 5.2 we introduced three different optimality criteria, one of which parameterless, optimizing expected runtime, two other characterized by a continuous parameter, optimizing solution probability at a given *contract*

time, and runtime *quantile* respectively. In Section 5.3 we extended these methods to allocate multiple processors, proving that for two of them the optimal allocation is *homogeneous*, i. e., the same on all CPUs. Section 5.4 discussed a simple approach to use any static TA to perform *dynamic* allocation, updating it periodically. In RTD based allocators, this update can be obtained conditioning the estimated distributions on the runtime already spent by each algorithm. One advantage of this simple approach is that it can be used to evaluate a dynamic schedule before start, allowing for a more efficient implementation. Also the greedy allocator of Streeter et al. [2007] can be evaluated in this way, producing a 4-optimal task switching schedule.

In principle, the methods proposed here perform per instance time allocation: the resulting allocation may consist in executing a single algorithm, or a more general resource sharing portfolio, depending on the RTDs. In this sense, they are more general than single algorithm selection (Sec. 2.3), as they depend on the whole RTDs of the algorithms, and can therefore be applied also in situations where a parallel portfolio is more efficient (Sec. 2.4). They are also more general than per set allocation (Sec. 2.5), as per instance allocation can only improve over per set allocation. To achieve per instance allocation, the RTDs of the algorithms on the instance are needed. In practice, estimated regression models of these RTDs (Sec. 3.4) may be used instead, conditioned on instance features. The resulting allocation will obviously be not optimal anymore, but heuristic: in the next chapter we will discuss this aspect in more detail, and see how censoring can reduce the computational cost of learning RTD models. In Part IV, we will show experiments where such an heuristic allocation already allows to significantly reduce the computational cost of problem solving.

Chapter 6

Modeling runtime distributions

In the previous chapter we described methods for allocating time based on the RTDs of each algorithm on each problem instance. In related work (Sec. 2.4), these functions are typically assumed to be known beforehand. This is usually not the case in practice: the allocators can still be implemented based on approximate *models* of such distributions.

Modeling the RTD separately for each instance is clearly not worth the effort, as each model would only allow to allocate time on an instance which has already been solved: what would be useful instead is a model which can *generalize*, and predict, with some approximation, what will be the RTD of an algorithm on a *new* instance, given its runtime on previously solved instances. In single algorithm selection, such generalization is obtained with regression models of the expected runtime, conditioned on instance features. In Section 3.4, we have seen that there are survival analysis techniques which allow to estimate regression models of the RTD. These models can be used to implement a per instance algorithm portfolio.

In this chapter we discuss two issues related with such estimation. The first one is that, as one would expect, using an estimated RTD violates the hypotheses on which the allocators are based, turning them into heuristics, albeit principled. Another issue is that modeling runtime distributions has an inherent computational cost, not necessarily due to the modeling itself, but rather to the preliminary phase of collecting a runtime sample, which in practice requires solving several problem instances with each algorithm.

As seen in Chapter 2, all non-oblivious allocators, model based or not, require such a runtime sample. To collect this data, an offline approach is followed in most cases (see, e.g., Sec. 2.3, 2.5): several "training" problem instances are solved repeatedly with each algorithm, possibly repeating the process if the algorithms are randomized. The computational complexity of this preliminary "runtime sampling" is roughly linear in the number of algorithms, and in the number of training instances, but it can nonetheless be relevant in practice, if not prohibitive[1]. While this phase can be easily parallelized on multiple CPUs, the same computation power could be used otherwise: there may be particularly inefficient (*algorithm*, *instance*) combinations which are not worth too much computation time, or others which are so efficient that it is easy to learn about them.

In our opinion, even offline methods could benefit from a sounder way of allocating machine time during the initial training phase, deciding which experiments to perform, how many in-

[1] For example, in the random category of SAT 2007 (Sec. 1.3), the total runtime of the winner amounts to 246 CPU hours, while the runtime for all 14 algorithms is 4.577 hours, almost 20 times larger.

stances to solve, how many times should randomized algorithms be tested on the same instance, how high should be the censoring threshold, and so on. Some of these questions will be dealt with in the next chapter: here, we will in particular discuss the choice of the censoring threshold.

This chapter is organized as follows: in Section 6.1, we discuss the bias induced in our methods when estimated RTDs are used, in place of the actual RTDs. Section 6.2 proposes the use of censoring as a tool to reduce the computational cost of sampling algorithm runtimes, arguing that a portfolio can be used to sample runtimes. Section 6.3 discusses the bias induced by this form of censoring. Section 6.4 discusses related work, and Section 6.5 concludes the chapter.

6.1 Allocation based on estimated RTDs

The time allocators described in the previous sections rely on the knowledge of the RTD of each a_n *on the particular problem instance* being solved. In practice, such distributions are not available beforehand, and have to be estimated, sampling the runtime of each algorithm by solving the instance repeatedly, with different random seeds. Obviously, from the point of view of time allocation, it would make no sense to invest all this computation time in solving an instance repeatedly, only to be faster at solving the *same* instance again. What is generally assumed in per instance selection (Sec. 2.3), is that some form of *knowledge transfer* across different instances is possible, and the algorithms will present a similar behavior on similar instances: sampling the performance on a problem instance may then help speed up the solution of *other* similar instances.

This intuitive notion of similarity is usually formalized via a set of *features*, $\mathbf{x} \in \mathbb{R}^d$, which characterize each problem instance. As seen in Section 3.4, such information can be taken into account also when modeling RTDs, using regression models conditioned on $\mathbf{x}$, $\{\hat{S}_n(t \mid \mathbf{x})\}$. Such models are learned based on runtime samples $\mathcal{R} = \{(t_m, c_m, \mathbf{x}_m)\}$, one for each algorithm, obtained collecting its (possibly censored) runtime t_m on several training problem instances b_m, each characterized by its feature value $\mathbf{x}_m$: c_m is the event indicator, 1 if the instance was solved at t_m, 0 otherwise. The estimates $\{\hat{S}_n(t \mid \mathbf{x})\}$ of the RTDs on a novel instance b, with features $\mathbf{x}$, are obtained based on the available sample: such evaluation depends on the modeling method used. In parametric regression models, the parameters are themselves a parametric function of $\mathbf{x}$, whose weights have been optimized based on the likelihood of the sample. In nonparametric models, the different observations $(t_m, c_m, \mathbf{x}_m)$ in the sample are used to estimate the RTD for the novel instance, based on the similarity among $\mathbf{x}$ and each $\mathbf{x}_m$. In any case, the estimate of the RTD of each algorithm on instance b is based on the runtimes observed on a set of *different* instances. Let us look in more detail at the possible implications of such approach.

Recall that the RTD of the portfolio is evaluated based on the assumption of independence of the runtime values $\{T_n\}$ of the different algorithms, which allows the joint probability (2.16) to be expressed as a product (Sec. 2.4). This assumption is satisfied only if each $S_n(t)$ represents the actual RTD of a_n on b. Formally, using regression models $S_n(t \mid \mathbf{x})$, the *conditional* independence of the runtime values would be sufficient to satisfy the assumption, evaluating $S_{\mathcal{A}}(t; \mathbf{s})$ as:

$$\hat{S}_{\mathcal{A}}(t \mid \mathbf{x}; \mathbf{s}) = \prod_{n=1}^{N} \hat{S}_n(s_n t \mid \mathbf{x}). \tag{6.1}$$

An analogous approach can be taken to choose a uniform restart strategy, minimizing (2.12) with an estimated $\hat{F}(t \mid \mathbf{x})$ in place of the real $F(t)$.

In both cases, the use of a model renders the resulting method potentially suboptimal for each instance. In practice, its performance will depend on how close the estimated RTD is to the real one. This will in turn depend on many factors, including the modeling technique used, the size of the available runtime samples, the features used to describe the instances, and so on. Regarding the features $\mathbf{x}$, the precision of the estimate will depend on how *informative* they are about the RTDs of each algorithm, i.e. on how similar the RTDs are for instances with similar features. For restarts, the optimal uniform strategy, obtained minimizing (2.12) based on a model $\hat{F}(t \mid \mathbf{x})$, may in general be suboptimal for each instance with features $\mathbf{x}$. For portfolios, if the algorithms display different RTDs on instances with similar covariates $\mathbf{x}$, there is the possibility that these differences will be *correlated*, violating the assumption (6.1). In the following subsections we analyze a toy example where the effect of this correlation is particularly evident, proposing ways of correcting it; the impatient reader may skip to Section 6.2.

6.1.1 An example: SAT/UNSAT

The potential effect of a correlation among runtimes of the different algorithm composing a portfolio can be better explained with an example.

Consider the following time allocation scenario (see also Sec. 9.5): two solvers for the satisfiability problem (Sec. 2.1.1), one complete (a_1, `Satz-Rand`, Gomes et al. [2000]), one incomplete (a_2, `G2WSAT`, Li and Anbulagan [1997]), and two instances from SATLIB [Hoos and Stützle, 2000], one SAT (`uf250-01`) and one UNSAT (`uuf250-01`). In the following we will refer to this as the SAT/UNSAT scenario. We are asked to allocate time to the two algorithms in order to minimize the time to solve one of the two instances, chosen at random, whose satisfiability is unknown. As we will see from the RTDs of the algorithms, `G2WSAT` can be much faster if the instance is SAT; while only `Satz-Rand` can prove unsatisfiability of the UNSAT instance. The correct allocation therefore should be: run only `G2WSAT` if the instance is SAT, otherwise run only `Satz-Rand`.

The two instances cannot be differentiated based on features: this means that any regression model $\hat{S}_n(t \mid \mathbf{x})$ would predict the same RTD for a_n, regardless of the satisfiability of the instance. Moreover, if the model was learned on a set of other SAT and UNSAT instances, it would estimate the RTD of that set. This is an extreme example of the correlation we were referring to. If the instance is satisfiable, both a_n will be faster than predicted by the corresponding $\hat{S}_n(t)$. If the instance is unsatisfiable, a_1 (`Satz-Rand`) will be slower than predicted, and a_2 (`G2WSAT`) will never solve it. It is clear that in these conditions $\hat{S}_1(s_n t)$ and $\hat{S}_2(s_n t)$ do not represent the probabilities of independent events, so the joint probability $S_{\mathcal{A}}(t \mid \mathbf{x}; \mathbf{s})$ of the two events cannot be expressed as their product, and (6.1) does not hold.

Based on this example, we can better analyze the issue, and also propose three simple approaches which can in principle solve it. The difference among the RTD on an instance and that on a set of instances is that the latter also models the random aspect of choosing an instance from the set (Sec. 3.1). To abstract from other sources of error, suppose we are given the exact RTDs on the set. It is interesting to see what happens if we use these RTDs to allocate time with one of the methods described in the previous chapter, as it allows to isolate the effect of correlation.

In this case, the set is composed of 2 instances, one UNSAT and one SAT: in the following we refer to these two instances as 0 and 1, respectively. Be $S_n(t \mid 0)$ and $S_n(t \mid 1)$ the corresponding survival functions of a_n. Even if we do not know these functions, we can use them to express the

$S_n(t)$ of a_n on the set according to (2.7), as

$$S_n(t) = 0.5S_n(t \mid 0) + 0.5S_n(t \mid 1). \tag{6.2}$$

If we substitute (6.2) in (2.16), we see the result of evaluating the RTD of the portfolio based on the RTDs on the set:

$$\begin{aligned} S_{\mathcal{A}}(t;\mathbf{s}) &= \prod_{n=1}^{2} S_n(s_n t) = \prod_{n=1}^{2} [0.5S_n(s_n t \mid 0) + 0.5S_n(s_n t \mid 1)] = \\ &= 0.25[S_1(s_1 t \mid 0)S_2(s_2 t \mid 0) + S_1(s_1 t \mid 0)S_2(s_2 t \mid 1) + \\ &+ S_1(s_n t \mid 1)S_2(s_2 t \mid 0) + S_1(s_1 t \mid 1)S_2(s_2 t \mid 1)]. \end{aligned} \tag{6.3}$$

Let us now look at the RTD of the process which we are actually trying to speed up: pick one of the two instances, with equal probability, and solve it with the portfolio. With probability 0.5, instance 0 will be chosen, and the two a_n will have RTDs $S_n(t \mid 0)$; otherwise their RTDs will be $S_n(t \mid 1)$. The RTD of the portfolio will then be a mixture of the RTDs (2.16) in the two cases:

$$\begin{aligned} S_{\mathcal{A}}(t;\mathbf{s}) &= 0.5\prod_{n=1}^{2} S_n(s_n t \mid 0) + 0.5\prod_{n=1}^{2} S_n(s_n t \mid 1) = \\ &= 0.5[S_1(s_1 t \mid 0)S_2(s_2 t \mid 0) + S_1(s_1 t \mid 1)S_2(s_2 t \mid 1)]. \end{aligned} \tag{6.4}$$

Compared to the correct (6.4), equation (6.3) contains two additional terms, where the survival functions *on different instances* are multiplied. Indeed, (6.3) models the RTD of a different process: pick one of the two instances, with equal probability, *independently for each algorithm*; run the resulting portfolio, and stop as soon as one of the instances is solved.

To illustrate the situation, Figure 6.1(a) displays the RTD of the two algorithms on the two instances, while Figure 6.1(b) displays their RTDs on a randomly chosen instance. The same figure also reports the correct (6.4) and wrong (6.3) evaluations of the RTD of the uniform portfolio ($s_n = 0.5$). The two curves confirm the interpretation given above. The correct RTD describes a portfolio where the two algorithms are solving the same instance. The curve is composed of two portions, corresponding to the two additive terms in (6.4). The portion until the median corresponds to the SAT instance: in this case `G2WSAT` is always faster, and the curve corresponds it its RTD, scaled and shifted in time, as it is only run for half of the time. The portion above the median corresponds instead to the UNSAT instance, and to the RTD of `Satz-Rand`. The wrong RTD can be described dividing it in four segments, corresponding to the four terms of (6.3), and to four possible combinations of instances and algorithms. In two of them, `G2WSAT` is solving the SAT instance, and will again be the fastest, so the two curves coincide until the median. The part above the median corresponds to the two combinations in which `G2WSAT` is trying to solve the UNSAT instance: we can recognize the shape of the RTD of `Satz-Rand` on the SAT instance (dashed black line in Figure 6.1(b)), between the median and quantile 0.75, and on the UNSAT instance for the remaining quartile.

In this example, there is an *unobserved feature* of the instance, its satisfiability, which is not taken into account by the RTD models. Using such models in (6.1) violates the independence assumption; evaluating the share accordingly optimizes (6.3) instead of (6.4), so it may produce a suboptimal share. As said, this is an extreme example, as the unobserved feature has in this case an important impact on the runtime: it can make a difference of orders of magnitude for a_1, and make the problem unsolvable for a_2. In general, we will have a similar issue when using

the RTD of a set of instances *as if* it was the RTD of a single instance, which is unavoidable when using regression models of the RTDs, as they do not allow to discriminate among different instances with the same features $\mathbf{x}$. In practice, the impact will depend on how different the $S_n(t \mid m)$ are for different instances b_m: looking again at (6.3), we can see that its difference from (6.4) depends on the difference among $S_n(t \mid 0)$ and $S_n(t \mid 1)$. If $S_n(t \mid 0) \approx S_n(t \mid 1)$, the two equations would also be similar.

Estimating the RTD of the portfolio based on the RTDs of the algorithms on the set gives a result which is visibly wrong. Ultimately, what we are interested in is not a faithful model of this RTD, but an efficient allocation of processor time. Let us then see how this mistake impacts the time allocators from the previous chapter (Sec. 5.2). Figure 6.2(a) displays the expected runtime of the portfolio as a function of the share to `Satz-Rand`, s_1, as evaluated from (6.3), compared with the correct expected time of the portfolio, from (6.4): the wrong optimum $s_1 = 0.95$ is actually very close to the correct one, at $s_1 = 0.97$. The same figure plots the correct expected runtime evaluated separately for the two instances, based on the instance RTDs, with minima at $s_1 = 0$ for the SAT instance, and $s_1 = 1$ for the UNSAT, as expected. Note that the wrong allocation would still be used to control the correct process, whose RTD is (6.4). In order to compare it with the correct allocation, in Figure 6.2(b) we display the correct RTD for both allocations, on a randomly picked instance, as well as on each instance: the wrong allocation has an advantage if the instance is satisfiable, while the correct one will be faster if the instance is UNSAT (hardly visible given the log scale). The continuous lines refer to the solution of a randomly picked instance: also in this case the curves can be divided in two parts, below and above the median, corresponding to the SAT and UNSAT instance respectively. The difference among the two allocation is more visible for the satisfiable instance, and it can be better understood looking at the RTDs of the two algorithms on the SAT instance, in Fig. 6.1(a). `G2WSAT` is run for a fraction of time $s_2 = 0.05$ in the wrong portfolio, and $s_2 = 0.03$ in the correct one. The resulting RTDs can be obtained multiplying the time values of the original one with factors 20 and 33, respectively, so they will both intersect the RTD of `Satz-Rand`, which is only slightly affected as this algorithm is run for most of the time. This intersection happens earlier for the wrong allocation, as giving more time to `G2WSAT` also increases the probability that it will be faster than `Satz-Rand`, even if run for a much smaller fraction of machine time.

Figures 6.3–6.5 report similar plots for the quantile allocator $\mathrm{TA}_Q(\alpha)$ (5.3), evaluated for different values of α. For the lower quartile ($\alpha = 0.25$, Fig. 6.3) the allocation is exactly the same, $s_1 = 0$, so the RTDs coincide. The same situation is observed for the median ($\alpha = 0.5$): in this case the curves optimized differ (see Fig. 6.4(a)), but have the same minimum, again at $s_1 = 0$. In both cases, the choice is to run only `G2WSAT`, as expected: the two allocators minimize the time to solve a randomly selected instance with probabilities 0.25 and 0.5 respectively. As the instance is satisfiable with probability 0.5, in both cases `G2WSAT` is the correct choice.

The opposite situation can be observed in Figure 6.5, where a quantile $\alpha = 0.75$ is optimized. In this case, both the correct and the wrong allocators rely on `Satz-Rand`. The functions optimized and their optima differ only slightly: $s_1 = 0.99$ minimizes the quantile of the wrong (6.3), $s_1 = 1$ the correct (6.4). Also in this case, the wrong allocator will have a small advantage if the instance is SAT, and a small disadvantage if the instance is UNSAT.

In the next section we present a small modification of the RTD evaluation which allows to correct the estimate in this and similar cases.

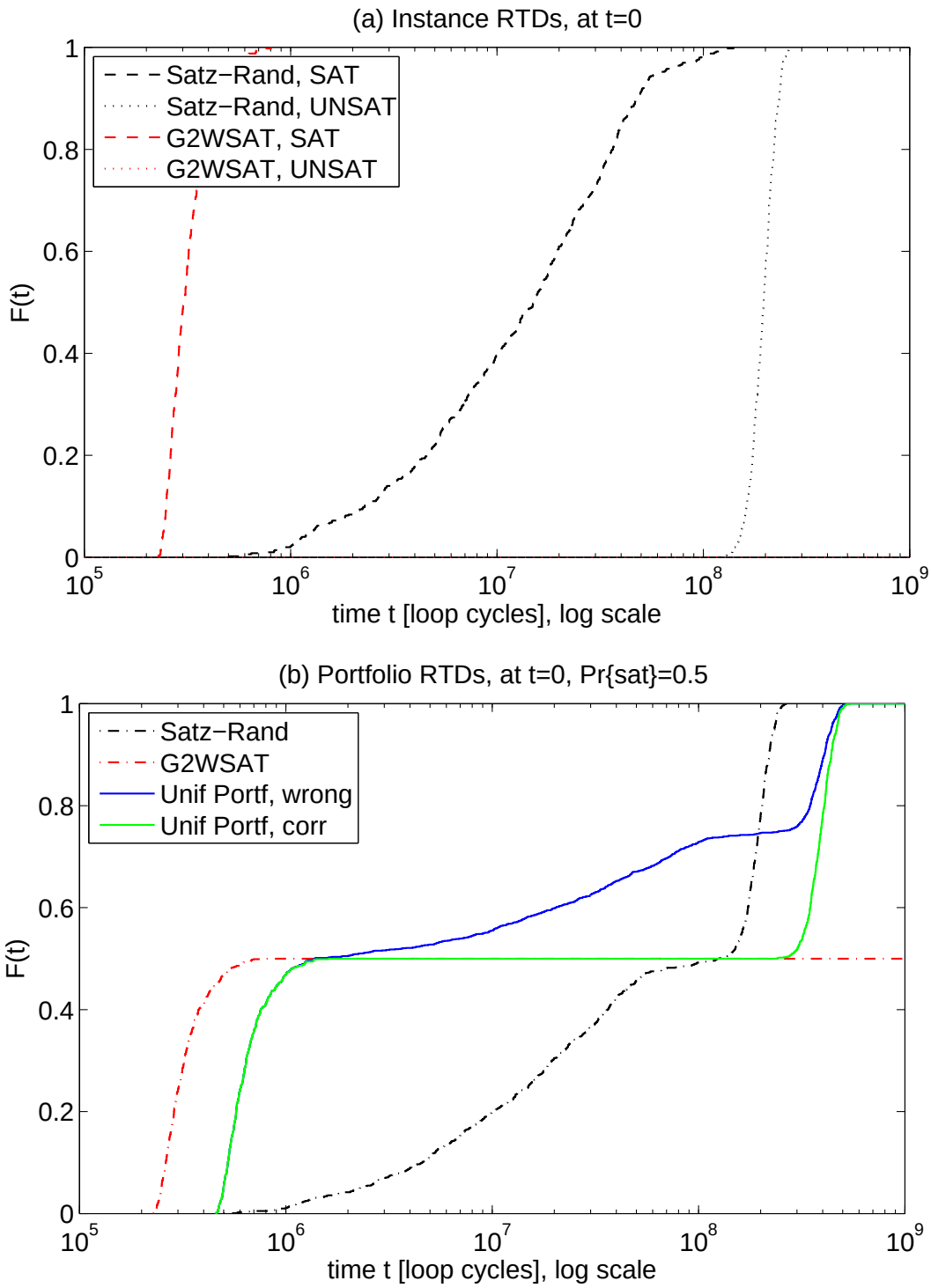


Figure 6.1. (a) RTD of the two algorithms on the two instances. The line for G2WSAT UNSAT is constant at 0, as this algorithm cannot prove unsatisfiability. (b) RTD of the algorithms and uniform portfolio on a randomly picked instance. Comparison of correct (6.4) and wrong (6.3) evaluations. See text for details.

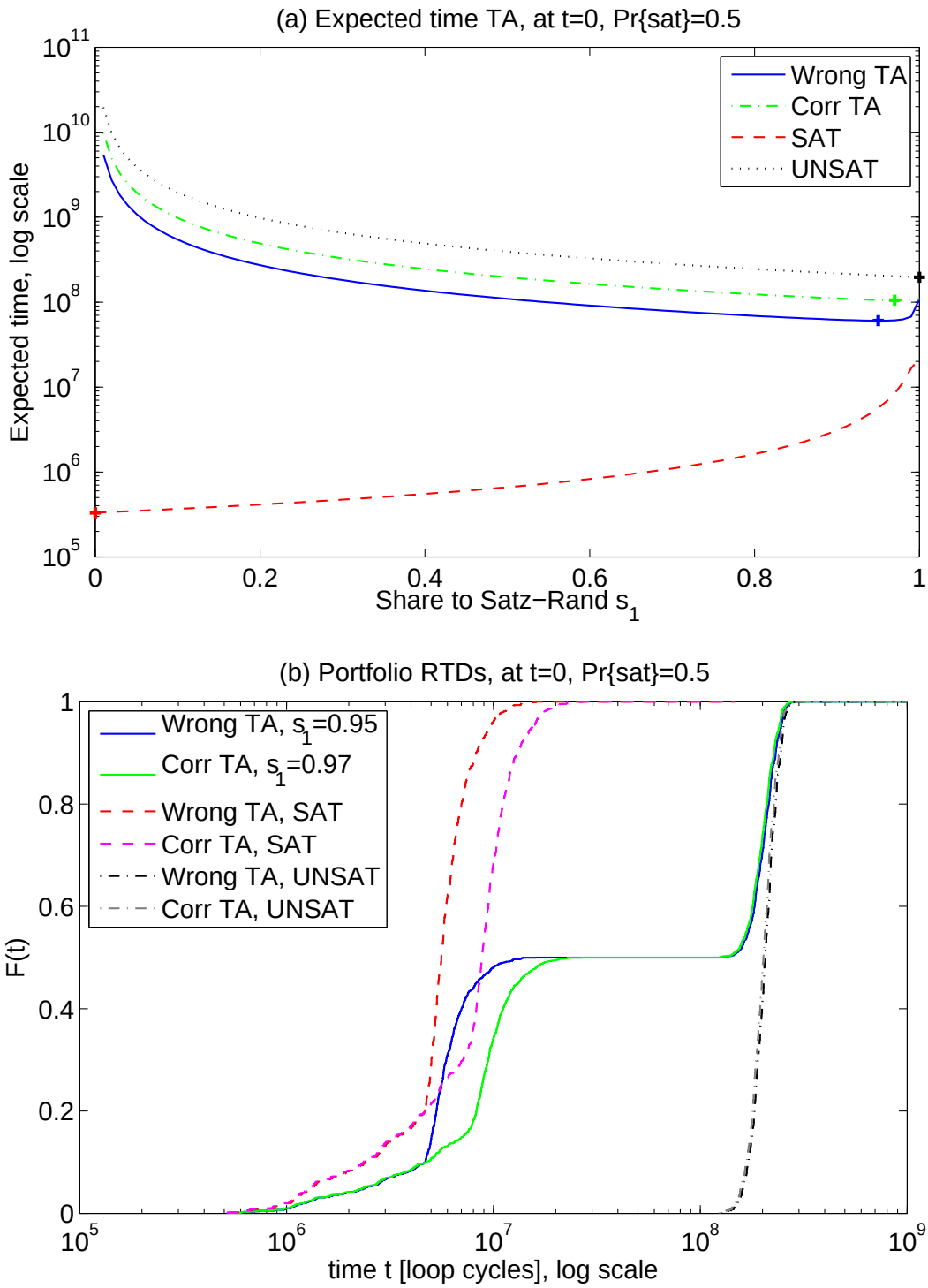


Figure 6.2. Expected Time: Comparison among wrong and correct allocation. (a) Functions optimized. Plus signs indicate the optima of the different objectives. (b) Optimal allocations. See text for details.

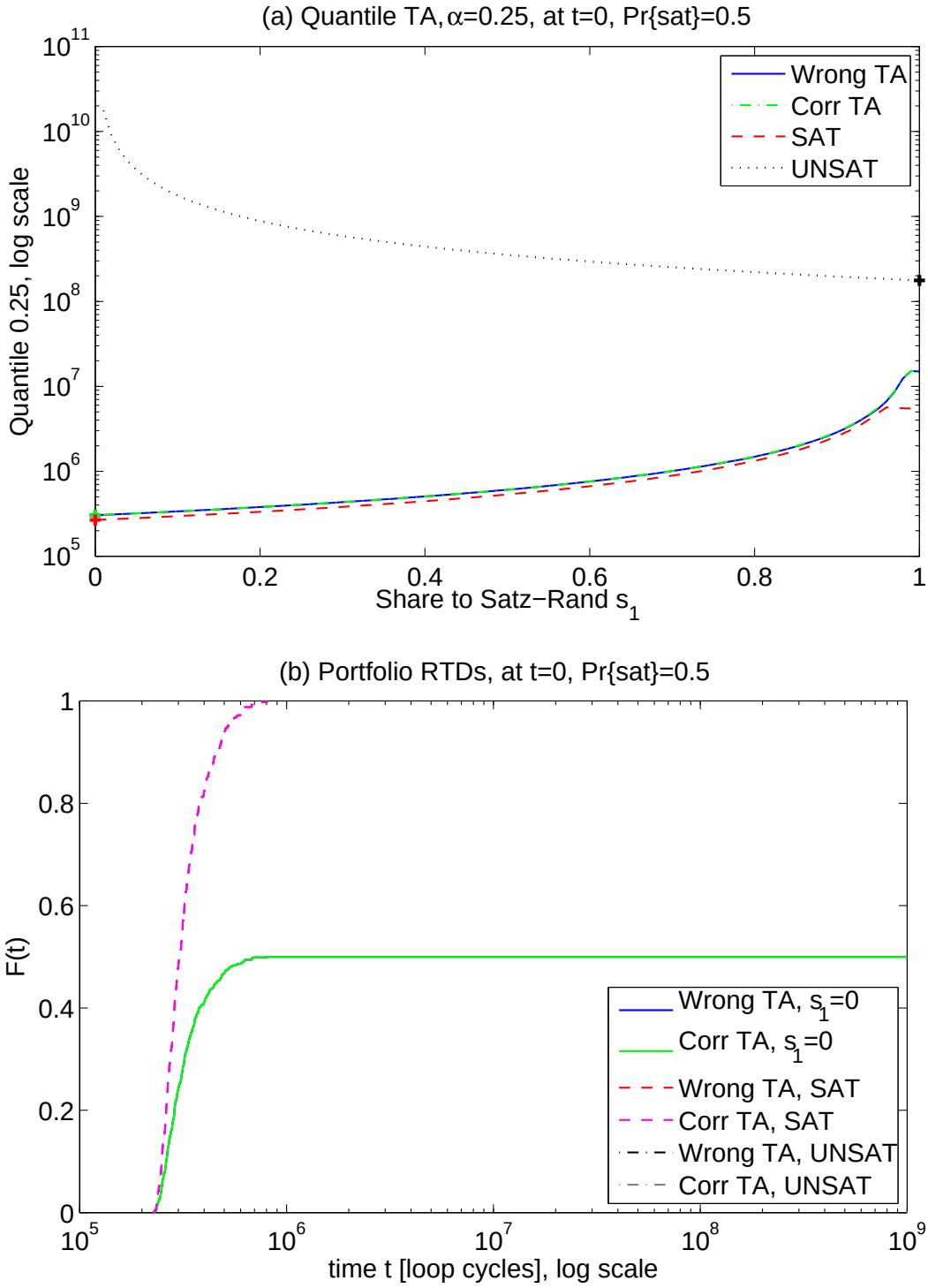


Figure 6.3. QUANTILE ($\alpha = 0.25$): Comparison among wrong and correct allocation. (a) Functions optimized. Plus signs indicate the optima of the different objectives. (b) Optimal allocations. In this case the wrong and correct TA evaluations produce the same share, so the corresponding CDF are superimposed: for the UNSAT instance, they remain at 0, as $s_1 = 0$. See text for more details.

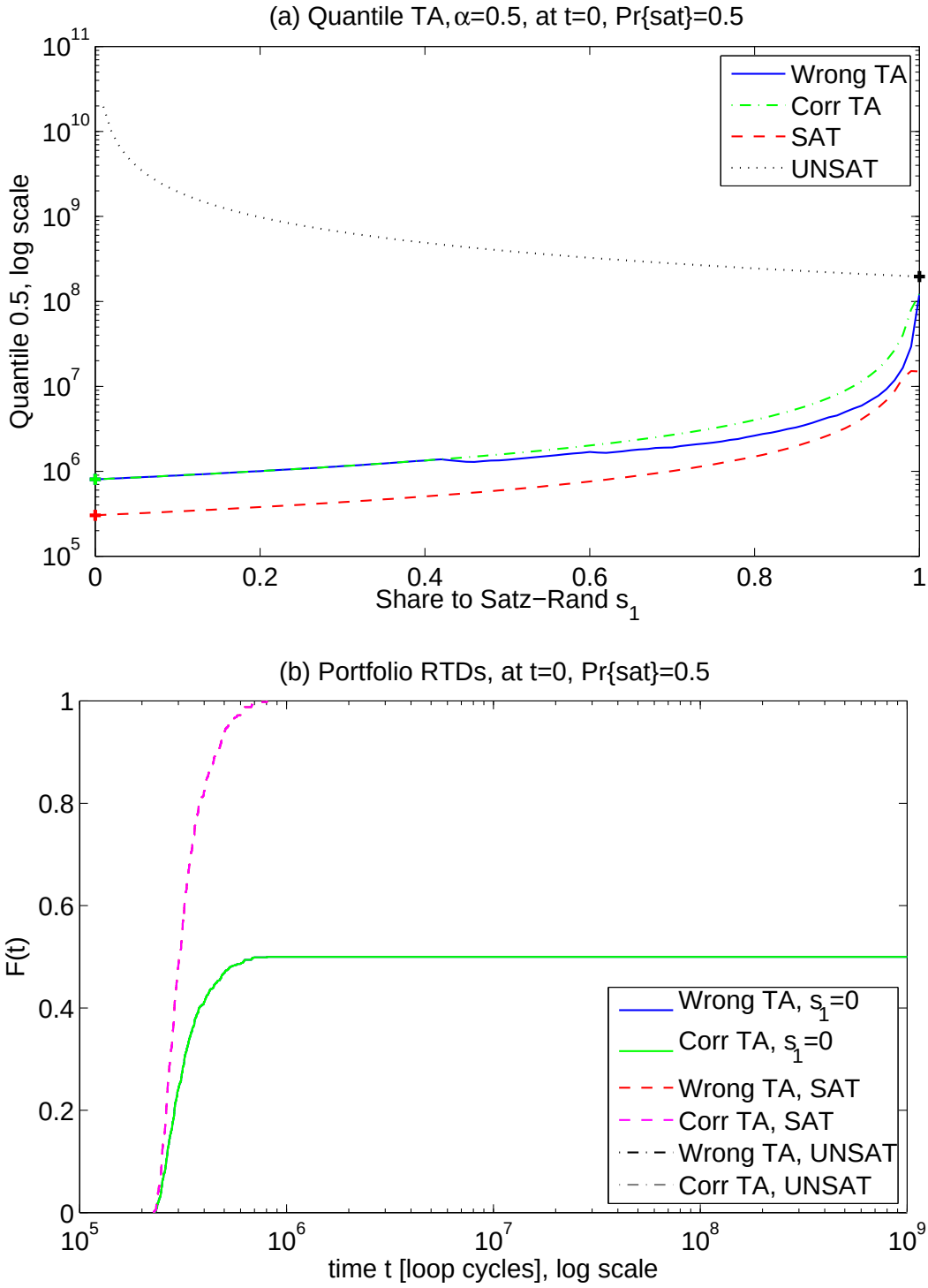


Figure 6.4. QUANTILE ($\alpha = 0.5$): Comparison among wrong and correct allocation. (a) Functions optimized. Plus signs indicate the optima of the different objectives. (b) Optimal allocations. Also in this case the wrong and correct TA evaluations produce the same share, so the corresponding CDF are superimposed: for the UNSAT instance, they remain at 0, as $s_1 = 0$. See text for more details.

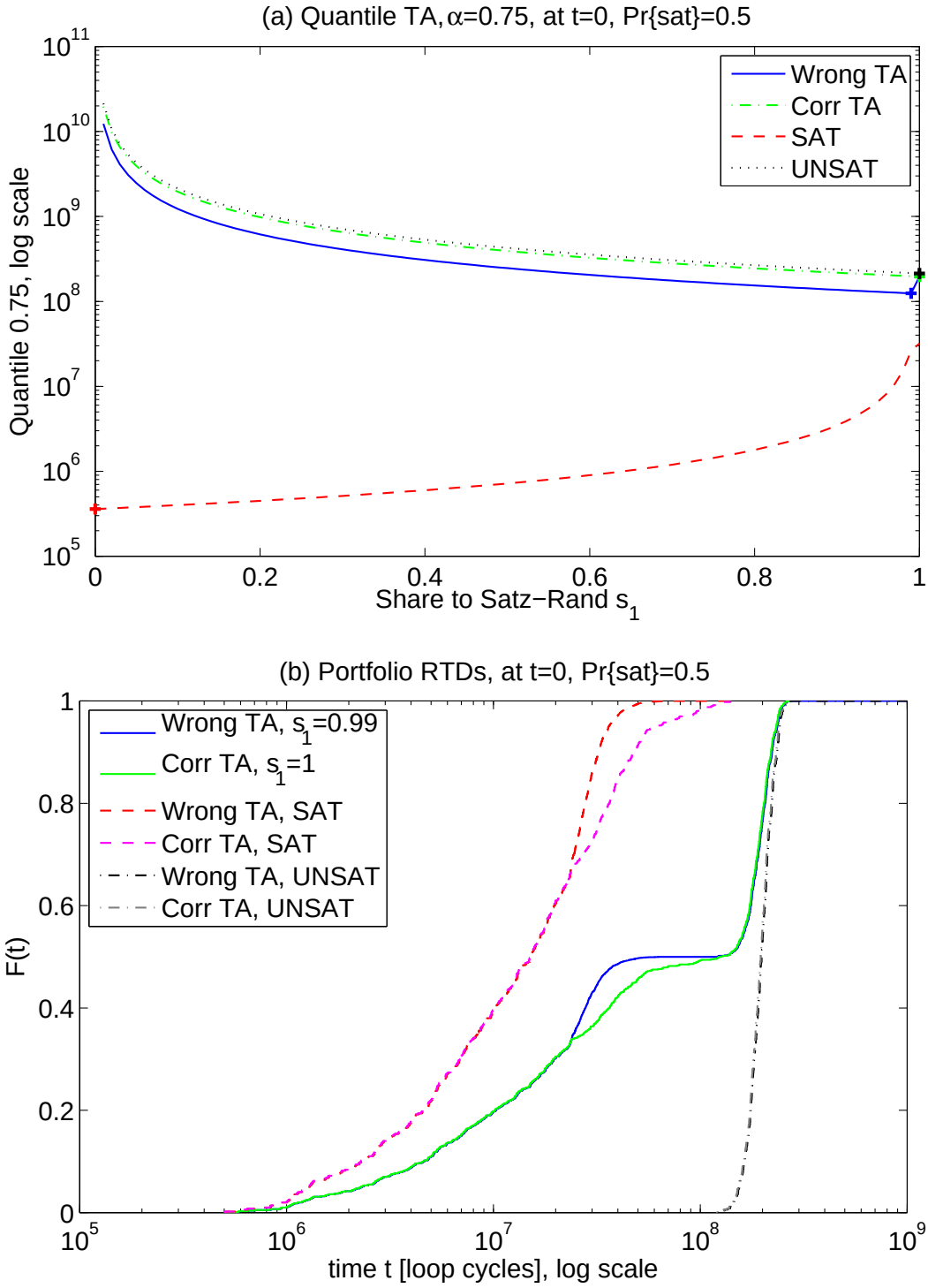


Figure 6.5. QUANTILE ($\alpha = 0.75$): Comparison among wrong and correct allocation. (a) Functions optimized. Plus signs indicate the optima of the different objectives. (b) Optimal allocations. In this case the wrong and correct TA evaluations produce almost the same share, so the corresponding CDF are almost superimposed, especially for the UNSAT instance, due to the logarithmic scale. See text for more details.

6.1.2 Correcting the correlation

This "toy" time allocation problem is quite revealing, and it implicitly suggests three ways of addressing this issue. The first one is a simple trick which solves the problem in a radical way. If we cannot evaluate the correct (6.4), let us make (6.3) correct: we can actually pick the instance independently for each algorithm, and solve different instances in parallel. This simple approach would guarantee that the independence assumption which allows to write (6.1) is verified *regardless of the features used*, as the runtime of the two algorithms solving different instances can always be considered independent. Unfortunately, the potential benefit of the parallel solution of the same instance would be not exploited. Imagine to apply this idea in the SAT/UNSAT example: there is a 0.25 probability that `Satz-Rand` is given the SAT instance, while `G2WSAT` tries to solve the UNSAT instance.

The second idea is less general, as it can only be applied to unobserved features which become available once the instance is solved. This is the case of satisfiability. In such case, if we know the probability that the instance we are solving is satisfiable, we can optimize the correct (6.4). In general, consider a discrete feature $x \in \mathcal{D}$, with a finite number of possible values, which is unobservable during the solution of an instance, but is revealed with solution. We can use x to learn a conditional RTD model $S_n(t \mid x)$ for each algorithm, and estimate the discrete probability $p(x)$ of the feature. For each unseen instance, we can set the share optimizing

$$S_{\mathcal{A}}(t;\mathbf{s}) = \sum_{x\in\mathcal{D}} p(x) \prod_{n=1}^{N} S_n(s_n t \mid x) \tag{6.5}$$

In the SAT/UNSAT example, $x \in \{0,1\}$ is the satisfiability of the instance, with $p(0) = p(1) = 0.5$, and the above formula corresponds to the correct RTD (6.4).

Another general idea could be to use the RTD of the set (2.7) explicitly in evaluating the portfolio RTD (2.16). More precisely, given the RTDs of each a_n on each of $M_{\mathbf{x}}$ training instances $\{b_m\}$, all with feature $\mathbf{x}$, one can estimate the portfolio RTD on a new instance with the same features as

$$S_{\mathcal{A}}(t \mid \mathbf{x};\mathbf{s}) = \frac{1}{M_n} \sum_{m=1}^{M_{\mathbf{x}}} \prod_{n=1}^{N} S_n(s_n t \mid m). \tag{6.6}$$

In Part IV, we will see that these ideas do not sensibly improve the performance. Indeed, an even more important lesson to be learned from this example is that even a visibly wrong model can allow to evaluate a near-optimal allocation. Looking at the plots in Figs. 6.2–6.5(b) we can see that the RTD of the portfolio estimated according to the RTDs of the algorithms on the set, as in (6.3), differs greatly from the correct RTD (6.4): however, the resulting allocations are nearly identical. As discussed above, in this case we would have the possibility of evaluating the correct RTD, as in (6.5). In general, we will not have this possibility: using regression RTD models $\hat{S}_n(t \mid \mathbf{x})$ implies accepting the possibility that the same $\mathbf{x}$ may characterizes different instances, on which the RTDs of the same algorithm may differ, and that these differences may be correlated for different algorithms, as in this example. In this case we know the reason of this correlation, and we can exploit this knowledge. In general, we will have no way of distinguishing two instances with the same $\mathbf{x}$, not even after we solved both of them. While solving them, however, we can improve the allocation, simply refining the RTD estimates as time passes, implementing a dynamic portfolio (Sec. 5.4), as exemplified in the next section.

6.1.3 The effect of dynamic updates

Another lesson to be learn from this example is that, as expected, static allocation is risky. The expected time and 0.75 quantile allocators both give the lion share to `Satz-Rand`, so their performance will be poor if the instance turns out to be SAT. On the contrary, the median and 0.25 quantile allocators only run `G2WSAT`, which means that their runtime will be infinite on the UNSAT instance. In Section 5.4, we have seen that a dynamic allocator can be easily obtained, updating a static allocation periodically. Even without taking into account time-varying covariates, an RTD model can be updated simply conditioning it on the time already spent, shifting and scaling them as in (3.22) (Sec. 3.5). To illustrate the effect of a dynamic update in the SAT/UNSAT example, suppose that we start solving the randomly chosen instance with the uniform portfolio, $s_1 = s_2 = 0.5$, and that none of the algorithms has solved it after a wall-clock time $t = 10^6$. At this point, each a_n will have spent a share $y_n = 5 \times 10^5$. Graphically, the effect of conditioning the RTDs on this information, as in (3.22), can be seen in Figure 6.6: the portion of the original RTDs (Figure 6.1) above y_n is shifted back of y_n in time, shifted down of $F(y_n)$ in probability, and scaled with a factor $1/S(y_n)$. Compare the RTDs on on each instance (Fig. 6.6(a)) and on the set (Fig.6.6(b)). To evaluate the original ones (Fig. 6.1(b)), at $t = 0$, we could use (2.7), knowing that the two instances had the same probability of being chosen. If we repeated the same evaluation now, based on the updated RTDs on the instances (Fig. 6.6(a)), and again using $p(0) = p(1) = 0.5$ we would not obtain the correct RTDs on the set. Indeed, that value of $p(x)$ was correct before starting to run the portfolio: now that both algorithms have been running for some time, without success, we do have some additional information, and start suspecting that the instance may be UNSAT. In general, the posterior probability of an unobserved feature x, whose prior probability before starting the portfolio is $p_0(x)$, can be updated based on the information that each a_n has already spent a time y_n unsuccessfully, according to Bayes rule:

$$p(x \mid y_1, \ldots, y_N) = \frac{p_0(x) \prod_{n=1}^{N} S_n(y_n \mid x)}{\sum_{x \in \mathcal{D}} p_0(x) \prod_{n=1}^{N} S_n(y_n \mid x)}. \tag{6.7}$$

In our case, this evaluation tells us that, given the runtimes of the two algorithms, the probability that the instance is SAT is now only $p(0) \approxeq 0.06$. The runtime of `G2WSAT` is especially informative: compare its RTD on the set at $t = 0$ (Fig.6.1(b)), and at $t = 5 \times 10^5$ (Fig.6.6(b)). In the first case, the RTD is divided in two parts of equal probability: the part below the median corresponds to the SAT instance. The RTD on the set is improper, and remains at 0.5, the prior probability of the instance being SAT, and solvable by `G2WSAT`. The original function is shifted down as the time passes, and by the time we look at Fig.6.6(b), the horizontal line corresponding to the median has gone down to 0.06, the current value of $p(0)$. The same can be said about the RTDs of `Satz-Rand`, and of the portfolio.

Evaluating the allocators at this point will obviously give different results. The expected time allocator (Fig. 6.7) confirms its preference for `Satz-Rand`. Note that now the correct surface (6.4) almost corresponds to the one for the UNSAT instance (Fig. 6.7(a)), and has its optimum at $s_1 = 1$. The surface optimized by the wrong formula (6.3) is only slightly different at this point, and has again a similar minimum, at $s_1 = 0.99$. The lower quantile and median TAs have instead changed completely. At $t = 0$, both allocators favored `G2WSAT`, given its RTD: now the higher quantile that this algorithm can offer is down at 0.06, so both allocators will favor `Satz-Rand`. In the first case ($\alpha = 0.25$, Fig. 6.8), the correct allocation is again almost equivalent to the one for the UNSAT instance, $s_1 = 1$; the wrong formula optimizes a visibly different curve, but with a very similar minimum, at $s_1 = 0.98$. The median and higher quantile are reported in

Fig. 6.9. In both cases, the share evaluated is the same one of the expected time allocator: $s_1 = 1$ and $s_1 = 0.99$ for the correct and wrong allocation respectively, corresponding to the portfolios in Fig. 6.7(a).

The behavior of the two lower quantile allocators is the most interesting in this case: they both would initially allocate all time to `G2WSAT`, and gradually increase the share to `Satz-Rand`. Indeed, this results in the best time allocation strategy for the SAT/UNSAT example, as it allows to profit from the short runtime of the incomplete solver if the instance is SAT, with only a small overhead on the much longer runtime of the complete solver if the instance is UNSAT. In this case the quantile allocator with $\alpha = 0.25$ would obtain the best performance. It is equally easy to construct examples where another time allocator would be better, and it is in general not clear how to select which allocator to use on a given time allocation benchmark. We will propose a simple method of automating this choice in the next chapter. In the next section, we will discuss the second of the issues mentioned in the introduction to this chapter, namely the computational cost of sampling runtime distributions.

6.2 Sampling algorithm runtimes

In practice, even knowing the "right" features to use, the correctness of (6.1) will still depend on the fit of the models, which in turn will depend on the representativeness of the collected runtime sample, as well as on its size, which is also directly related to the computational cost of collecting it. The consideration made when describing censoring (Sec. 3.2), hold also in this case: there is a *trade-off* between training time, mostly spent in solving several training instances, and model precision.

In the context of time allocation methods, the aim of modeling is not to have a precise description of algorithm performance, but rather to guide the selection of the fastest algorithm. It is therefore more appropriate to consider the trade-off between training time and the *performance* achieved using the model: when minimizing solution time, the two quantities also share the same unit, time.

To test this intuition, in Section 8.2 we analyze this trade-off in the context of restart strategies (Sec. 2.4.2), reporting the training times, and resulting performance, of a model-based restart strategies, learned with different levels of censoring. The results are quite impressive: while the precision of the models decreases dramatically, the performance of the restart strategy remains practically unaltered until very high levels of censoring. Motivated by these encouraging results, we decided to exploit censored sampling in order to reduce the time cost of learning performance models.

Consider again the time allocation problem described in Section 5.1, i.e. a set $\mathcal{B} = \{b_m\}$ of M instances to solve, a set $\mathcal{A} = \{a_n\}$ of N algorithms, and Z processors. Additionally, we are given a set $\mathcal{M} = \{\hat{S}_n(t \mid \mathbf{x}_n)\}$ of N regression models (Sec. 3.4), which can be used to model the RTDs of the algorithms $\{a_n\}$ on problem instances, based of instance features[2].

A first idea to exploit such models may be to learn the RTDs of some of the available instances, in order to speed up the solution of the remaining ones. In practice, this requires using each a_n to solve a subset of M_0 instances, one or more times, in order to collect a *runtime sample* D_n. To save some computation time in this phase, we may censor exceedingly long runs: one advantage of survival analysis modeling techniques is that they allow to correctly take into account

[2] Note that different features $\mathbf{x}_n \in \mathbb{R}^{d_n}$ may be considered for different algorithms.

the time spent in unsuccessful runs as censored runtimes. While such censoring has an impact on model precision, it is not as bad as discarding long runtimes, or other heuristic approaches.

From a quantitative point of view, there are at least four design decisions which can affect both the cost of learning, and the performance of the learned models: the number of training instances, the algorithms to be considered for each instance, the number of runs for each algorithm, and the censoring threshold used. We will deal with the first choice in the next chapter, where we circumvent the issue adopting an *online* learning approach. The third was taken heuristically, simply performing one run for each algorithm.

Regarding the last decision, fixing a threshold in advance, as in type I censoring (Sec. 3.2) allows to set the duration of the training phase in advance, but has the disadvantage of requiring a guess of the typical runtimes for each instance, with the risk of wasting computation time in collecting more runtimes on very easy instances, which can be easily solved by all algorithms, and only a few on the hardest instances. Alternatively, the whole set of algorithms can be used to solve a training instance in parallel, with a uniform share $\mathbf{s}_{\mathcal{U}} = (1/N, 1/N, ..., 1/N)$, and the fastest algorithms can work as a censoring mechanism for the slower ones, as in a type II censoring experiment.

When one of the algorithms solves the current problem instance, its runtime can be used as an uncensored value to update its RTD model; continuing the execution of the remaining algorithms allows to collect more uncensored runtimes. The second question can then be reformulated in this way: how many algorithms should we wait for, in order to obtain their uncensored runtime? Also this question poses an exploration-exploitation trade-off, as gathering further uncensored runtimes allows to improve the RTD models, but has a significant time cost. If the fastest algorithm terminates in a time t_I, the time spent by the uniform portfolio will be $N t_I$; another uncensored runtime t_{II} would cost an additional $(N-1)(t_{II} - t_I)$, and so on. Also this trade-off will be addressed heuristically: for simplicity, and to keep time spent to the bare minimum, we will only collect one uncensored runtime sample per problem instance.

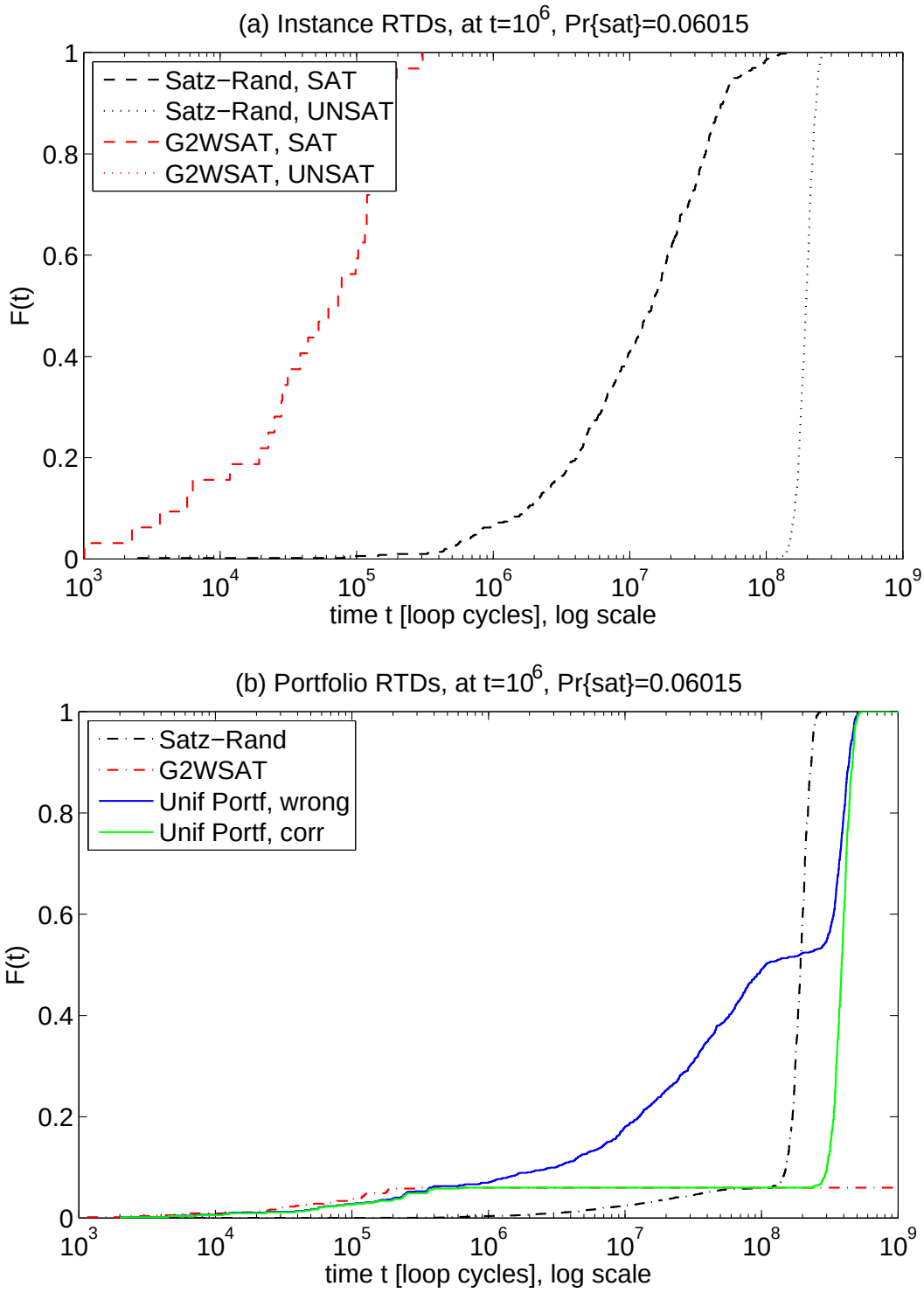


Figure 6.6. Effect of dynamic updates. (a) RTD of the two algorithms on the two instances. The line for G2WSAT UNSAT is constant at 0, as this algorithm cannot prove unsatisfiability. (b) RTD of the algorithms and uniform portfolio on a randomly picked instance. Comparison of correct and wrong evaluations. See text for details.

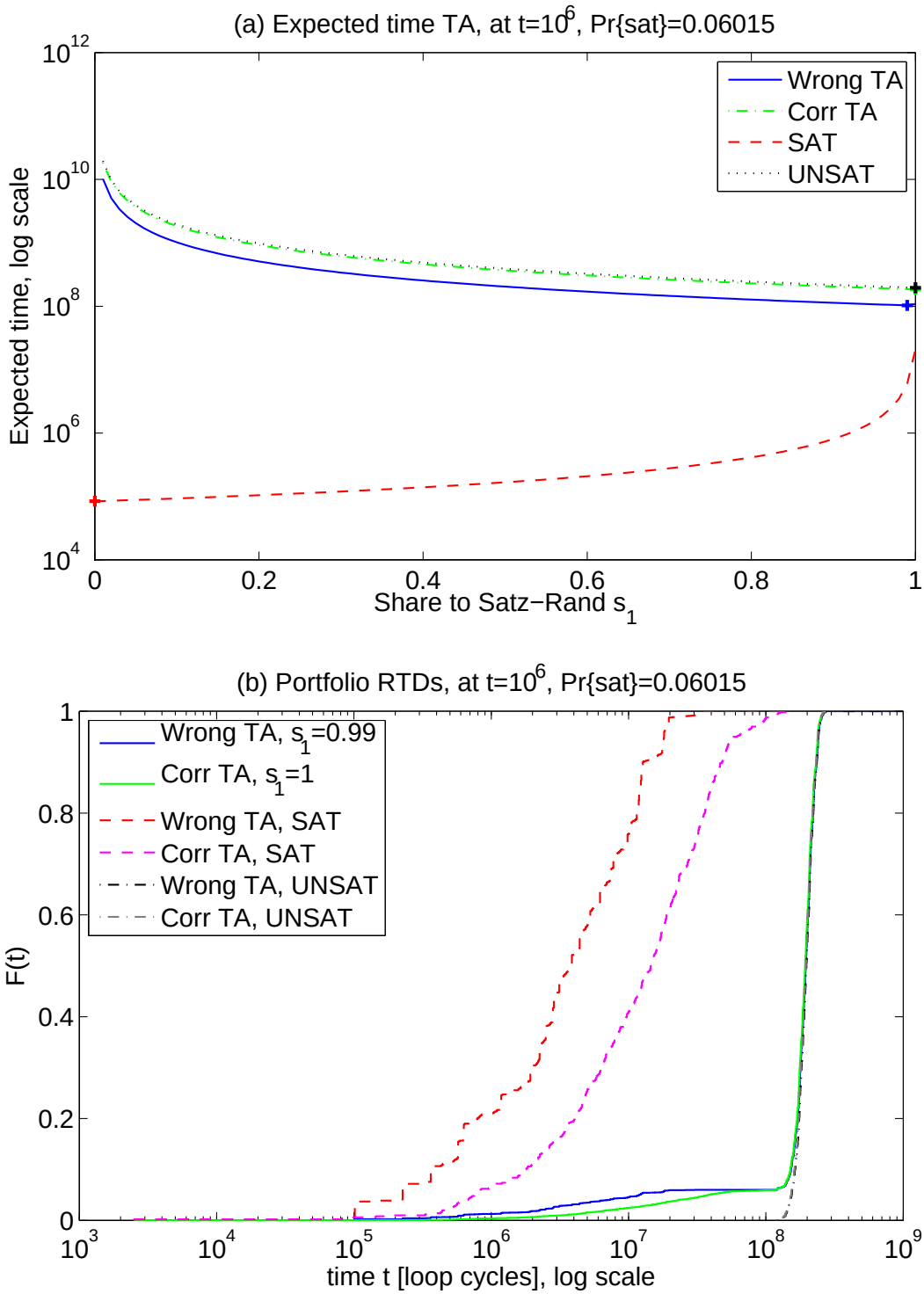


Figure 6.7. EXPECTED TIME: Effect of dynamic update. (a) Function optimized. (b) Optimal allocation. See text for details.

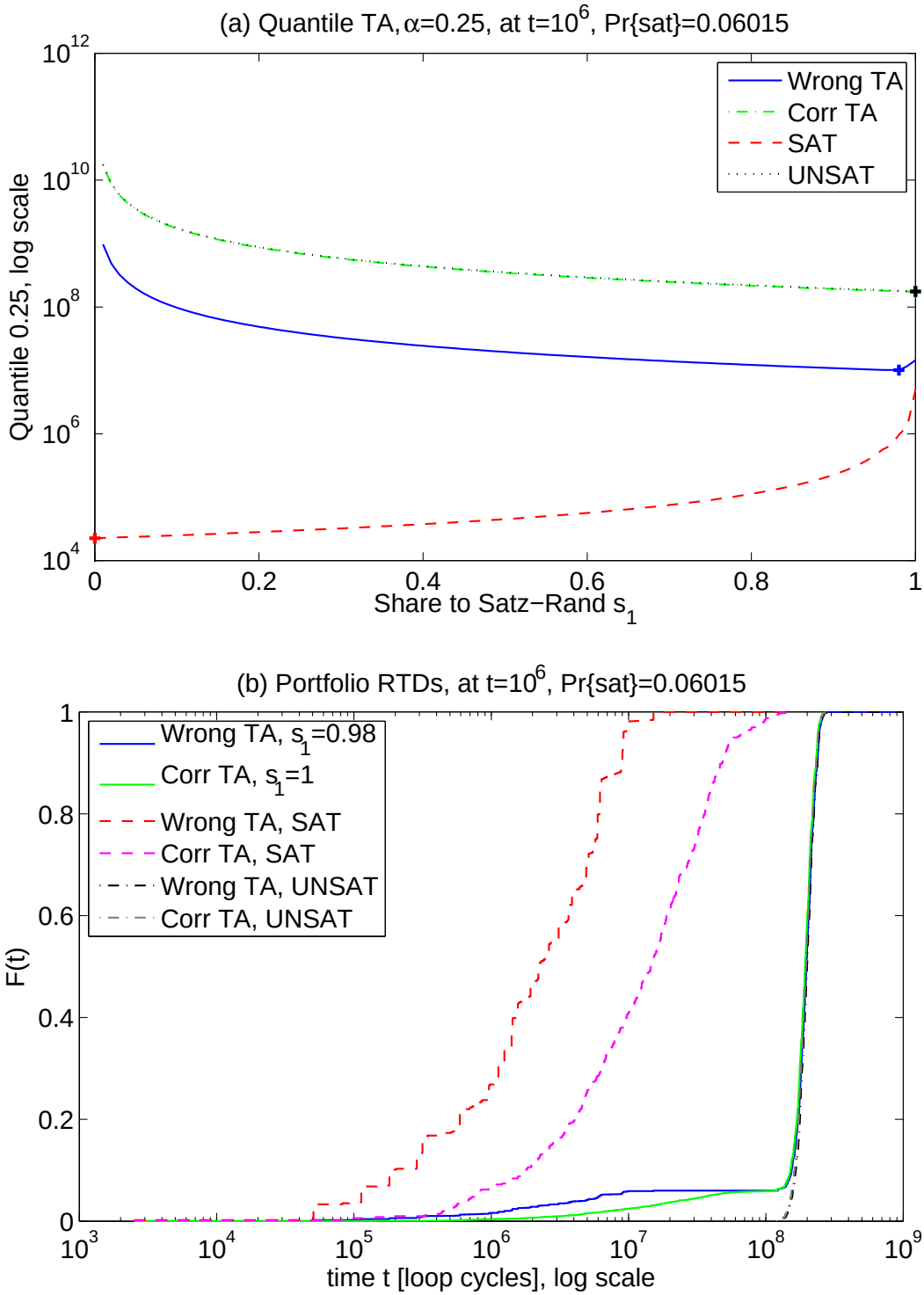


Figure 6.8. QUANTILE ($\alpha = 0.25$): Effect of dynamic update. (a) Function optimized. (b) Optimal allocation. See text for details.

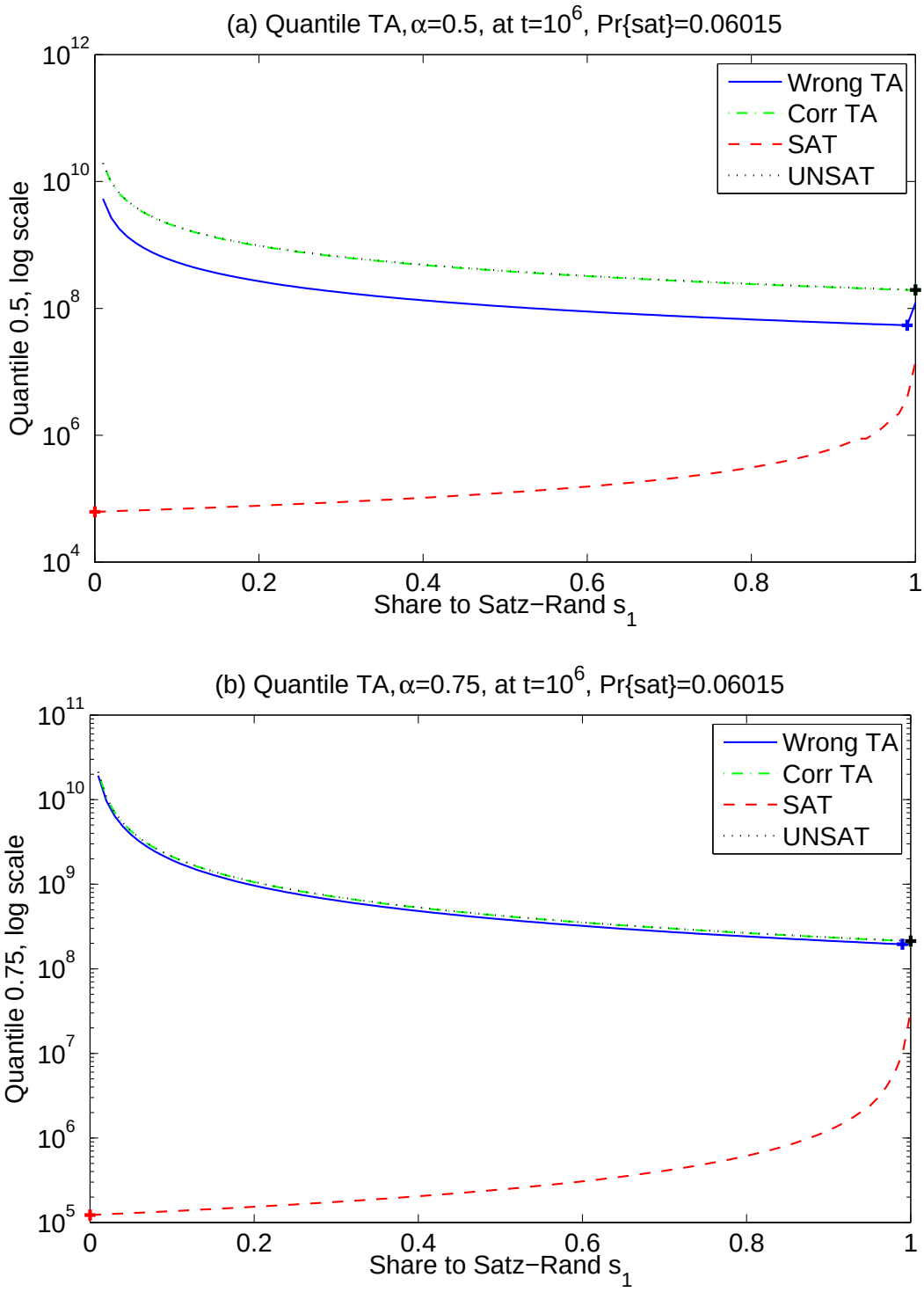


Figure 6.9. QUANTILE ($\alpha = 0.5$): Effect of dynamic update. (a) Function optimized. (b) Optimal allocation. See text for details.

6.3 Algorithms as competing risks

One potential drawback of using a portfolio to sample runtimes, is that it may induce a bias in the RTD models, as the runtimes of different algorithms on the same problem instance cannot be considered independent. In medical terms, we are viewing each instance b_m as a patient, with covariates $\mathbf{x}_m$, and the N algorithms as competing risks, one of which will eventually "kill" the patient, and censor the runtime values for the other algorithms. From a statistical point of view, a potential disadvantage of using a portfolio to gather runtime data is that, as we saw in Section 3.6, the resulting models will be biased, as the censoring mechanism (the fastest algorithm) is not independent from the observed events (the runtimes of all the algorithms): in practice, the survival probability of slower algorithms will be overestimated.

In the previous section we discussed another practical issue posed by the correlation of runtimes of different algorithms on the same instance, which does not allow to evaluate the RTD of the portfolio correctly. Also in this case, solving a *different* instance with each algorithm $a_n \in \mathcal{A}$ would solve the issue: the runtimes would be independent, and the resulting models would be correct.

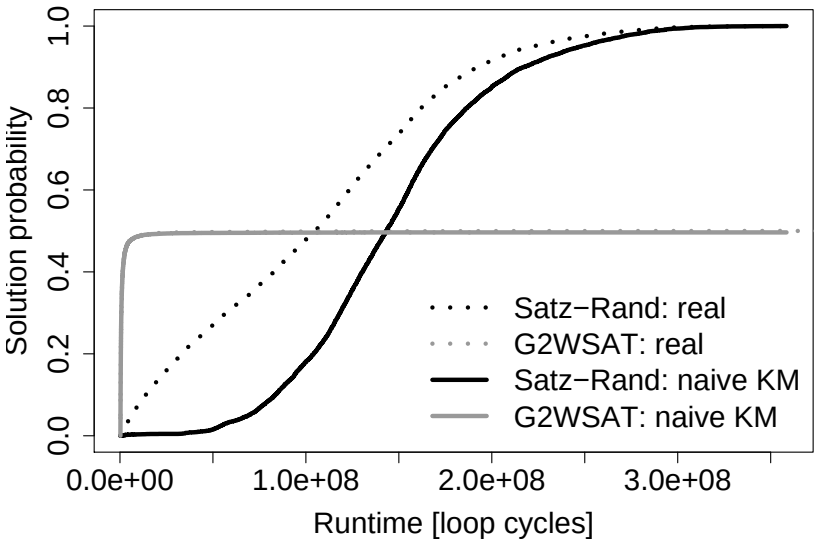
To illustrate the bias induced by competing risks, in Fig. 6.10 (a) we display the unbiased RTDs of the two algorithms of the SAT/UNSAT example (dotted lines), this time evaluated on a set of 200 instances of the same size ($n = 250$), solving each instance 100 times with each algorithm. On the same axes, we plot the KM estimates (3.16) of the RTDs of the two algorithms obtained with the portfolio approach, that is, censoring the runtime of the slowest algorithm for each run and each instance (continuous lines). While the model for `G2WSAT` remains accurate, as it mostly gets censored on unsatisfiable instances on which it would run forever anyway, one can clearly notice the bias of the product-limit estimator for `Satz-Rand`: the runtime is overestimated, especially for the satisfiable instances, on which this algorithm is slower, so it gets censored.

To reduce the bias, we repeated the sampling, randomly reordering the instance sequence for `Satz-Rand`: in this way, the two algorithms run in parallel, but each on a different instance. This reduces the statistical dependence among the runtimes of the two algorithms, allowing to consider the censoring mechanism uninformative, resulting in a more correct estimate. In Fig. 6.10 (b), we display again the "real" runtime distributions (dotted lines), compared against the estimates obtained after random reordering of the instances. The estimator for `Satz-Rand` is visibly more accurate on the whole range of runtimes observed.

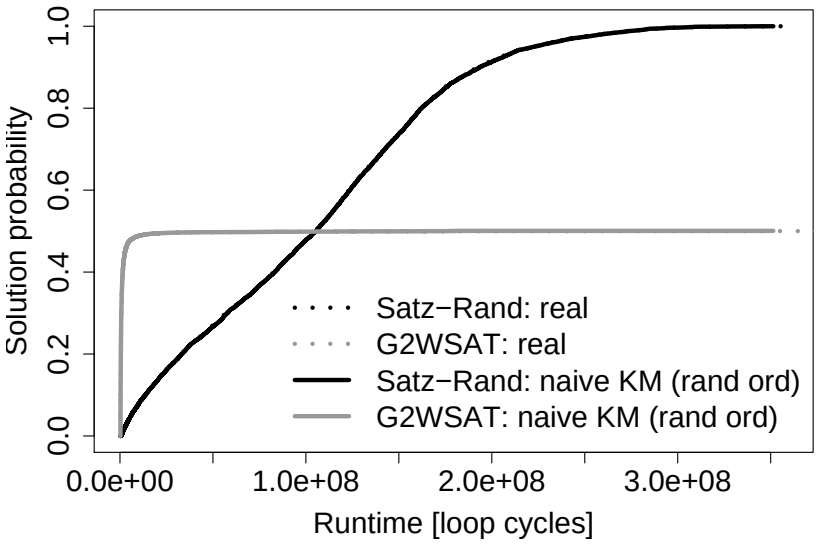
Also in this case, a price has to be paid for correctness: reordering the instances at random we loose the portfolio effect, which allows to save time on the SAT instances, where `G2WSAT` is much faster. Also in this case, we should not be concerned about the precision of the models, but rather on the time performance which can be attained allocating time based on the models, and this performance measure should also account for training time. In this sense, we will see in Part IV that the impact of the bias induced by the competing risks is indeed negligible (Sec. 9.7).

6.4 Related work

While there are other methods which allow to perform per instance selection, based on instance features (see Sec. 2.3), we are not aware of previous work on time allocation where the RTDs are estimated. In work where the instance RTDs are used to allocate time, these are assumed to be available beforehand, and the problem of sampling them efficiently is not explicitly addressed



(a) Same instances



(b) Random reordering

Figure 6.10. (a) The unbiased RTDs (dotted lines) of `Satz-Rand` (black) and `G2WSAT` (gray), compared with the biased estimates (continuous lines) obtained censoring, for each instance and each seed, the runtime of the slowest algorithm. Note the bias in the model for `Satz-Rand`. The two lines are nearly identical for `G2WSAT`. — (b) Same, with random reordering of the instances. Note the improvement in the model for `Satz-Rand`; in this case the estimates are nearly identical for both algorithms.

(Sec. 2.4). Some limited estimation is performed by Kautz et al. [2002], who model a binary RTD, distinguishing among short and long runs, respectively below and above the median for the training set. We proposed to learn the full RTDs in [Gagliolo and Schmidhuber, 2005], where parametric models of the form (3.17) were used.

In work on empirical hardness models (see Sec. 2.3.3), the models predict the logarithm of expected time. In the most recent versions, *ridge regression* [Bishop, 2006] is used to model such values. This method assumes errors to be normally distributed around the prediction. The whole selection technique, then, may be seen as equivalent to modeling RTDs of the algorithms with a lognormal distribution (2.10), and selecting the algorithm which minimizes expected runtime. In this sense, our time allocators are more general, as they can be correctly applied to algorithms with an arbitrary RTD: when problematic, parametric assumptions can be avoided, using non-parametric regression models instead.

Recently, Xu, Hoos and Leyton-Brown [2007] showed that instance features of the satisfiability problem do allow to estimate the probability that an instance is satisfiable, with an accuracy which is surprising, given the fact that the estimate takes polynomial time, and that satisfiability is an NP-complete decision problem. These authors use this estimated probability to mix the predictions of two separate models of the expected runtime, learned on SAT and UNSAT instances respectively. The method proposed in Section 6.1 to deal with unobserved features is in a sense analogous: in that case, the models being mixed are RTD models, and the prediction of the satisfiability of an instance is based on the current runtimes of the algorithms solving it. These two ideas could be combined, using a feature-based estimate of satisfiability as a prior, and updating it dynamically, based on the current runtimes of the algorithms.

In work on per-set optimal allocation, as [Petrik and Zilberstein, 2006; Sayag et al., 2006; Streeter et al., 2007], runtime values of training instances are used directly to evaluate the allocation (Sec. 2.5). The resulting allocation is per set: per instance allocation is proposed by Streeter and Smith [2008], limited to discrete features. Our time allocators can be conditioned on discrete or continuous features.

Regarding the use of censoring, it is simply ignored in most existing works on time allocation. Censored runtimes are either discarded, considered as uncensored [e.g., Xu, Hoos and Leyton-Brown, 2007], or set to an arbitrary high value [e.g., Petrik and Zilberstein, 2006]. Recall that in all these methods no attempt is made to reduce sampling time, so extensive experiments are run, with high censoring thresholds, and the runtimes censored are only a small portion of the resulting sample. Recently Xu, Hutter, Hoos and Leyton-Brown [2007] adopted a correct estimation method to deal with censored sampling [Schmee and Hahn, 1979], showing an improvement over previous heuristic approaches. Petrik [2005a]; Petrik and Zilberstein [2006] set censored runtimes to an arbitrary high value: this is not incorrect in the context of their method, which is not aimed at estimating the RTD, but directly at finding the per set optimal share. Other authors devised ad hoc statistical methods for taking into account censored runtimes. Fink [2004] presents an heuristic algorithm selection method where the expected runtime is estimated also based on unsuccessful runs. Streeter [2007] directly estimates the survival probability of a task switching portfolio based on multiple runs for each instance, with exponentially decreasing censoring thresholds. We adopted censoring as a tool to reduce the cost of sampling runtimes since [Gagliolo and Schmidhuber, 2005].

6.5 Discussion

In the previous chapter we presented exact methods for allocating computation time to a set of algorithms on a given problem instance, based on the instance RTDs of the algorithms. In this chapter, we took several steps towards the practical implementation of such allocators, discussing the use of RTD models in place of the unknown actual RTDs, and related issues. We focused on two aspects in particular: the impact of model precision on time allocation performance; and the computational cost of learning the models.

Instance RTDs can be approximated using regression models from survival analysis, as those exemplified in Section 3.4, conditioned on features of the instances. This approach to per instance selection is analogous to the 30 years old idea of Rice [1976], which has proved a successful paradigm in single algorithm selection (Sec. 2.3). The novelty here is represented by the used of models of the full RTDs, in place of simpler models of the expected time: this allows time to be allocated in a more general and principled way, analogous to the algorithm portfolios described in Section 2.4. Depending on the RTDs at hand, and on the criterion used, the result of the allocation can still be the selection of a single algorithm, or the parallel execution of several algorithms, according to a more general resource sharing schedule.

As expected, using estimates in place of the unknown real distributions turns the optimal allocators of the previous chapter into principled heuristics (Sec. 6.1). In practice, the models have to be learned based on a runtime sample, obtained collecting the runtimes of each algorithm on several training problem instances. The prediction of the RTDs on a novel instance is evaluated based on the available sample, and will depend on the similarity among the features of the instance, and the features of each training instance. The resulting estimate will be more or less correct, depending on the sample, and on the features used.

If the features do not allow to discriminate among instances where the algorithms have sensibly different RTDs, this may allow a correlation among the runtimes to pass undetected. In Section 6.1, we exemplified the effects of such a correlation. In the simple scenario considered, we could observe that, while the prediction of the models are visibly incorrect, they still allow to evaluate a near-optimal allocation. We also investigated the effect of updating the allocation dynamically, showing a potential reduction of runtime over a static allocation. We will further investigate these topics in the experimental section (Part IV).

In survival analysis, as in statistics in general, models are evaluated based on how well they predict new data. In our application, the only aim of performance modeling is to allocate time in an efficient way. In this sense, we should evaluate modeling methods only based on how well they allow to perform on unseen instances. This performance will obviously be related to the precision of the models, but this relation is not necessarily a strong one.

The issue of sampling algorithm runtimes efficiently was discussed in Section 6.2. The idea of reducing the length of a sampling experiment censoring excessively long observations, a common practice in failure analysis, can also be applied to runtime sampling: each training instance can be solved with a parallel portfolio, observing only the runtime of the fastest algorithm(s), and censoring the rest. The minimum runtime in this phase is attained collecting a single uncensored runtime for each instance: we will use this approach in our time allocation methods. Also in this case, runtimes correlation poses an issue, as it may induce a bias in the models, due to the competing risks effect (Sec. 3.6).

To summarize, a parallel portfolio can be used both for sampling algorithm runtimes on training instances, and for minimizing the runtime on new instances. During the sampling phase, a uniform share can be used. The collected runtime sample can then be used to learn regression

models of the RTDs. On a new instance, these models can be used to estimate the RTDs of the algorithms, and the share can be evaluated according to one of the time allocators described in the previous chapter. In both cases, correlations among the runtimes of the algorithms is a potential problem. The impact of this correlation can be low in practice, as the experiments in Part IV will show.

The next decision to be taken is: at which point should we stop the sampling phase, and start using the models to perform allocation? Rather than answering this question, we can instead adopt an online approach, updating the models after each instance is solved, and using them to allocate time on the next instance. In the next chapter we will discuss this idea in more detail, and present two example time allocators which combine our contributions.

Chapter 7

Online time allocation

In the previous chapters we presented time allocators based on the exact knowledge of the RTDs of the algorithms (Ch. 5), and discussed the use of regression models of these RTDs (Ch. 6) for their practical implementation. Such models have to be learned from runtime samples, collected by solving several problem instances. In this chapter we propose to reduce the computational complexity of runtime sampling, adopting an online approach, in which sampling and allocation are integrated in a single method, and the runtimes observed so far are used to allocate time on the next problem instance. Our approach involves the use of a bandit problem solver (Ch. 4), to balance among the exploration of algorithm performance, and exploitation of the knowledge acquired so far.

This chapter is organized as follows. Section 7.1 describes the general idea of performing online allocation using a bandit problem solver. In Section 7.2, we start by considering a simpler setting, where a model based restart strategy (Sec. 2.4.2) is combined with a uniform restart strategy, as two arms of a bandit. The model is updated based on the runtimes observed during a sequence of problems, and used to evaluate an optimal uniform strategy. The bandit problem solver compares the performance of the two strategies, and uses the model based one more frequently as its performance improves. Section 7.3 introduces a more advanced per instance allocation scheme, in which the selection is performed among different time allocators. Section 7.4 describes an earlier, more complex version of this idea, where the shares evaluated by the various allocators are mixed.

In all these cases, runtimes can be used as a loss, to be minimized by the bandit problem solver. As discussed in Section 4.3, the huge variability of algorithm runtimes makes it difficult to use most available solvers, which assume losses to be bounded. Section 7.5 addresses this issue, presenting a modification of an existing solver that allows to prove a bound on regret which holds for arbitrarily large losses. Section 7.6 discusses related work on online allocation, and Section 7.7 concludes the chapter.

7.1 Time allocation as a bandit problem

The necessity of runtime sampling is common to any non-oblivious allocator, model-based or not: in order to save computation time in the future, allocation has to take into account results obtained in the past. In this sense, a disadvantage of any offline approach, consisting in solving training instances repeatedly, one or more times with each algorithms, is represented by its

computational cost, which can be relevant. To be useful in practice, a time allocation method should explicitly aim at reducing the computational complexity of this preliminary sampling phase.

Intuitively, the more data is collected, the more accurate will be the allocation on future instances, and the higher the cost of the sampling phase. In other words, we are facing a trade-off between the *exploration* of the performance of the various a_i, and *exploitation* of this performance, based the collected data. A well-known paradigm for dealing with such a trade-off is the multi-armed bandit problem (Ch. 4), where the choice among alternatives based on limited experiments is examined in the online setting.

In Sections 4.4 and 4.5, we discussed the application of a BPS to time allocation, concluding that it can only be used to implement a per set method, which allows to favor a single one among several alternatives, based on the performances observed on a sequence of trials. These performances can change over time, or even be deceptive in a worst case setting, but the choice among alternatives cannot be done on a per instance basis, nor the alternatives can be combined. Per instance selection is in general more efficient than per set selection (Sec. 2.3), and sharing resources among many algorithms can sometimes be preferable over selecting a single one (Sec. 2.4). Recall the results of the SAT competition discussed in Section 1.3. In that situation, a BPS could only hope to reduce its regret compared to the best algorithm, which would have been outperformed by a uniform portfolio of all contestants in parallel. For these reasons, a BPS should not be used to select directly among algorithms.

In the previous chapter, we also saw a first idea for reducing sampling time, inspired by censoring (Sec. 3.2). As failure analysts do when testing light bulbs, we can run a portfolio with a uniform share $\mathbf{s}_{\mathcal{U}} = (1/N, 1/N, ..., 1/N)$ on each training problem instance. Instead of waiting for all the algorithms to end, we stop after the first one solves the problem, and switch to the next. As discussed in Section 3.2, this has an impact on the accuracy of the model, but the uniform share would at least assure that the fastest algorithm would not be censored. In this way the model would be less accurate for less efficient algorithm/problem combinations.

Consider again the time allocation problem problem represented by $\mathcal{B}$ and $\mathcal{A}$ (Sec. 5.1). To summarize the findings of Chapters 5 and 6, we now have several ways of allocating time to $\mathcal{A}$ based on a set of models $\mathcal{M}$ of the RTDs.

Suppose we fix a choice of a model-based time allocator $\text{TA}_{\mathcal{M}}$. We can start solving instances with the uniform portfolio $\text{TA}_{\mathcal{U}}$, whose runtime is always N times that of the per instance best algorithm, which, as we will further confirm experimentally, makes its performance already competitive. After some instances have been solved, the models will be reliable enough to allow the time allocators to achieve a good performance: the issue is that we do not know *how many* instances we should wait before starting to use the $\text{TA}_{\mathcal{M}}$. The performance of the uniform allocator $\text{TA}_{\mathcal{U}}$ will not change along the instance sequence. The performance of $\text{TA}_{\mathcal{M}}$ will be poor in the beginning, but we expect it to improve as the model improves, eventually outperforming $\text{TA}_{\mathcal{U}}$.

We are in a situation in which we would like to select the per set best alternative, among $\text{TA}_{\mathcal{M}}$ and $\text{TA}_{\mathcal{U}}$, and the per set best changes over time. We can model this problem as a non-stationary bandit problem: to include the worst-case possibility of a "deceptive" instance sequence, we can consider the problem as adversarial. In both cases, we can use a bandit problem solver to choose, for each subsequent instance, among $\text{TA}_{\mathcal{M}}$ and $\text{TA}_{\mathcal{U}}$. If $\text{TA}_{\mathcal{M}}$ does not converge to a good performance, the BPS will keep favoring $\text{TA}_{\mathcal{U}}$, minimizing the regret with respect to its total runtime.

In this way, problem solving and runtime sampling can be integrated, in an online fashion:

each time an instance is solved, we can update the runtime sample with an uncensored observation and $N-1$ censored observations. Indeed, we do not even need to choose an $\text{TA}_{\mathcal{M}}$ ourselves: we can leave this choice to the BPS, adding alternative allocators as additional arms.

This same approach can be used to turn any offline non-oblivious allocator into an online method, simply combining it with the uniform portfolio, and updating the sample collected for each solved instance. The allocators we proposed earlier have the advantage of being based on RTD models, which can be correctly updated also based on censored information. This justifies the use of the portfolio also as a sampling tool, as it represents a good compromise among sampling cost (no instance is solved twice) and model precision (censored runtimes can be correctly accounted for).

Before applying this idea to our time allocators, in the next section we consider a simpler setting, in which a model-based restart strategy is learned online.

7.2 GambleR

In section 4.4 we saw that a bandit problem solver can be used to perform per set algorithm selection. In this section we present an example of a similar selection, but among restart strategies (Sec. 2.4.2). GAMBLER (Alg. 7) is an online method for learning an optimal restart strategy. It consists in alternating the two strategies proposed by Luby et al. [1993], the oblivious universal strategy, and the estimated optimal strategy. The estimate is performed minimizing the expected runtime of the uniform strategy (2.12), evaluated based on a model $\hat{F}$ of the RTD. Such model can be updated each time an instance is solved, using the durations of unsuccessful runs as censored runtimes. Depending on the model used, the resulting method may be per set, if the set RTD is learned, or per instance, if a regression model of the RTD is learned instead.

Here the bandit problem solver is used to select among two restart strategies, representing the two arms. Each strategy proposes a different sequence of restart thresholds for the same randomized algorithm. Each run of the algorithm corresponds to a trial. The associated loss is 0, except when the instance is solved: in such case, the loss is the total time t_k spent by the strategy k being used, including unsuccessful runs.

Algorithm 7 GAMBLER(M) Gambling Restart for algorithm a. t_a indicates its runtime.

```
Universal RS τ_1(1), τ_1(2), ... [Luby et al., 1993]
Estimated optimal RS τ_2, based on RTD model F̂ [Luby et al., 1993]
initialize BPS (2, M), p
for each problem 1, ..., M do
  set t_k := 0, j_k := 0, k = 1, ..., K
  repeat
    pick k ~ p
    run a with cutoff τ_k(j_k + 1)
    update counter j_k := j_k + 1, timer t_k := t_k + min{t_a, τ_k(j_k)}
  until problem solved by strategy k
  observe loss l_k := t_k for winning strategy
  let BPS update p
  update F̂ based on collected runtime data
end for
```

The ratio behind this approach is that, as soon as the RTD model is good enough to allow the model-based optimal restart strategy to outperform the universal strategy, the BPS will select the former more often.

In other words, the RTD model is learned online, and the distinction between an initial exploratory "training" phase, during which the model is learned from scratch, and the subsequent exploitation phase, during which the model is used to evaluate an optimal RS, is not set in advance: the BPS will switch gradually from exploration to exploitation, based on how well the model-based RS performs with respect to the universal RS. In the worst case, if the model does not allow for a good performance, the BPS will stick to the universal strategy.

GAMBLER is general in the sense that it can be applied to an arbitrary set of K strategies, model-based or not. Moreover, it can be turned into a per instance method, simply using a regression model for the RTD of the algorithm, and conditioning it on instance features.

Note that in this case each trial of the BPS corresponds to a run of the algorithm, such that the same instance can take several trials to be solved. Moreover, we require the BPS to keep a nonzero probability over each arm. This is to keep some guarantees on performance also in a worst-case scenario in which the threshold evaluated by the uniform strategy is too small: the BPS ensures that the universal strategy will be used now and then, and the instance will eventually be solved.

Let us now discuss how the bound (2.13) can be used to evaluate a worst-case bound on the performance on GAMBLER, if the universal strategy is included as one of the arms, and the BPS used features a minimum exploration probability, such that $p_k \geq p_{\text{min}}$ for all trials.

Theorem 4. Worst case performance of GAMBLER**.** *On a given problem instance, if R_U is a bound on the number of runs of the universal restart strategy, t_U the bound on its runtime (2.13), and the BPS plays according to a* $\mathbf{p}$ *such that $p_k \geq p_{min}$ for all arms k, and for all trials, then the runtime t_G of* GAMBLER *on the instance is bounded in expectation as*

$$E\{t_G\} \leq E\{t_U + R_U \hat{\tau} \frac{1 - p_{min}}{p_{min}}\} \tag{7.1}$$

The proof is given in the appendix (Sec. A.2). Section 8.3 reports experimental results obtained with GAMBLER, which prove to be much better than this pessimistic bound.

7.3 GambleTA

We already saw two examples of the use of a BPS for performing a *per set* selection: among algorithms (GAMBLEAS, Sec. 4.4), and among restart strategies (GAMBLER). In the previous chapters, we saw examples of model-based time allocators, able to perform algorithm selection on a *per instance* basis. In this section we address the problem of learning the necessary RTD models online.

Consider a set $\mathcal{T}$ of K arbitrary *time allocators*, as defined in Section 5.2, $\mathcal{T} = \{\text{TA}_1, \ldots, \text{TA}_K\}$. At this higher level, one can use a BPS to select among different time allocators, working on a 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 Alg. 8, where $t_k(m)$ is the runtime of TA_k on instance b_m.

Algorithm 8 GAMBLETA $(\mathcal{A}, \mathcal{T}, BPS)$ Gambling Time Allocator.

Algorithm set $\mathcal{A}$ with N algorithms
Problem set $\mathcal{B}$ with M instances
A set $\mathcal{T} = \{\text{TA}_k\}$ of K time allocators
A bandit problem solver BPS

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

An interesting feature of this selection scheme is that the additional requirement that each algorithm should be capable of solving each problem, introduced for GAMBLEAS, can be relaxed again, requiring instead that Hypothesis 1 is satisfied, i.e. *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[1] imposing a $s_n > 0$ for all a_n. This allows to use interesting combinations of complete and incomplete solvers in $\mathcal{A}$, as in the SAT/UNSAT example (Sec. 6.1.1,9.5). No additional hypothesis on $\mathcal{T}$ is required, except that it includes the uniform allocator.

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 as the uniform share is included in the set of TAs. Additionally, one could add a set of N trivial allocators, each giving all computation to a single algorithm a_i: this would require to restrict again Hypothesis 1, imposing that each algorithm can solve each instance.

In the next section we describe an earlier version of GAMBLETA, along with reasons for abandoning it. The reader may optionally skip to the following Section 7.5, where we address the issue of unbounded losses.

7.4 Time allocators as experts

The original version of GAMBLETA [Gagliolo and Schmidhuber, 2006c] was based on a more complex alternative, inspired by the bandit problem with expert advice, as described by Auer et al. [1995, 2002]. In that setting, 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 our

[1] 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 not enough observations larger than the current runtimes), the allocation defaults to the uniform share. In this way, all algorithms will eventually be executed, satisfying the hypothesis.

case, the time allocators played the role of the experts, each suggesting a different $\mathbf{s}$, on a per instance basis; and the arms of the lower level game were the N algorithms, to be run in parallel with the mixture share. EXP4 [Auer et al., 2002] was used as the BPS. Unfortunately, the bounds of this solver could not be extended to GAMBLETA in a straightforward manner, as they require the loss function to be convex in $\mathbf{s}$, an hypothesis which is violated for time allocation, if the runtime of the portfolio (2.15) is used as a loss; moreover, EXP4 cannot deal with unbounded losses, so in [Gagliolo and Schmidhuber, 2006c] we had to adopt an heuristic reward attribution instead of using the plain runtimes. Attempts to devise a novel algorithm, inspired by EXP4, but using our concave loss function instead, always resulted in algorithms whose bound on regret was $O(KL^*)$, K being the number of time allocators, L^* the cumulative loss of the best one. This can be better clarified by analyzing a worst-case scenario. Consider a situation in which there are N algorithms of identical performance, and $K \leq N$ time allocators. Suppose that the k-th allocator always allocates all time to the k-th algorithm. Picking a random allocator would in this case give the optimal loss L^*, while mixing their allocations uniformly would share time equally among K equally performing, but distinct, algorithms, with a loss KL^*: and as all time allocators would score the same loss, the BPS would always keep the same uniform mixture of time allocators.

7.5 Unbounded losses

A common issue of the above approaches is the difficulty of setting reasonable upper bounds on the runtime of 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 (Sec. 2.4.1).

As seen in Section 4.3, only limited research has been carried out on how to deal with unbounded losses in a principled manner. The method of Allenberg et al. [2006] considers losses whose bound grows in the number of trials i with a known rate i^ν, $\nu < 1/2$. This 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. Such setting is considered in [Cesa-Bianchi, Mansour and Stoltz, 2005; Cesa-Bianchi et al., 2007], but in a full information game. A bound on the expected regret is obtained with a simple doubling trick: the unknown maximum loss is initially set to 1, and is doubled each time a larger loss is observed, resetting the loss estimates for each arm. This latter approach is general enough to be applied to other solvers as well, including those for partial information games. Cesa-Bianchi, Mansour and Stoltz [2005] also introduce an algorithm for loss games with partial information (EXP3LIGHT, see Sec. 4.2), which requires losses to lay in $[0, 1]$, and is particularly effective when the cumulative loss of the best arm is small. In this section, we apply the doubling trick to this algorithm, and present the resulting bound on regret.

Recall the notation used in Section 4.2: we consider a partial information game with K arms, and M trials; an index (j) indicates the value of a quantity used or observed at trial $j \in \{1, \ldots, M\}$; k indicate 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 $I(j)$ indicates the arm chosen at trial (j), so it is a discrete random variable with value in $\{1,\ldots,K\}$; r, u will represent quantities related to an *epoch* of the game, which consists of a sequence of 0 or more consecutive trials; log with no index is the natural logarithm.

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 (Algorithm 2), simply dividing all losses by $\mathcal{L}$. Based on Theorem 5 by Cesa-Bianchi, Mansour and Stoltz [2005] (4.3), it is easy to prove the following

Theorem 5. 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:*

$$\begin{aligned} & E\{L_E(M)\} - L^*(M) \leq \\ \leq\ & 2\sqrt{6\mathcal{L}(\log K + K\log M)KL^*(M)} \\ +\ & \mathcal{L}[2\sqrt{2\mathcal{L}(\log K + K\log M)K} \\ +\ & (2K+1)(1+\log_4(3M+1))] \end{aligned} \tag{7.2}$$

The proof is trivial, and is given in the appendix (Sec. A.3).

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 2) as a subroutine. EXP3LIGHT-A (Algorithm 9) is based on the doubling trick used by Cesa-Bianchi, Mansour and Stoltz [2005] for 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* using a bound $\mathcal{L}_u = 2^u$.

Algorithm 9 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) ∈ [0, L] ∀ i = 1,...,M, j = 1,...,K
unknown L < ∞
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 = ⌈log_2 l_{I(i)}(i)⌉
    restart EXP3LIGHT (K, M − i)
  else
    update EXP3LIGHT with loss (l_{I(i)}(i)/2^u) ∈ [0,1] for arm I(i)
  end if
end for
```

A bound for EXP3LIGHT-A can be derived from the bound for EXP3LIGHT:

Theorem 6. Regret of EXP3LIGHT-A.

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

$$\begin{aligned} & E\{L_E(M)\} - L^*(M) \leq \\ & 4\sqrt{3\lceil \log_2 \mathcal{L} \rceil \mathcal{L}(\log K + K \log M) K L^*(M)} \\ + \quad & 2\lceil \log_2 \mathcal{L} \rceil \mathcal{L} [\sqrt{4\mathcal{L}(\log K + K \log M) K} \\ + \quad & (2K+1)(1 + \log_4(3M+1)) + 2] \end{aligned} \tag{7.3}$$

The proof is given in the appendix (Sec. A.3). The regret obtained by EXP3LIGHT-A is $O(\sqrt{\mathcal{L} L^*(M) K \log M})$, which 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 EXP3LIGHT-A as a BPS for selecting among different time allocators in GAMBLETA (Algorithm 8). Regarding the multiple CPU version of the time allocators (Sec. 5.3), the *total* CPU time can be used as a loss (i. e., zt if z CPUs are used for a wall-clock time t), in order to favor time allocators that do not use more CPUs than necessary. Unfortunately, in this case the bound on the regret does not hold, as the proof assumes a *sequential* game, in which the probability distribution over the arms is updated after each arm pull, based on the observed loss. In the multiple CPU version of GAMBLETA, instead, the choice of time allocators continues until there are CPUs available, and the feedback on the loss is received asynchronously, only when the corresponding problem instance is solved.

7.6 Related work

In this chapter we proposed a principled online solution to the exploration-exploitation trade-off implicit in time allocation, initially proposed in [Gagliolo and Schmidhuber, 2006c], albeit in a slightly different form (see Sec. 7.4). In [Gagliolo and Schmidhuber, 2005], we addressed the trade-off heuristically, updating the model after each instance solution, and gradually shifting, through the problem sequence, from a uniform initial share to a model-based share, again heuristically evaluated.

An alternative offline approach can be based on the PAC learning framework [Mitchell, 1997], evaluating the minimum number of training instances required to guarantee a chosen level of performance on future instances. Such approach is taken in the per set, model-free methods of Petrik and Zilberstein [2006]; Sayag et al. [2006]; Streeter et al. [2007] (Sec. 2.5). Compared to our method, it has the advantage of giving statistical guarantees on future performance, with respect to to the best per set schedule, but it does not solve the problem of deciding how much should be learned. The final user is still required to set a regret measure in advance. This amounts to a different trade-off, which can be equally difficult in practice: setting the regret too low may require an unreasonable amount of learning. In this sense, our method has the advantage of not requiring any a priori guess of the difficulty of a particular instance of the time allocation problem. Moreover, in the above cited work, each training instance is solved multiple times with each algorithm, and the cost of this exhaustive sampling phase can be order of magnitudes higher than the cost of an uniform portfolio.

An orthogonal approach, also aimed at reducing training time, but keeping guarantees on the quality of the collected data, is *active learning*. This was recently proposed by Hutter [2009], in the context of a parameter tuning method based on local search.

The only other online methods which we are aware of are those proposed by Streeter et al. [2007]; Streeter and Smith [2008] (Sec. 4.5, see also [Streeter, 2007]). Both are based on a BPS, used in a different way. The runtimes are assumed to be bounded, and the bound is chosen arbitrarily: when trespassed, the authors propose to use a uniform portfolio to solve the current instance. In [Streeter et al., 2007], the BPS is a label efficient forecaster, and is used to decide whether to perform an exhaustive runtime sampling (solving the current instance with multiple runs for each algorithm), or to exploit a per set schedule based on the current runtime sample. This is indeed similar to our approach. In [Streeter and Smith, 2008], several BPS are used, one for each segment of a task switching schedule: when an instance is solved, a payoff is given only in the BPS corresponding to the last segment. This is analogous to using a sequence of GAMBLEAS (Alg. 3, Sec. 4.4). The resulting schedule is per set: Streeter and Smith prove it to be an approximation of their 4-optimal greedy schedule (2.23). The main advantage of our method is that it can perform per instance allocation, in a principled manner, based on discrete or continuous instance features.

Some of the design decisions characterizing GAMBLETA were inspired by the Optimal Ordered Problem Solver [OOPS, Schmidhuber, 2004]. OOPS is an online method for search in program space, where a set of candidate programs are executed sequentially, according to a task switching schedule. When a problem instance is solved, the successful program is stored on the input tape. For the following problem instance, two searches are run in parallel: half of the time is allocated to testing prolongations of the latest solver, the other half to fresh programs, starting from scratch. Time is allocated according to a distribution over programs, which is obtained multiplying the probabilities of the individual instructions. If the instruction set includes primitives that can modify these probabilities, then OOPS can in principle display meta-learning capabilities by finding programs that speed-up the search for future solutions.

7.7 Summary

In this chapter we described our online approach to time allocation. We started by pointing out the potential advantages of online over offline time allocation, as well as the *exploration-exploitation* trade-off arising in this context. We adopted the well-know framework of the *multi-armed bandit* problem to optimally address this trade-off. We first discussed an application to restart strategies (GAMBLER, Sec. 7.2). We then considered selection among alternative time allocators (GAMBLETA, Sec. 7.3). In both cases, two or more allocators represent the arms of the bandit. One of these allocators is oblivious, with a bound on performance w.r.t. the best possible allocator. For GAMBLER, this arm is represented by the universal restart strategy, with a bound w.r.t. the optimal strategy (Sec. 2.4.2). For GAMBLETA, it amounts to the uniform portfolio, i.e. the parallel execution of all N algorithms, which is always N times slower than the per instance fastest algorithm. The remaining arms are non-oblivious allocators, which are updated whenever an instance is solved. Each instance corresponds to a trial of the game: the bandit problem solver selects an arm/allocator, uses it to solve the instance, and observes the time spent as the corresponding loss.

The long term loss of a BPS approaches that of the best arm: in our case this means that if one of the non-oblivious allocators eventually outperforms the oblivious one, this fact will

eventually be exploited; if this does not happen, the overall performance will be comparable to that of the oblivious method.

One of the fundamental obstacles to the representation of algorithm selection as a bandit problem is that losses (i. e., runtimes) are inherently unbounded. After recalling the scarce literature on this topic (Sec. 4.3), we described a variation (EXP3LIGHT-A, Sec. 7.5) of an existing bandit problem solver (EXP3LIGHT, Sec. 4.2) that allows to deal with an unknown bound on losses, via a doubling trick, and derived a bound on its regret. The proposed solver is parameterless, and requires no hypotheses about the loss sequence: this means that it will guarantee a bound on regret regardless of the order and difficulty of the problem instances.

In short, GAMBLETA is a method to perform per set selection among different time allocators. Such allocators can be per instance portfolios, as those proposed in the previous chapters, but the method is general and can be applied to train arbitrary non-oblivious allocators online, and at the same time select the best performing one. In the next part we will present experiments with GAMBLER and GAMBLETA, showing that in most cases they considerably improve over the performance of the oblivious arm.

Part IV

Experiments

Chapter 8

Experiments with restart strategies

Heavy tailed RTDs pose a practical issue, which may be addressed combining multiple runs, either restarting the algorithm, or running multiple parallel copies in a portfolio (Sec. 2.4.2). In both cases, the optimal allocation is performed according to a model of the instance RTD of the algorithm. In practice, the RTD on a set of instances has to be used instead. In the following, we exemplify the impact of this approximation, and report the performance of our novel restart strategy, GAMBLER (Sec. 7.2), where the model is learned online.

Section 8.1 describes the benchmark used in the experiments, presenting plots of the instance RTDs. Section 8.2 reports preliminary experiments aimed at studying the impact of censored sampling on the performance of a model based restart strategy, showing that the relationship among model precision and restart performance is not a strong one. Section 8.3 presents experiments with GAMBLER. Section 8.4 discusses the results.

8.1 Satz-Rand on morphed graph coloring

The experiments described in this chapter were performed using the SAT solver `Satz-Rand` [Gomes et al., 2000] on a publicly available benchmark of graph-coloring problem instances, on which the algorithm displays heavy tailed RTDs [Gent et al., 1999]. Satz-Rand is a complete solver for the satisfiability problem, proposed by Gomes et al. [2000]. It is a randomized version of Satz [Li and Anbulagan, 1997], in which random noise influences the choice of the branching variable. Satz is an heuristic modification of the complete DPLL procedure, where variables are ordered based on first and second level unit propagation, and the first one is chosen for the next branching. Satz-Rand differs in that, after the list is formed, the next branching variable is randomly picked among the top h fraction of the list. All experiments were performed with the heuristic starting from the most constrained variables, as suggested also in [Li and Anbulagan, 1997], and the noise parameter set to 0.4.

The benchmark considered in this chapter consists of different sets of SAT-encoded "morphed" graph-coloring problem instances from [Gent et al., 1999], with 100 vertexes, 400 edges, 5 colors, resulting in 500 variables and 3100 clauses when encoded as a CNF3 SAT problem. Each graph is composed of the set of common edges among two random graphs, plus fractions $p \in [0, 1]$ and $1 - p$ of the remaining edges for each graph, chosen as to form regular ring lattices. In practice, the parameter p controls the amount of structure in the problem. Each of the 9 sets contains 100 instances, generated with a logarithmic grid of 9 different values for the

parameter p, from 2^0 to 2^{-8}, to which we henceforth refer with labels $0, ..., 8$. Gent et al. show that the behavior of Satz-Rand on the different sets varies according to the structure parameter p, displaying heavy-tailed behavior for some of its values.

We recorded the runtimes of 500 runs of `Satz-Rand` on each instance. As the CPU runtime measures are quite inaccurate [see also Hoos and Stützle, 2004, p. 169], especially for short runs, we modified the original code of the algorithm, adding a counter, that is incremented at every loop in the code. The resulting time measure was consistent with the number of backtracks. All runtimes reported for this algorithm are expressed in these loop cycles: on a 2 GHz machine, 10^{10} cycles correspond to about 4 minutes. The sampling was performed with a censoring threshold of 10^{12} cycles, corresponding to more than 6 hours. Only 7013 of the total 450000 runs were censored, for a total runtime of 4.9×10^{14} (about 130 days of CPU time).

In Figure 8.1 we display the log-log plot of the survival function of Satz-Rand on each instance, for each set of instances, using the same axes for all sets. The RTDs for sets 7 and 8 are very similar, so the latter is omitted. As other RTD plots in this thesis, also these were obtained from a non-parametric, product-limit estimate (3.16), which is correctly displayed as a stepwise function, constant among uncensored observations. Given the large number of runs, and the high censoring threshold, these plots are a good approximation of the real instance RTDs. Table 8.1 reports the number of censored runs for each set, along with the corresponding censoring portion, and the number of instances on which censoring was observed.

Apart from three long runs while solving set 0 (two on instance 23, one on instance 91), which were completed before the censoring threshold, the heavy tailed behavior of `Satz-Rand` is mostly evident on sets 1 to 5, on which some of the runs were censored, and the runtimes are spread across six orders of magnitudes, ranging from 0.01 seconds to more than 6 hours. In these plots you can see that the tail of the survival function can be approximated by a straight line, at least for a tract. This cannot be observed on the remaining sets, where all instances are solved before the threshold. Recall that the RTD of a complete solver cannot actually be heavy-tailed, as all runs will finish in a finite time, so the CDF will eventually reach 1. What makes restart strategies effective is the presence of a tract of the tail where the survival function decreases less fast than an exponential, corresponding to a straight line on the log-log plot, even when the tail eventually ends with an exponential decay.

Before solving this benchmark using GAMBLER (Sec. 8.3), in the following section we will show, via a simple experiment, that even a heavily censored runtime sample may allow to evaluate an effective restart strategy.

Set	0	1	2	3	4	5	6	7	8
Censored inst.	0	28	98	100	100	19	0	0	0
Censored runs	0	66	923	2925	3063	36	0	0	0
Portion of cens.	0	0.0013	0.0185	0.0585	0.0613	0.0007	0	0	0

Table 8.1. Censored runs of `Satz-Rand` on the morphed graph-coloring benchmark, with threshold $t_c = 10^{12}$ cycles. The benchmark is divided in 9 sets of 100 instances each. The table reports, for each set, the number of instances on which `Satz-Rand` was censored at least once over the 500 runs performed; the number of runs censored, over the total 50000; and the corresponding portion of censored data in the sample.

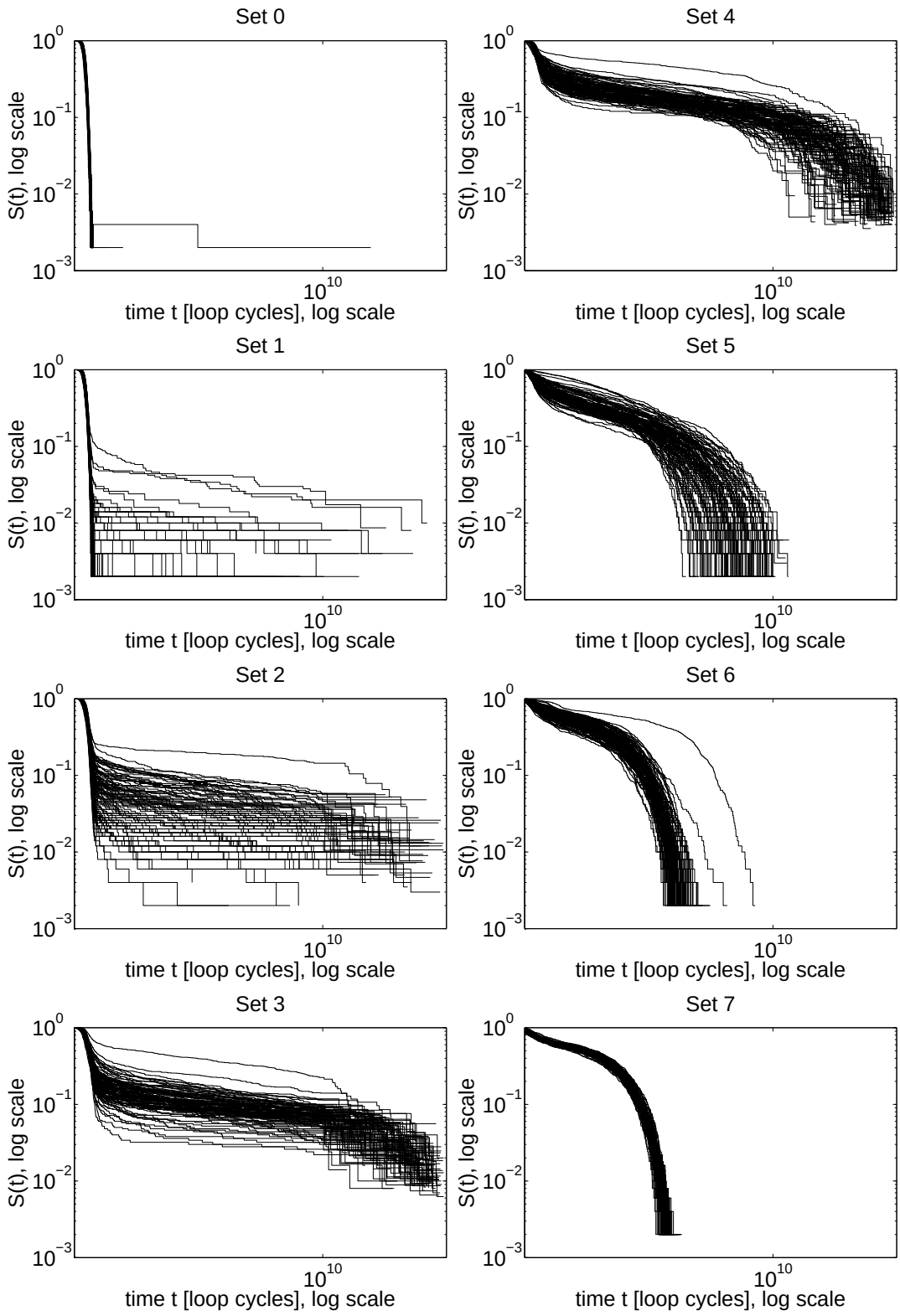


Figure 8.1. Log-log plots of the survival function of `Satz-Rand` on each instance of the morphed graph-coloring benchmark, divided in sets with different amounts of structure. Heavy tails are clearly visible for sets 1 to 5, for which a portion of the survival function can be approximated with a straight line, indicating a less than exponential decrease. Plots for set 9 are nearly identical to those for set 8, and are therefore omitted.

8.2 Impact of censored sampling

Formula (2.12) from [Luby et al., 1993] allows to evaluate the optimal restart strategy for an algorithm, based on its RTD on the instance at hand (Sec. 2.4.2). In practice, this function is not available, and has to be estimated. Such estimate may be carried out collecting a censored runtime sample on a set of instances of similar difficulty, on which the algorithm displays a similar RTD. The performance of the corresponding sub-optimal uniform strategy $\hat{\tau}$, evaluated minimizing (2.12) with an estimated $\hat{F}(t)$ in place of the real $F(t)$, will depend on the precision of the estimated $\hat{F}(t)$, which will in turn depend on the number of samples used to estimate it, and the amount of censoring.

More precisely, there are two sources of sub-optimality at play here: the use of a set of instances to collect the runtime sample, instead of a single one; and the fraction of observations which are censored. Let us focus on the latter: If we fix the number of samples, and vary the fraction of censored observations, we expect to observe a trade-off between the time spent running the sampling experiments, from whose outcomes $\hat{F}(t)$ is estimated, and the performance of the corresponding strategy $\hat{\tau}$. It is precisely this trade-off that we intend to analyze here.

In order to do so, we set up a simple learning scheme. Given a set of instances, and a randomized solver a, we first randomly pick a subset of n instances. For each instance, we start r runs of the algorithm a, differing only in the random seed, for a total of $k = nr$ parallel runs. We control the duration of these "training" experiments with Type II censored sampling (see Sect 3.2), fixing a censoring fraction $c \in [0, 1)$ in advance: as the first $\lfloor (1-c)k \rfloor$ runs terminate, we stop also the remaining $\lceil ck \rceil$. In other words, we use a parallel portfolio to collect the sample, as discussed in Section 6.2.

The gathered runtime sample is then used to train a model $\hat{F}(t)$ of the CDF, from which a uniform strategy $\hat{\tau}$ is evaluated, by minimizing (2.12) numerically. The performance of $\hat{\tau}$ is then tested on the remaining instances of the set. Varying c, we can measure the corresponding variations in the training time, and in the performance of $\hat{\tau}$ on the test set.

The experiments were conducted using Satz-Rand on the graph coloring benchmark described in the previous section, composed of 9 sets on which the heavy-tailed behavior of Satz-Rand varies. For each set, we repeated the simple scheme described above, using pre-collected runtimes to simulate the execution of $r = 20$ copies of Satz-Rand on each of $n = 50$ randomly picked training instances, estimating a model of the RTD of the set based on the resulting sample, and testing the corresponding optimal strategy $\hat{\tau}$ on the remaining 50 instances. The process was repeated for 10 different levels of the censored fraction c during training, from $c = 0$ to $c = 0.9$.

As for the model, we tried different alternatives, including Weibull[1] (2.9), lognormal (2.10), double Pareto lognormal (3.12) and right-hand Pareto lognormal (3.13). We also tested various mixtures of pairs of these distributions. The models were trained by maximum likelihood, as described in Section 3.3.1: to avoid numerical problems due to the large runtime values, we divided all times by 10^6. To compare with a non-parametric approach, we repeated the experiments using the Kaplan-Meier estimator (3.16). Among the parametric models, we obtained the best results with a mixture including one lognormal (2.10), and either one double-Pareto lognormal, or only the heavy-tailed component, the Double Pareto distribution (3.13). Here we describe results obtained using a parametric model, a mixture of a lognormal (2.10) and a double Pareto (3.13) distribution, labeled `logndp`; and for a simple non-parametric model,

[1] Interestingly, the Weibull distribution, reported by Frost et al. [1997] as having a good fit on satisfiable instances near the sat-unsat phase transition, had instead a very poor fit in this case, with instances in the underconstrained region.

the Kaplan-Meier estimator (3.16), labeled `kme`. All quantities reported are upper 95% confidence bounds obtained from results of 10 repetitions of the experiment, with a different random choice of the training instances, and different seeds. In the right column of Figures 8.2 to 8.4, we present the trade-off between training cost, labeled `train`, and restart performances on the test set, for the two models, respectively labeled `logndp` and `kme`, at different values of the censoring fraction c. We also plot the cost of the universal strategy, labeled U, on the test set (the performance on the training set is similar, as both are composed of 50 randomly picked problem instances). For the test time, we can appreciate some degradation of performance only for very heavy censoring ($c = 0.8, 0.9$), for which the advantage in training time is negligible anyway. Note that this does not mean that the accuracy of the model is unaffected: to highlight this apparent contradiction, we also plotted, in the left column, the value of a χ^2 statistic (Sec. 3.3.1) of the parametric model `logndp`. This statistic was measured as suggested by Frost et al. [1997], dividing the uncensored data into m bins (on a logarithmic scale), and comparing the number of samples o_i in each bin to the one e_i predicted by the fitted distribution, according to (3.7). A high value indicates a poor fit: the model passes the test with confidence α if χ^2 is lower than the $1 - \alpha$ quantile of the χ^2 distribution with $m - k$ degrees of freedom, k being the number of parameters in the model. In these plots, white bars indicate the 95% acceptance threshold. While the χ^2 statistic is near or below the threshold for the uncensored estimate, its value increases rapidly even for low values of c, exposing the degradation of the model.

In type II censoring, the number of censored observations is fixed, and the censoring threshold is a random variable (Sec. 3.2). In Figures 8.5 to 8.8, each column corresponds to a different set. The top row reports the average of the censoring threshold t_c, resulting from different censoring fractions c. This, along with the training cost, allows to appreciate the tail behavior of Satz-Rand on the different problem sets. On problem set 1, most runtimes have a similar value, and the remaining few are very large. t_c is greatly reduced by a modest $c = 0.1$, but further censoring does not decrease it much: the same obviously applies to training cost. On problem set 8, runtime values are spread along two orders of magnitude. Increasing c has a more gradual impact on t_c, and on training cost. This situation varies continuously for intermediate problem sets. Problem set 0 is less interesting, as all runs of Satz-Rand end in a similar time, and heavy tailed behavior is not observed. The resulting plots are similar to problem set 1, without the heavy tail effects.

In the second row, we display the log-log plot of the survival function (2.4) estimated by `logndp`, on a single run, for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with a better approximation of the real $S(t)$ of the set, the empirical Kaplan-Meier (3.16) evaluated on 200 runs for each instance (labeled `real`). One can visually appreciate the degradation of the model, induced by censored sampling, especially for values of t larger than t_c, which are not seen by the model. So why is the performance of the restart strategy not affected?

The third row of Figures 8.5 to 8.8 seems to suggest an answer. It plots the expected cost (2.12) of a uniform restart strategy, against the restart threshold τ, evaluated based on `logndp` estimates at different levels of censoring. The comparison term `real` is the *actual* performance of a restart strategy τ, evaluated *a posteriori* on the same run: averaging this on multiple runs, one would obtain an estimate of the real $E\{t_\tau\}$ for the instance set. Note that this may in general differ from (2.12) evaluated from the RTD of the set. We can see that the estimated and real cost differ greatly, but have a similar minimum: this allows $\hat{\tau}$ to be efficient also with a poor $\hat{F}(t)$, obtained from a heavily censored runtime sample.

Figures 8.9, 8.10 plot again the tail of $\hat{S}(t)$ and the cost of restarts (2.12) obtained with the Kaplan-Meier estimator (3.16). This simple model proved similar in performance to `logndp`,

also in the few cases where this latter failed to converge due to numerical issues, namely the near-singularity of a matrix. (see again Figures 8.2 to 8.4).

The performance of the universal restart strategy U on the test set is consistently worse. Its advantage on the training set was predictable, as in our simple scheme 20 copies of Satz-Rand are run in parallel on each training instance. Such advantage obviously decreases with c: on sets 7, 8, training cost is actually lower for $c = 0.8, 0.9$.

There is only an apparent contradiction between the rapid degradation of the model, following the increase in censored data, and the stability of the performance of the estimated optimal restart strategy. Traditional statistical tests are in fact intended to measure the fit of a pdf along the whole spectrum of possible values. The formula for the restart performance (2.12) is instead based on the CDF (2.3), which is the integral of the pdf; and on its further integration up to τ, which is usually small. This means that the actual shape of a large portion of the distribution is irrelevant, as long as its mass does not vary; while for values lower than the restart threshold τ, the integration involved in the CDF acts as a "denoising" filter, making (2.12) more robust to a loss of fit of the model.

These experiments suggest that the cost of learning RTD models can be greatly reduced by censoring, and that a rough model already allows to evaluate a near-optimal strategy. These results encouraged us to devise an online method for learning restart strategies, GAMBLER (Sec. 7.2), whose performance is reported in the next section.

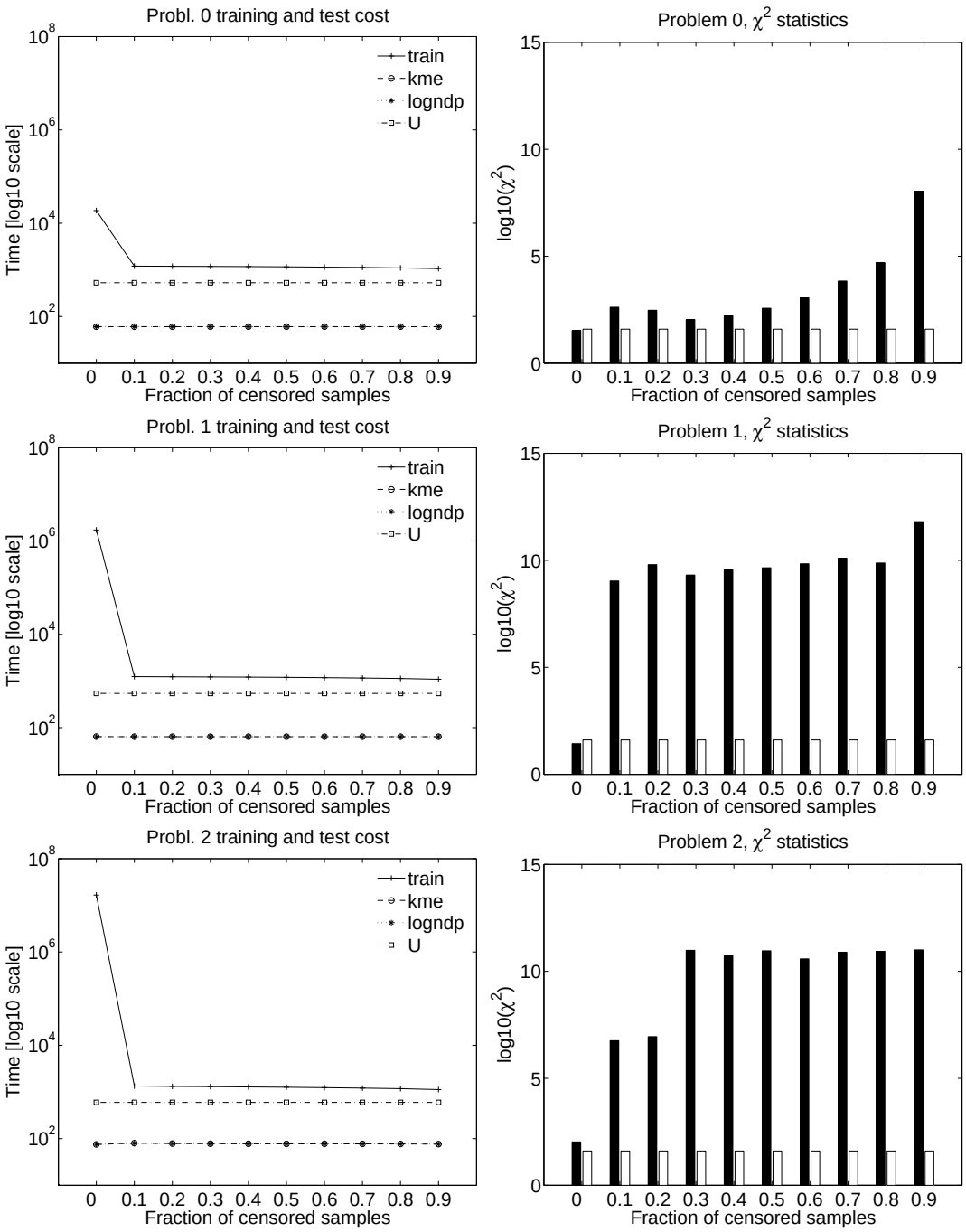


Figure 8.2. Problem sets 0 to 2. Left: the trade-off between training cost (`train`) and test performances of the parametric mixture lognormal-double Pareto (`logndp`), and the non-parametric Kaplan-Meier estimator (`kme`), for different censoring fractions c. The latter two are practically the same, so the corresponding lines are superimposed. `U` labels the performance t_U of the universal strategy on the test set. Right: $\log_{10}$ of the χ^2 statistics for `logndp` (black), compared to $\log_{10}$ of the acceptance threshold (white).

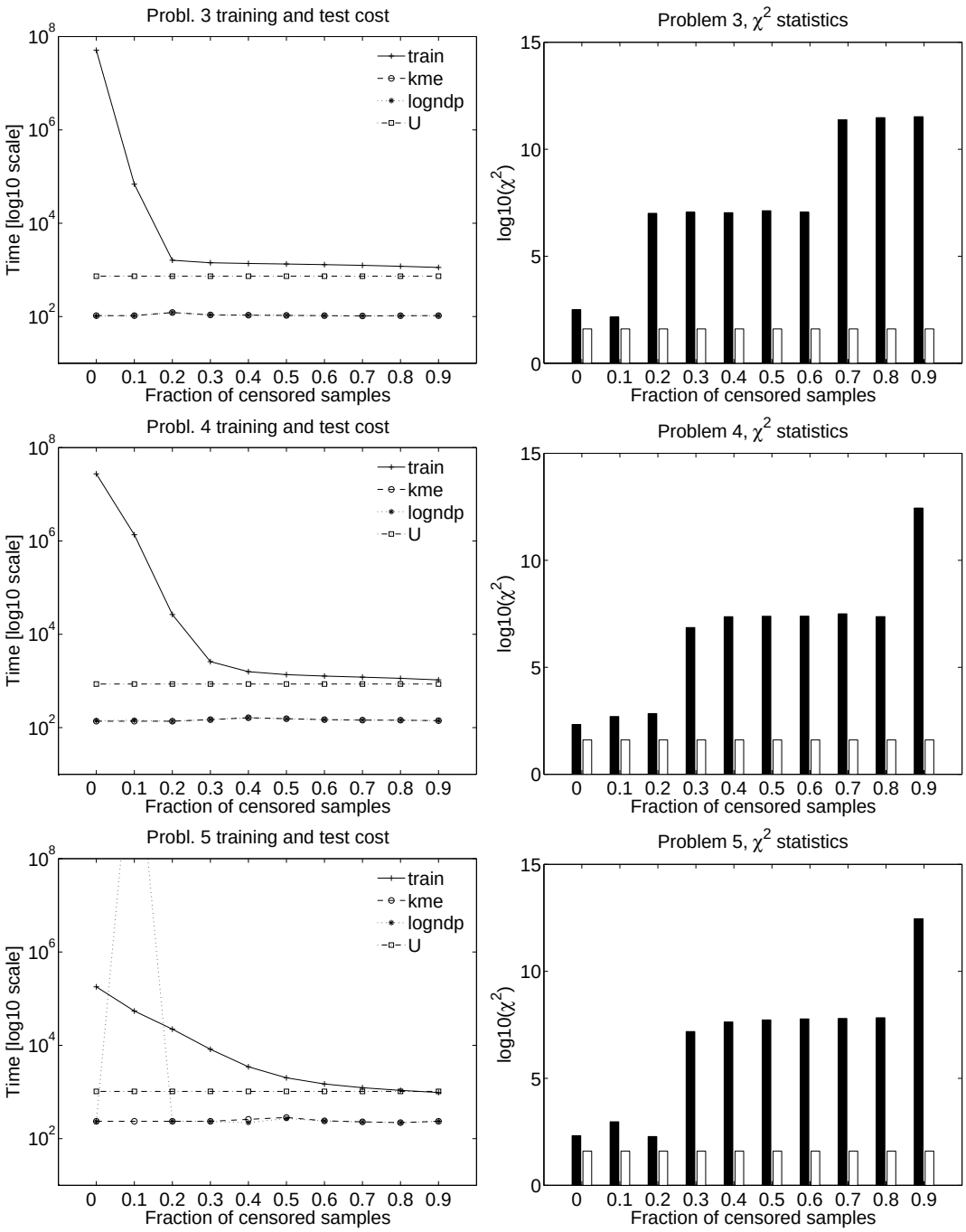


Figure 8.3. Problem sets 3 to 5. Left: the trade-off between training cost (`train`) and test performances of the parametric mixture lognormal-double Pareto (`logndp`), and the non-parametric Kaplan-Meier estimator (`kme`), for different censoring fractions c. The latter two are practically the same, so the corresponding lines are superimposed, except for set 5, $c = 0.1$, where parametric estimation did not converge at all runs, due to numerical issues. `U` labels the performance t_U of the universal strategy on the test set. Right: $\log_{10}$ of the χ^2 statistics for `logndp` (black), compared to $\log_{10}$ of the acceptance threshold (white).

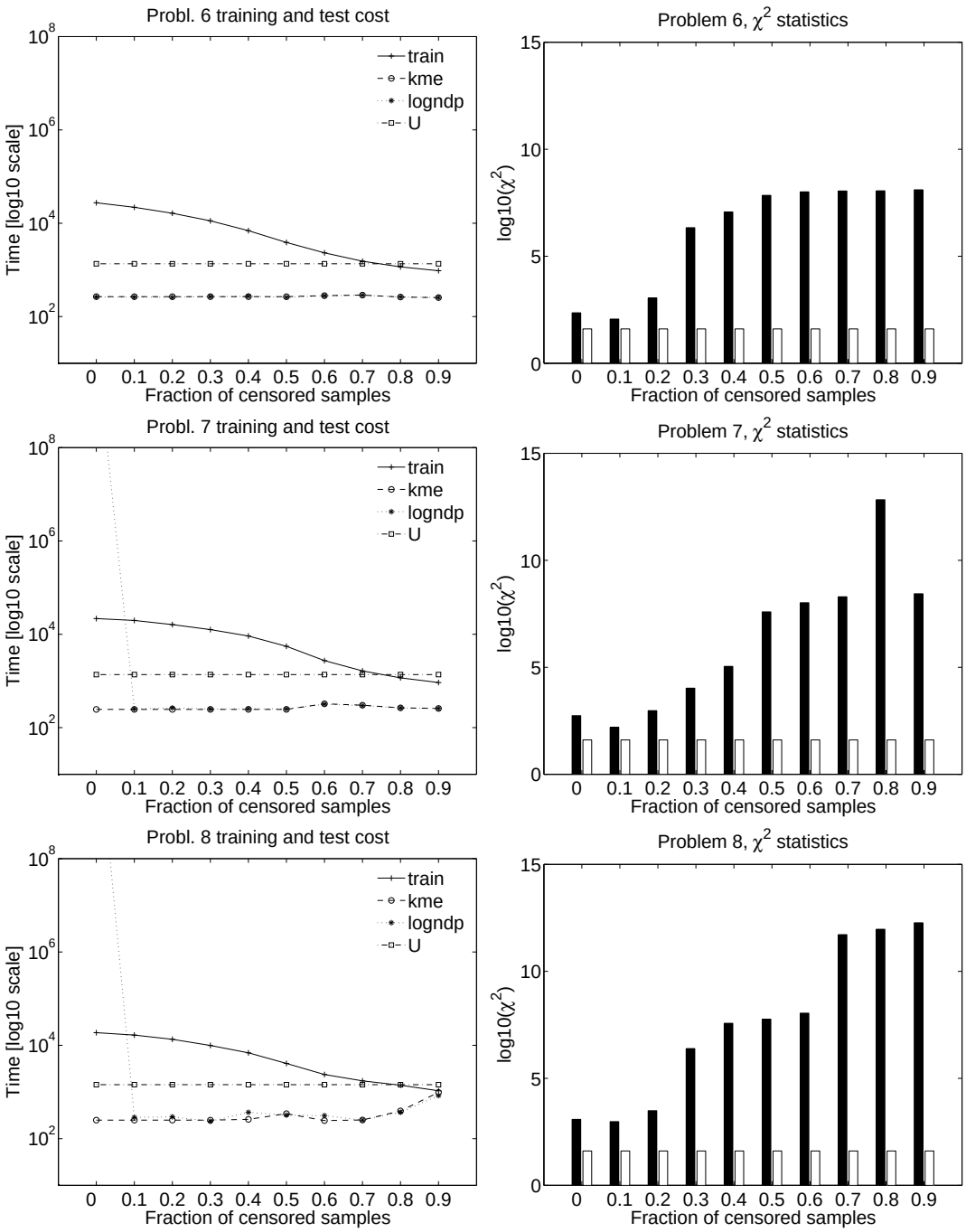


Figure 8.4. Problem sets 6 to 8. Left: the trade-off between training cost (`train`) and test performances of the parametric mixture lognormal-double Pareto (`logndp`), and the non-parametric Kaplan-Meier estimator (`kme`), for different censoring fractions c. The latter two are practically the same, so the corresponding lines are superimposed, except for sets 7, 8, $c = 0$, where parametric estimation did not converge at all runs, due to numerical issues. `U` labels the performance t_U of the universal strategy on the test set. Right: $\log_{10}$ of the χ^2 statistics for `logndp` (black), compared to $\log_{10}$ of the acceptance threshold (white).

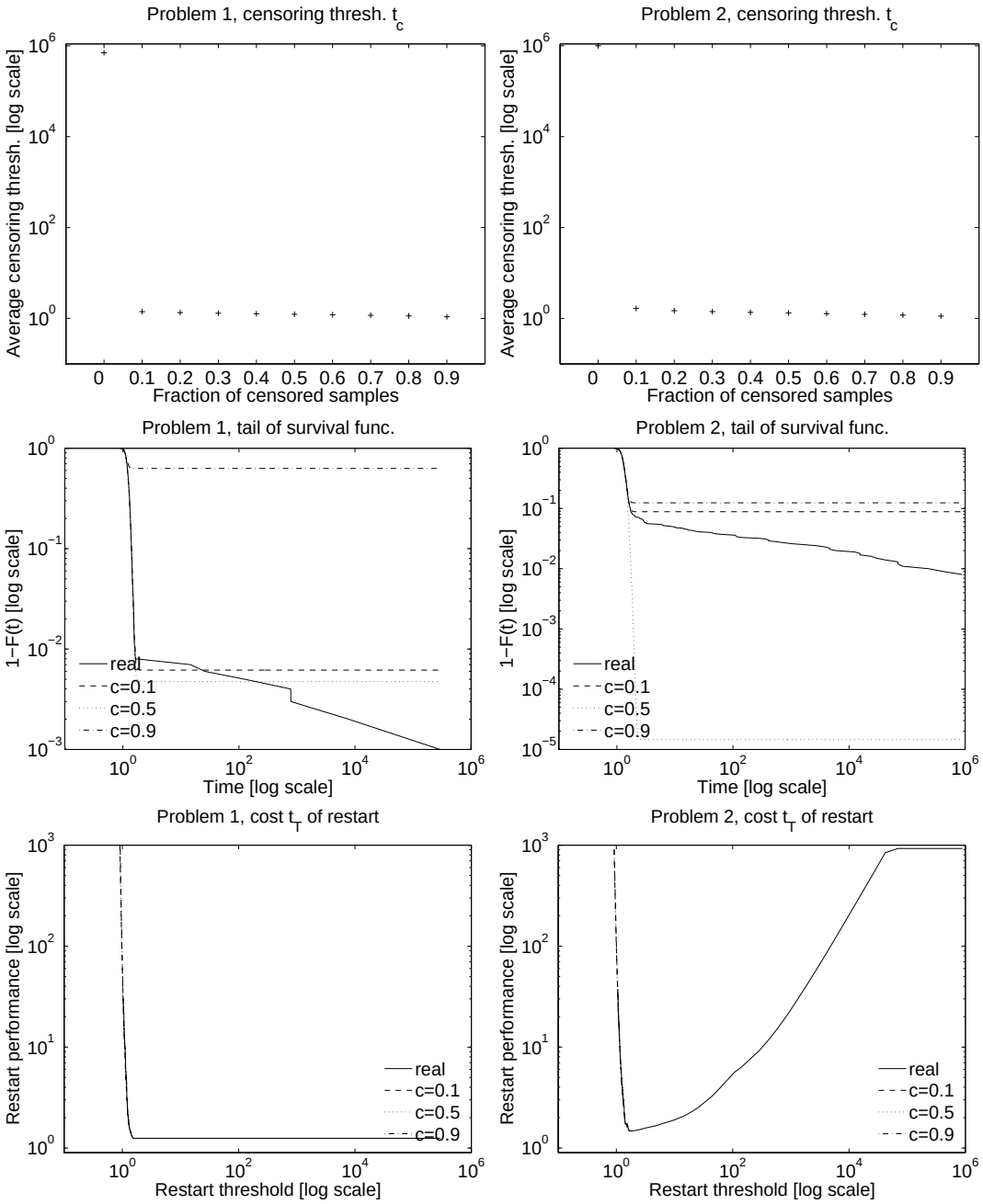


Figure 8.5. Problem sets 1 (left column) and 2 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; tail of the survival function estimated with `logndp`; estimated expected cost of restart (2.12) for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last two rows refer to a single run. The lines are practically superimposed.

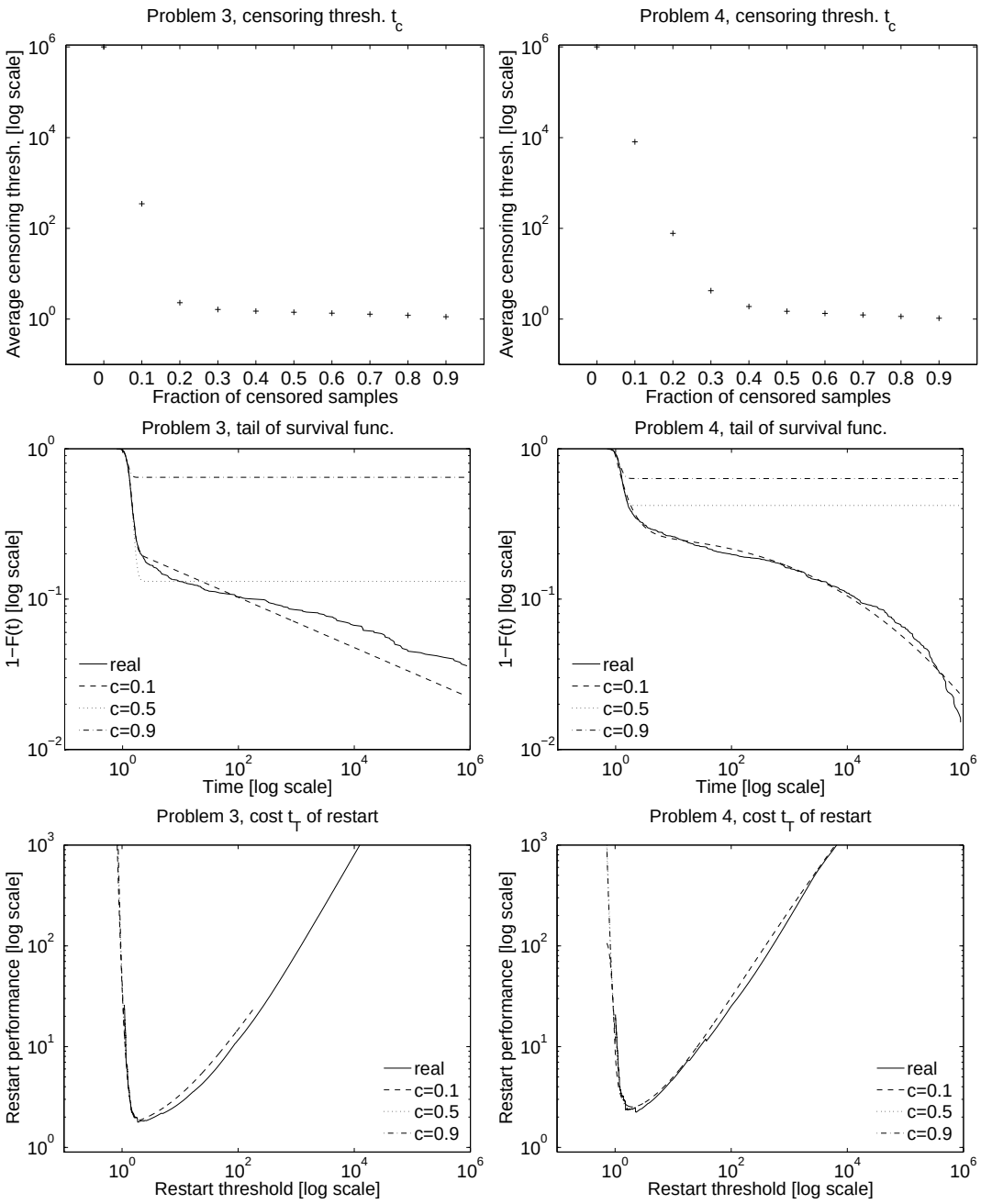


Figure 8.6. Problem sets 3 (left column) and 4 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; tail of the survival function estimated with `logndp`; estimated expected cost of restart (2.12) for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last two rows refer to a single run. Different levels of censoring correspond to a nearly identical estimate, which is very similar to the real cost.

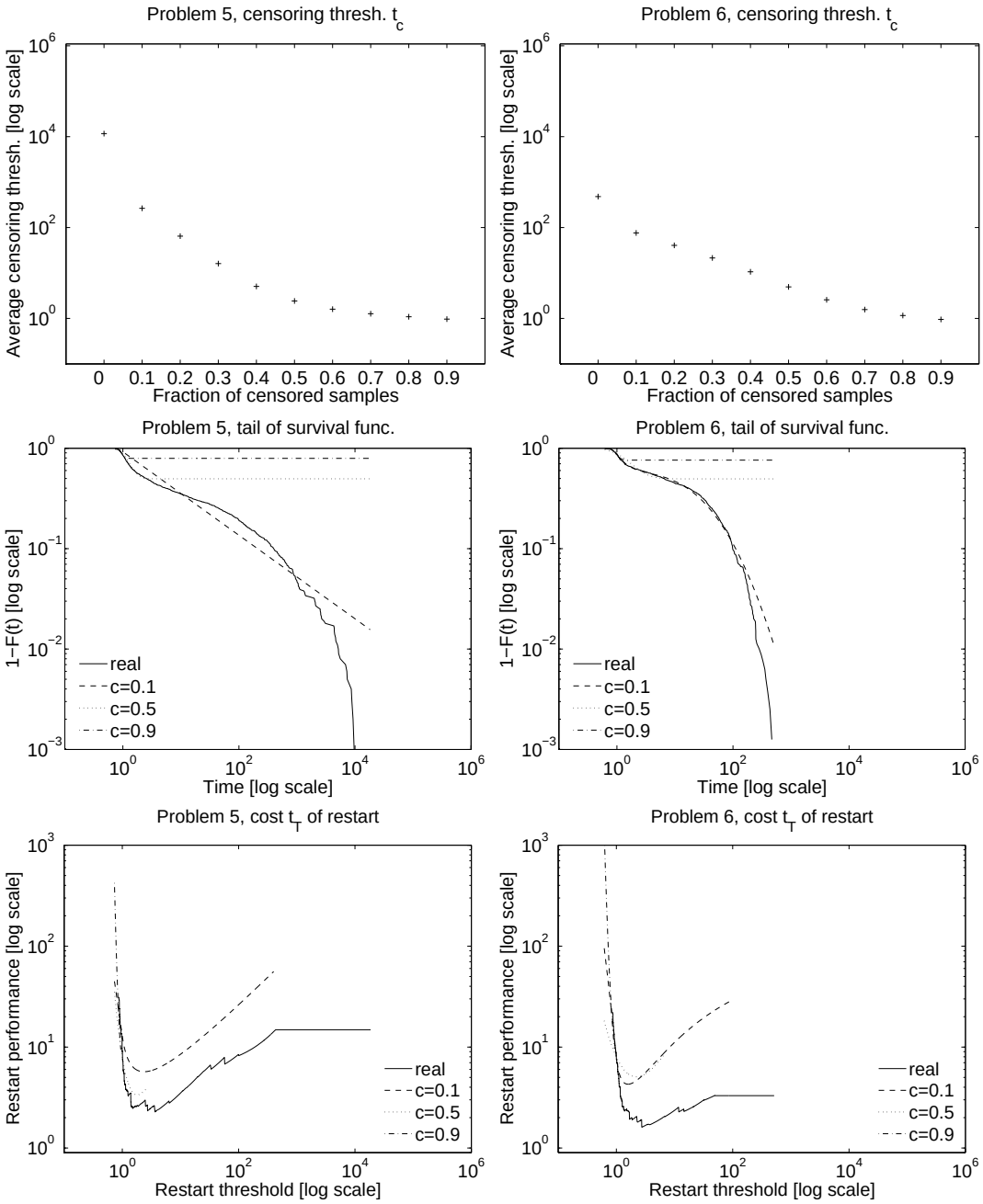


Figure 8.7. Problem sets 5 (left column) and 6 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; tail of the survival function estimated with `logndp`; estimated expected cost of restart (2.12) for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last two rows refer to a single run. Note the similar minima in last row.

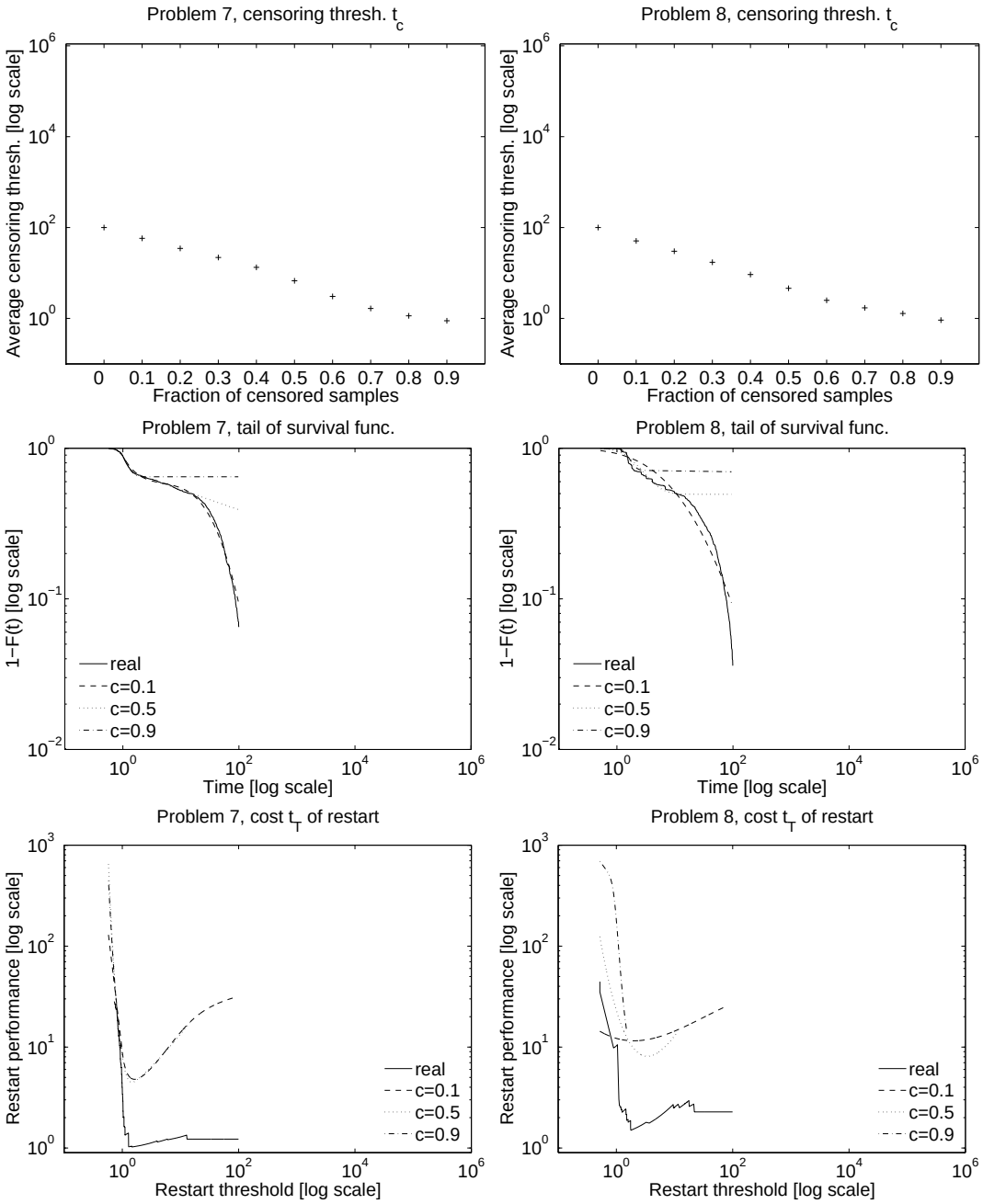


Figure 8.8. Problem sets 7 (left column) and 8 (right column). From top to bottom: average censoring threshold t_c for different fractions of censoring; tail of the survival function estimated with `logndp`; estimated expected cost of restart (2.12) for different censoring levels ($c = 0.1$, $c = 0.5$, $c = 0.9$), compared with real cost, evaluated *a posteriori*. Last two rows refer to a single run. Note the similar minima in last row.

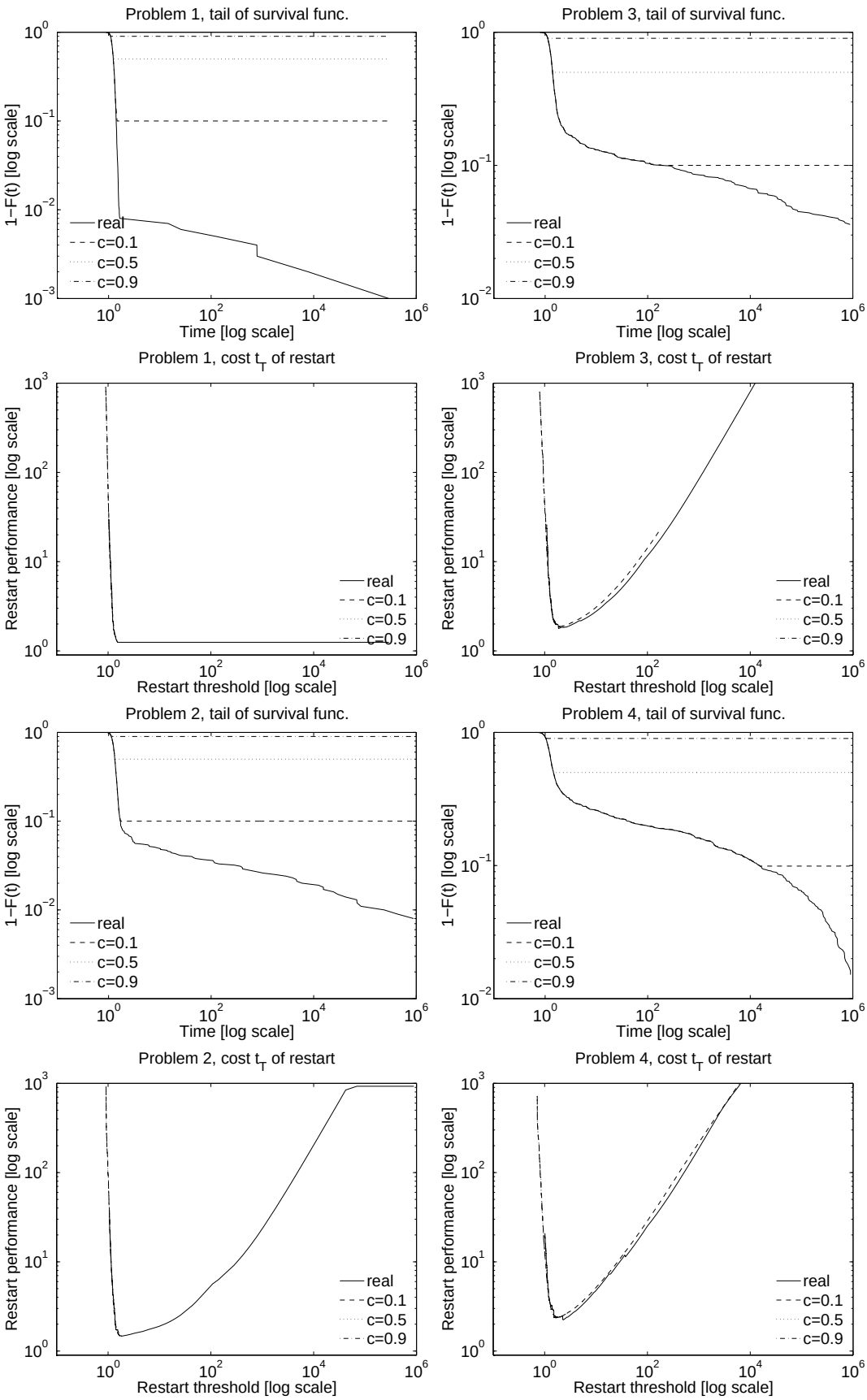


Figure 8.9. Problem sets 1 to 4, results for `kme`. See Figure 8.5 for details.

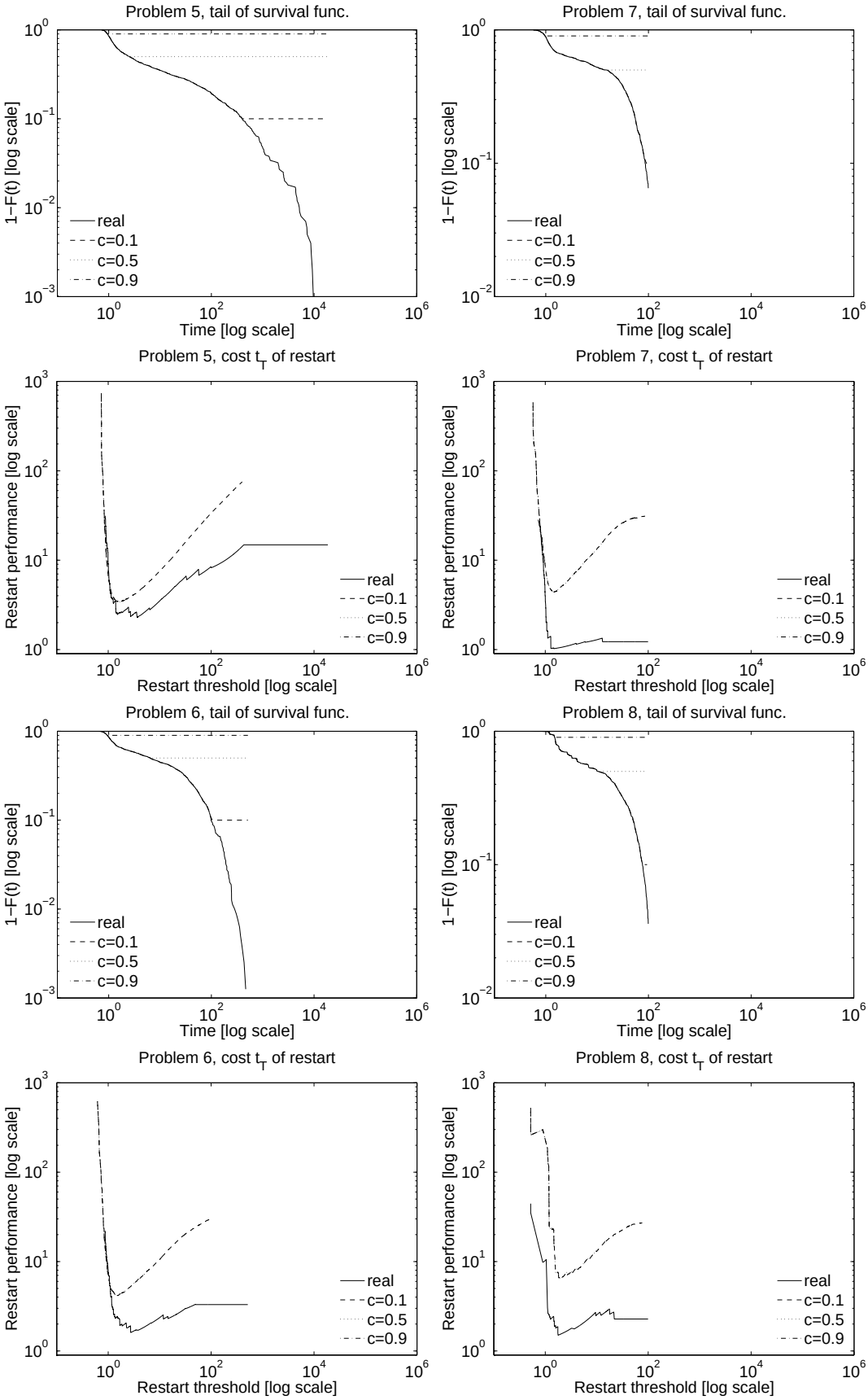


Figure 8.10. Problem sets 5 to 8, results for `kme`. See Figure 8.5 for details.

8.3 Experiments with GambleR

In this section we present experiments with GAMBLER, the online scheme for learning restart strategies presented in Section 7.2. Also these experiments were simulated using runtimes collected for Satz-Rand on the graph coloring benchmark (see Sect 8.1). As bandit problem solver, ϵ-GREEDY was used, as EXP3LIGHT-A does not feature a minimum exploration probability. As a model, we used the non-parametric product limit estimator (3.16), given its competitive results in the previous set of experiments, and its lower computational cost.

Each experiment was repeated 25 times with different random seeds, and a different random order of the instances. For comparison purposes, we repeated the experiments running the original algorithm, without restarts (labeled *SR*) and the universal strategy alone (U). To compare with the ideal performance of τ^*, we evaluated, *a posteriori* for each run, the τ that minimized the cost of the instance set ($\tau_L(set)$), and the cost of each instance ($\tau_L(inst)$), based on the *actual* run-time outcomes. Note that these can be different for each run, and their performances are *lower bounds* on the performances of the optimal τ^*, evaluated from the unknown actual RTD of, respectively, the instance set, and each problem instance.[2] The difference of these two bounds is particularly interesting, as it is an indirect measure of the heterogeneity of the RTD of the instances in each set. To show the effect of a more heterogeneous instance set, we also run experiments with all the 900 instances grouped as a single set, again randomly mixed. In Figures 8.11, 8.12 we present, for each set, the total computation time for GAMBLER and the comparison terms, on the 25 runs, using a box-plot representation[3].

The results are quite impressive, and further confirm that the estimated restart strategy $\hat{\tau}$ is not very sensitive to the fit of the model $\hat{F}$. In a typical run, between 40% and 80% of the sample is censored, as there is only one uncensored run-time for each instance; the fit of the model is visibly bad, and it is limited to the lower portion of the time scale, near or below $\hat{\tau}$, but $\hat{\tau}$ itself quickly converges close enough to τ^*, giving a near-optimal strategy.

Instances in 0 are easy. `Satz-Rand` solves all instances in a similar time, any larger τ is an optimal restart value, i.e., restarts are never executed, and the performance of a single copy of `Satz-Rand` (SR) is the same as the ones of the optimal restarts. Also our GAMBLER quickly learns that, while U has to reach this value for every instance, resulting in an eight times worst median performance. In problems 1 to 5 we see the effect of a heavy-tailed RTD. Only a few "unlucky" runs have very long completion times, but this is enough to penalize the performance of `Satz-Rand` dramatically. GAMBLER scores fairly against the lower bounds, and U is between 3 and 5 times worst. From problem 6 on, we see that the heavy tail effect is less marked, but the two lower bounds diverge noticeably: instances in these sets present more heterogeneous RTDs, and the instance-optimal $\tau^*(inst)$ may vary of more than one order of magnitude. Here U is about 2.5 times worse than GAMBLER. The worst performance of GAMBLER compared to $L^*(set)$ can be seen on problem 8, where t_G is about 2.6 times $t_{L^*(set)}$. Here the optimal threshold $\tau^*(inst)$ is small for most instances, and about ten times larger for a few ones. In this situation, also $\tau^*(set)$ varies visibly among different runs, and $\hat{\tau}$ sometimes overestimates the optimal threshold. Results on the whole set of instances are better than expected: $t_G/t_{L^*(set)}$ is about 1.23, but the performance compared to $t_{L^*(inst)}$ is obviously worst (1.78). This is natural, as both

[2] As $E(t_{\tau^*}) = \min_\tau\{E(t_\tau)\} \geq E(\min_\tau\{t_\tau\}) = E(t_L)$ in both cases.

[3] 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.

GAMBLER and $\tau^*(set)$ cannot discriminate different restarts for each instance. U completes the set with a median performance about 4.2 times worse than GAMBLER.

8.4 Discussion

The results reported in Section 8.2 suggest that a rough RTD model already allows to evaluate a near-optimal strategy. Therefore the cost of runtime sampling can be greatly reduced by censoring, with a negligible impact on restart performance. From a practical point of view, these experiments show that even a sub-optimal restart strategy can have a relevant advantage over the universal. This advantage can be obtained for an additional training effort, which can be greatly reduced by censored sampling. On a larger test set, this initial training effort would pay off quite rapidly. Note also that there is no reason to stop the training process, as it is relatively cheap compared to problem solving, and each restart can also be interpreted as a censored sample. These considerations motivated our effort in devising an online learning method, which resulted in GAMBLER (Sec. 7.2).

Regarding the RTD models used, parametric estimation was found to be computationally heavy, and did not always converge, due to numerical issues. The simple non-parametric Kaplan-Meier estimator (3.16) obtained a consistently good performance, with a much lower computational complexity: updating the model requires insertions in a sorted list, and the cost of estimation is linear in the size of the sample. Later experiments were therefore limited to non-parametric methods.

The experiments with GAMBLER (Sec. 7.2) met our expectations: its performance was consistently better than the universal strategy alone, and often competitive with the best per set strategy. It could save a few orders of magnitude in computation time for instance sets where `Satz-Rand` is heavy-tailed, and, unlike the universal strategy, it did not worsen the performance otherwise. Note that the method can be made per instance simply modeling the instance RTD with a regression model (Sec. 3.4). We expect analogous results with any heavy-tailed randomized algorithm.

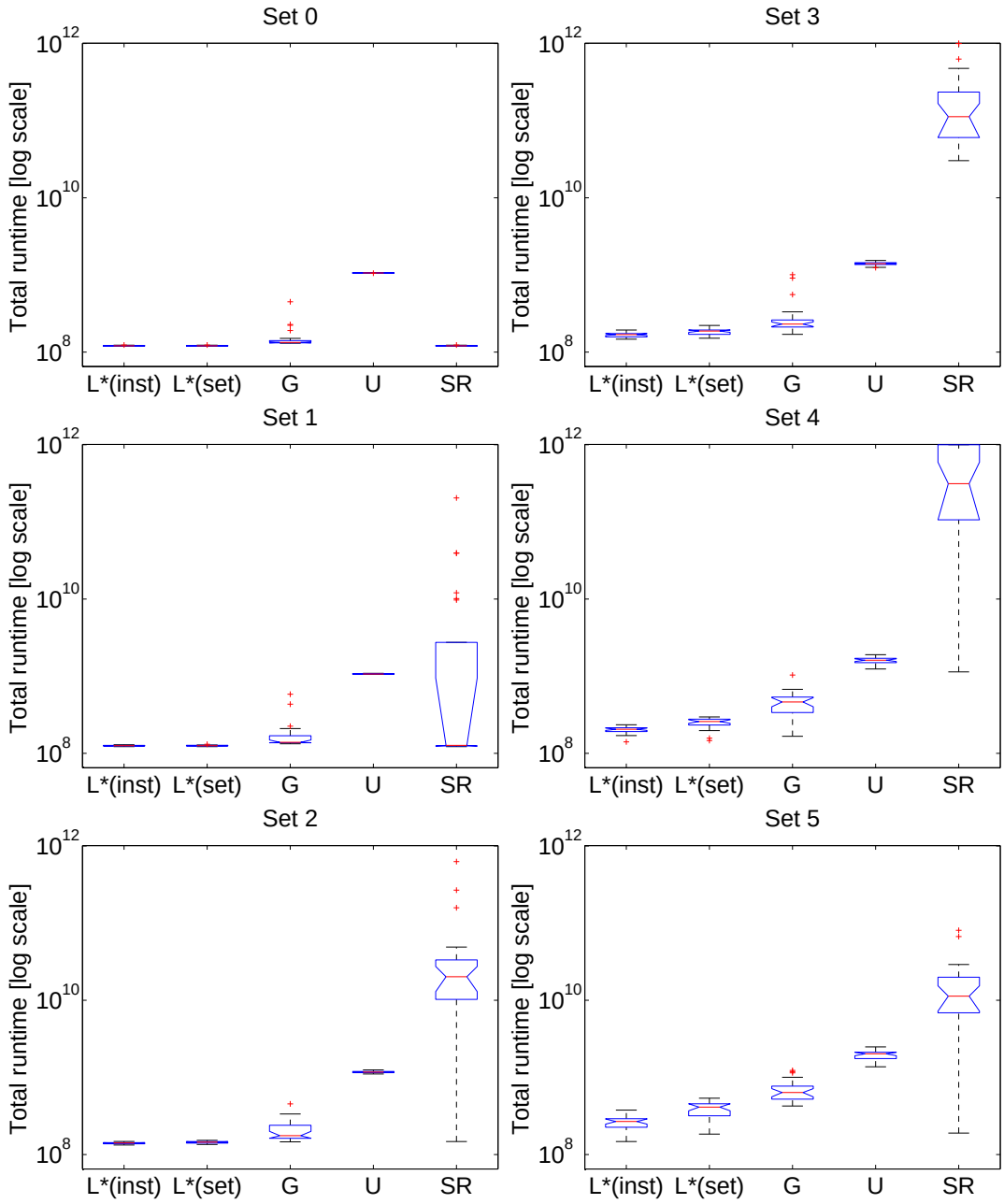


Figure 8.11. Problem sets 0 to 6. Results of 25 runs. Time to solve each set of instances using GAMBLER (G), compared with the universal strategy alone (U), and `Satz-Rand` without restarts (SR, lower bound). $L^*(set)$ is a lower bound on the performance of the unknown optimal restart strategy for the whole set. $L^*(inst)$ is instead a lower bound on the performance of a distinct optimal restart strategy for each instance.

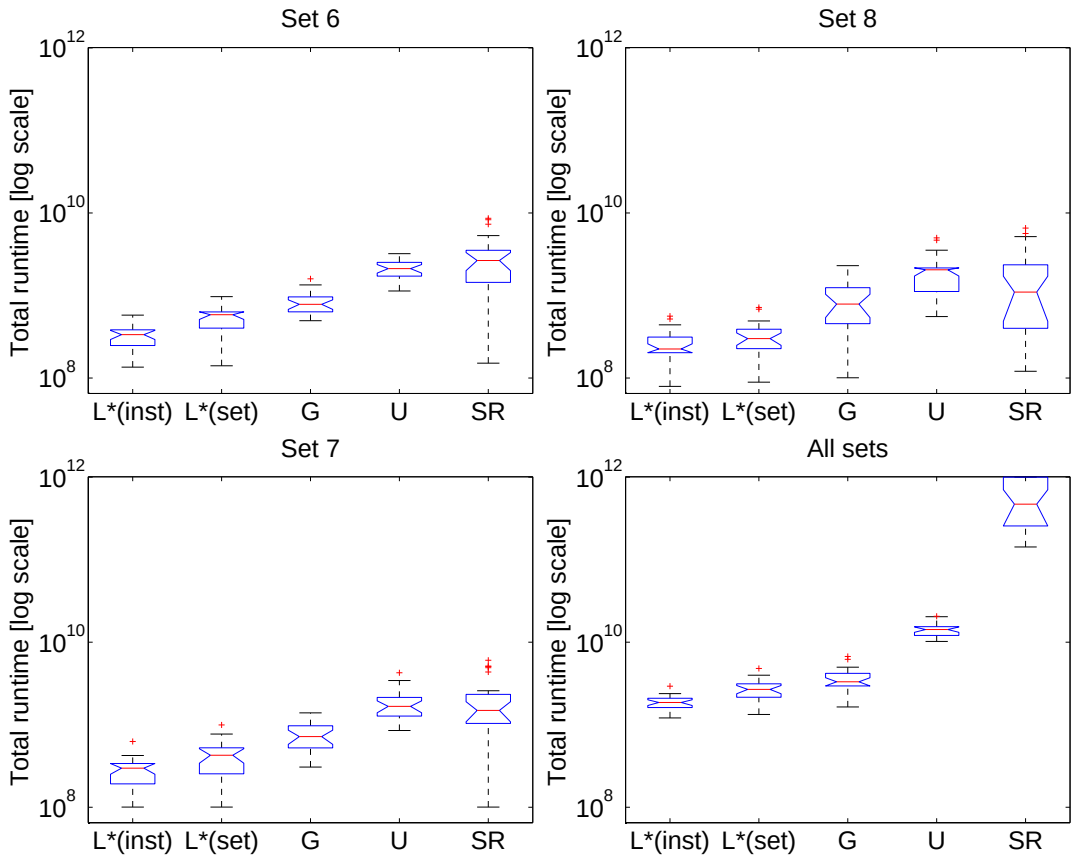


Figure 8.12. Problem sets 7 to 9, and all sets together. Results of 25 runs. Time to solve each set of instances using GAMBLER (G), compared with the universal strategy alone (U), and `Satz-Rand` without restarts (SR, lower bound). $L^*(set)$ is a lower bound on the performance of the unknown optimal restart strategy for the whole set. $L^*(inst)$ is instead a lower bound on the performance of a distinct optimal restart strategy for each instance.

Chapter 9

Experiments with Algorithm Portfolios

In this chapter we analyze the performance of GAMBLETA (Ch. 7) on several time allocation benchmarks, comparing it with other methods where possible, and validating design decisions with several deletion experiments.

GAMBLETA is a framework for time allocation, rather than an actual allocator: in Section 9.1 we present the specific version used here, describing the BPS, the set of time allocators, and the RTD models. The same settings will be used for all experiments. We will also present results in a uniform manner, using the same plots, as described in Section 9.2.

Each benchmark consists of a set of problem instances, and a set of algorithms. We first present results from several solver competitions, comparing our performance with that of the offline and online allocators of Streeter [2007] (Sec. 9.3). In Section 9.4 we compare with results of a static algorithm selection approach [Leyton-Brown et al., 2002], on a large set of instances of the Auction Winner Determination problem. In Section 9.5 we revisit the SAT/UNSAT scenario introduced in Section 6.1.1, where two SAT solvers, one complete and one incomplete, are used to solve a large set of SAT and UNSAT instances. In Section 9.6 we present experiments with multiple processors, on the SAT/UNSAT benchmark, and on the morphed graph coloring problems of Section 8.1. The following two sections investigate the impact of several design decisions (Sec. 9.7), and the performance of the BPS (Sec. 9.8). Section 9.9 concludes discussing results.

9.1 Settings

This section describes the details of the implementation of GAMBLETA used in the following. All experiments were performed in Matlab, simulating the actual execution of the algorithms based on runtime data.

As BPS, we adopted EXP3LIGHT-A (Sec. 7.5). All the allocators described in Chapter 5 are used, in their dynamic version (Alg. 5). More precisely, the set of allocators $\mathcal{T}$ consists of

1. The uniform allocator $\text{TA}_{\mathcal{U}}$.

2. The expected time allocator TA_{Et}.

3. A quantile allocator TA_Q, with $\alpha = 0.25$.

4. A "greedy" contract allocator TA_C, with a dynamic $t_c = \tau_k$

5. The greedy task switching allocator (5.8) of Streeter and Smith [2008], labeled TA_{Gr}.

The reason for choosing this set is that it allows to test the performance of each method discussed in Chapter 5. 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. All other 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 (see below), this is guaranteed to happen at some point. Therefore, each TA will eventually allocate a portion of time to each algorithm, satisfying hypothesis 1 (Sec. 5.1).

For TA_{Et}, $\text{TA}_Q$and TA_C, the RTD of the portfolio was evaluated as in (6.1), based on the survival functions of each algorithm, as estimated by the corresponding models. The share was optimized numerically[1]. The allocation was updated dynamically, as in Alg. 5, using an exponentially spaced sequence of time intervals $\tau_i = 2^i \tau_0$, with $\tau_0 = 1$ second.

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 $1 - F(\infty) \leq \epsilon$, and allocating uniformly otherwise. We arbitrarily set $\epsilon = 0.01$.

The quantile parameter α of $\text{TA}_Q$was also chosen arbitrarily, based on the observation that high quantiles produce allocations similar to TA_{Et}(see 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 (5.8). 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. Another reason for including this allocator is to highlight the fact that GAMBLETA can usefully exploit any existing TA, simply adding it as an additional "arm" of the bandit.

The allocators described in Chapter 5 are based on estimates of the instance RTDs of the algorithms, which can be obtained using regression models, conditioned on instance features (Sec. 3.4). After some unconvincing experiences with parametric and semi-parametric methods, we implemented the non-parametric hazard estimator of Wichert and Wilke [2005] (see (3.20), Sec. 3.4). The method requires to specify a kernel function, and a bandwidth b_n depending on the size of the sample n, with $b_n \in [n^{-1/2}, n^{-1/4}]$. We present results for the uniform kernel (0.5 on $[-1, 1]$, and 0 elsewhere), and $b_n = n^{-1/4}$, which provide the widest allowed kernel. This decision was validated with several experiments, with different kernels and bandwidths, which are not reported here as the impact on performance was usually negligible.

[1] Using the Matlab function `fmincon`

A separate model was learned for each algorithm: when these were available, we used a small set of problem specific features as covariates[2]. To update the estimate dynamically, the time spent y_k was used, as in (3.22), discarding hazard values h_j with $t_j \leq y_k$. The hazard (3.20) was used to perform a product-limit estimate (3.16) of the survival function.

A decisive advantage of this method is that it measures similarity among covariates based on their distribution, and not on their actual value: therefore, it is not sensitive to scaling, it does not require knowing the range of the covariates $\mathbf{x}$ in advance, nor it does pose the problem of balancing the impact of different dimensions of the covariate, which is typical of parametric regression models. Another interesting feature of this model is its computational simplicity[3].

9.2 Reporting and plotting results

Each experiment described in this chapter consists in the solution of a sequence of problem instances. The parallel execution of the algorithms was simulated, using stored runtime data. Time values reported only include the algorithm runtimes, as our implementation is far from being efficient.

All experiments were repeated 20 times, each time with a different random reordering of the problem instances, and a different random seed for the BPS of GAMBLETA, as well as for the algorithms, if randomized. This was only possible for benchmarks on which we personally collected algorithm runtimes: in most cases, the runtimes were obtained online, and were only available for a single run. In such cases, the results could differ if the same experiments were performed after collecting the runtimes on several runs, but we do not expect such differences to be relevant.

Most experiments were performed based on the results of solver competitions, where the runtime of each algorithm on each instance was limited to a maximum "timeout" value. In these cases, we report the number of instances solved before timeout, and the total runtime, discarding instances which none of the algorithms could solve. In the remaining experiments, we did not use any timeout, and we only report the total runtime, as all instances are eventually solved.

We will report the results of each run whenever it is possible to do so in a readable manner. When a single value is given, it will be a 95% confidence bound, evaluated on the 20 runs, based the Z distribution. Upper confidence bounds will be reported for quantities which we want to minimize, as the total runtime; lower bounds for those which should be maximized, as the number of solved instances.

We assess the performance of GAMBLETA by comparing it with other time allocators when possible. For all experiments, we also report the performances of the UNIFORM time allocator $\mathbf{s}_{\mathcal{U}} = (1/N, ..., 1/N)$ alone; and the one of an ORACLE, with foresight of the runtime values, which only executes, for each problem instance, the algorithm that will be fastest, achieving the best possible performance. We will also report the number of runs on which GAMBLETA is worse

[2] As the algorithms were not related. For different parametrization of the same algorithm, a single model can be used, conditioned also on parameter values.

[3] The learning phase consists in sorting independently the event times and the d dimensions of the covariates $\mathbf{x} \in \mathbb{R}^d$. The cost of prediction is d searches on the sorted covariate data, and the cost of (3.20). Quantiles can also be evaluated just by searching a value on a sorted list. We therefore expect that an optimized implementation would have a very low computational overhead. A simple optimization could consist in preserving the order when merging two hazard vectors. The fact that the data is sorted would allow for more advanced optimizations, based for example on balanced trees, with a cost $O(\log n)$ both for search and insertion, n being the number of samples.

than UNIFORM (WTU). Recall that, if $t_n(m)$ is the runtime of algorithm a_n on problem instance m, the runtime of ORACLE is $t_O(m) = \min_k\{t_n(m)\}$, while UNIFORM solves the instance in a time $t_{\mathcal{U}}(m) = N t_O(m)$. The performance $t(m)$ of an arbitrary allocator can be compared to both ORACLE and UNIFORM reporting the *overhead*

$$\mathrm{ovh}(m) = \frac{t(m) - t_O(m)}{t_O(m)}, \tag{9.1}$$

which is 0 for ORACLE ($t(m) = t_O(m)$), and N for UNIFORM ($t(m) = t_{\mathcal{U}}(m)$). For the competitions, the overhead is measured regardless of the number of instances solved, so for UNIFORM it may be less than N if this allocator does not solve all instances.

The following quantities may be reported for the whole set of M instances, as a measure of performance on the benchmark, or for any $J \leq M$, to display an improvement in performance along the task sequence. We will describe the performance of an allocator after J instances reporting the *cumulative time*

$$\mathrm{T}(J) = \sum_{m=1}^{J} t(m), \tag{9.2}$$

and the *cumulative overhead*

$$\mathrm{OVH}(J) = \frac{\sum_{m=1}^{J}[t(m) - t_O(m)]}{\sum_{m=1}^{J} t_O(m)}, \tag{9.3}$$

relative to the performance of the oracle.

The best per set best algorithm is labeled WINNER, and its performance is (2.1)

$$\mathrm{T}_W(J) = \min_n \sum \sum_{m=1}^{J} t_n(m). \tag{9.4}$$

To compare with this algorithm, we report the *speedup*

$$\mathrm{SU}(J) = \frac{\mathrm{T}_W(J)}{\mathrm{T}(J)}, \tag{9.5}$$

which is larger than 1 for methods with a cumulative time smaller than T_W. For competitions, we consider as WINNER the algorithm which solves most instances, breaking ties based on time. This is not necessarily the criterion used in the actual competition, but it allows us to compare with [Streeter, 2007].

We will report the same kind of plots for each benchmark. In the following we describe each plot in detail, giving examples for the SAT 2007 competition, Random category (see Sec. 1.3, Sec. 9.3.1).

Log-log comparison with Oracle We will use such plots to report the performances of WINNER and GAMBLETA, comparing them with both UNIFORM and ORACLE. We already saw an example of such plots in Section 1.3. In Figure 9.1 we report these plots for the same competition (Sat 2007, Random category, see Sec. 9.3.1). In these plots, each point corresponds to a single problem instance. When WINNER is compared to ORACLE, 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). Such plots can then 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 by a time allocator. In the case reported in Figure 9.1, UNIFORM would have solved the most instances, as one could guess looking at the many points above the UNIFORM line. When plotting results of a competition, we limit to instances which could be solved by at least one algorithm: instances which the WINNER could not solve before the timeout are represented by circles, whose ordinate is the timeout. 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. Note that points which are exactly on the line of UNIFORM are likely to correspond to instances for which the BPS picked the uniform allocator.

Runtime distributions. A common tool to describe the performance of both algorithms and allocators is the runtime distribution. We will therefore display, for each benchmark, the CDF of the RTDs on the set of all instances, for GAMBLETA and all comparison terms, as well as for each algorithm, always evaluated using the non-parametric product limit estimator (3.16). Recall that this estimator is constant among uncensored observations, so it is displayed in the plots as a stepwise function. Figure 9.2 (left) reports such plot for the SAT'07 Random category. Comparing algorithms or allocators based on their CDF has only a statistical meaning: if two lines never intersect, it means that one term is likely to be faster than the other, but it does not determine the result on a single instance. For example, in this plot you can see that GAMBLETA statistically dominates UNIFORM, as its CDF is always larger, but from Figure 9.1 (right) we know it is WTU on several occasions.

Cumulative overhead. Plotting the overhead (9.1) for each instance is not very informative, as the impact on performance depends on the actual runtimes: a very large overhead for an easy instance may be irrelevant, compared to a small overhead for an instance where the fastest algorithm spends several hours. In order to portray the improvement in performance along the task sequence, we can plot the evolution of cumulative overhead (9.3) vs. the number of instances solved. The fact that the instance order differs for each run the lines for a single run very noisy, due to the large variability in runtimes. We will then report the average, with a 95% confidence interval evaluated over the 20 runs, as in Figure 9.2 (right). In this case the overhead decreases quickly to reach that of UNIFORM, and continues decreasing at a slower rate. We will see other examples in which GAMBLETA sensibly improves over UNIFORM.

Overall performance. The performance measure which we aim at improving is the total time $\mathrm{T}(M)$ spent in solving the whole sequence of M instances. We will report this quantity for each run, using box-plots. For competitions, the number of instances solved before time out is the most important performance indicator, and time is used to resolve draws among algorithms. In this case, we will plot the average time $\mathrm{T}(M)/M$ vs. the number of instances solved, for each run of GAMBLETA, and for the comparison terms. Figure 9.3 (left) shows such plot for the Random SAT competition. In these plots, each point represents a performance on the whole sequence. The diamond is the performance of SATZILLA, which participated to the competition as one of the algorithms, and is the actual winner in this case, according to the rules used in the competition. Note that this time allocator used a different set of algorithms, therefore it cannot be fairly compared to GAMBLETA based on its results. The other symbols indicate the performance of other time allocators, from [Streeter, 2007], used as comparison terms: the square represents the performance of the online greedy algorithm, which generates task-switching schedules only based on runtimes observed during the sequence, starting from scratch, so it is a fair comparison

term for GAMBLETA. The star corresponds instead to the offline greedy method (2.23). Note that this method is based on the 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, as in this case.

Regarding the competition, these plots allow to appreciate the gap among WINNER and ORACLE, which is often important, as in this case: only in a few competitions the two terms are almost equal, but they are characterized by few instances and very short runtimes. 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.

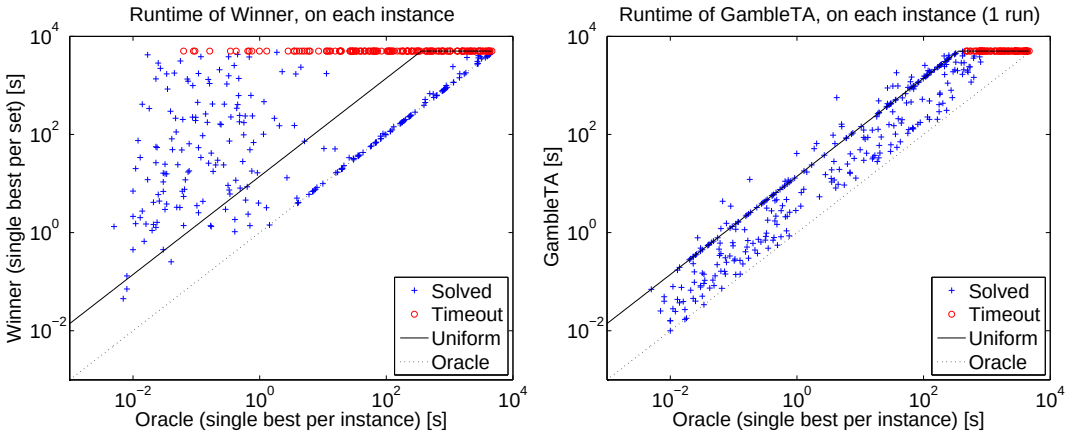


Figure 9.1. **Log-log comparison with Oracle.** Right: WINNER. Left: GAMBLETA (1 run). In these plots, each point corresponds to a single problem instance. The horizontal axis reports the runtime of the per instance best algorithm, which corresponds to the performance of ORACLE. The vertical axis reports the runtime of the algorithm or allocator being compared. As the same limits are used on each axis, the diagonal of the plot corresponds to the performance of ORACLE, which means there will never be points below the diagonal. The performance of UNIFORM is represented by a continuous line, parallel to the diagonal, which becomes horizontal at the timeout: points whose abscissa lies in the horizontal portion correspond to instances where UNIFORM times out. Instances where the comparison term times out are instead represented by red circles, whose ordinate is the timeout. For GAMBLETA (right), points which are exactly on the line of UNIFORM likely correspond to instances for which the BPS picked the uniform allocator. Points above the line are instances on which GAMBLETA is WTU. Note that information about the order with which instances are solved is lost in this kind of plots.

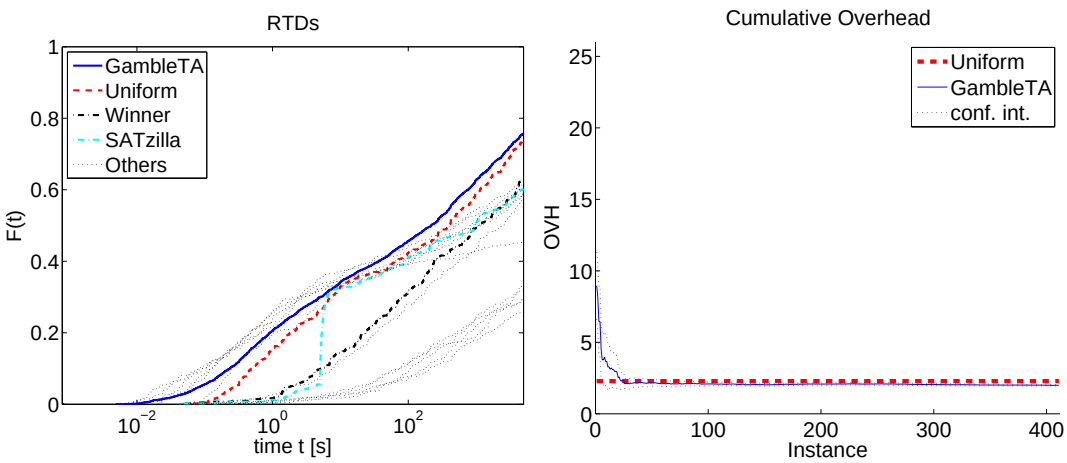
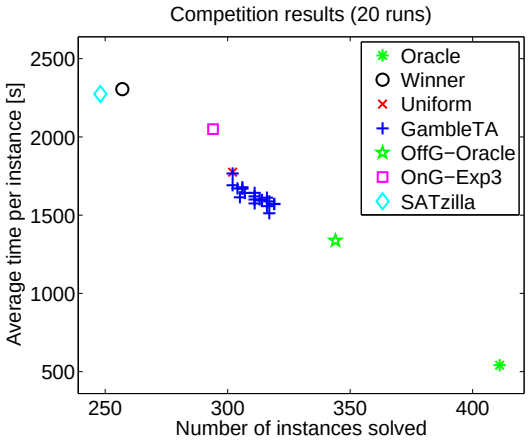


Figure 9.2. Left: **runtime distributions.** The CDF of the RTD of GAMBLETA, UNIFORM, WINNER, and other algorithms in the set. The horizontal axis, reporting runtimes, has a logarithmic scale. The left border corresponds to the timeout used in the competition. The fact that none of the lines reaches the unity means that none of the corresponding algorithms could solve all instances before timeout. Right: **cumulative overhead.** Evolution along the instance sequence, averaged over 20 runs (blue continuous line). The dotted lines represent a confidence interval, evaluated based on the Z distribution. The red dotted line reports the final overhead of UNIFORM, which can be less than N, as in this case, due to timeouts.



	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
SATZILLA	248	2274.5	1.01	3.21

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

Figure 9.3. **Overall performance.** Right: graphical representation. In these plots, each point represents a performance on the whole sequence. Points towards the right correspond to better performances (more instances solved): for points along the same vertical, lower is better (same instances solved in less time). Each plus sign corresponds to a run of GAMBLETA; the asterisk and the cross indicate the performances of ORACLE and UNIFORM, respectively, while the circle corresponds to the WINNER. Note that in this case these do not change, as each run is done using the same runtime values, the only ones available. Left: summary table. Quantities reported for GAMBLETA are confidence bounds evaluated on 20 runs. Upper bounds are reported for quantities to be minimized, as runtime and overhead (OVH); lower bounds for those to be maximized, as the number of solved instances and speedup (SU). Below the table we indicate the number of runs on which GAMBLETA would have, respectively, won the competition (counting only algorithm runtimes), been worse than UNIFORM (WTU), and worse than ONG-EXP3 (WTG).

9.3 Solver competitions

In this section we present experiments with runtime data[4] from 43 recent solver competitions, the same used by Streeter [2007], and compare GAMBLETA with their online greedy allocator ([Streeter and Smith, 2008], see Sections 2.5, 4.5,) when its performance is available. In this case the algorithm set is composed of all the contestants in the competition. The instance set is composed of all instances which could be solved by at least one of the algorithms before the timeout.

While instance features could be obtained online for some competitions, they were not available for most of them. Streeter [2007] 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: in this case the RTD models were simple Kaplan-Meier hazard estimators (3.15).

We present results in tabular form for all competitions, and in graphical form for the most relevant ones, in terms of number of instances and runtimes involved. Most competitions were held at the SAT 2007 conference. In the following subsections we describe results of each competition in more detail: the next one describes competitions on which a comparison term is available.

9.3.1 Satisfiability (SAT 2007, 2009)

Four 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, and the special track on and-inverter graphs (AIG). For the first three we can also compare with the results of another online allocator, the online greedy method from Streeter and Smith [2008], using EXP3 [Auer et al., 2002] as BPS, labeled ONG-EXP3 in the following. As SATZILLA took part to these competitions, we will also highlight its results; however, we 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. 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 [Streeter, 2007]. 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[5].

The results for both GAMBLETA and ONG-EXP3 are obtained on the same data, and both algorithms are online, so in this case the comparison is fair. We also report the results of the offline greedy allocator from Streeter and Smith [2008], labeled OFFG-ORACLE, to underline the fact that it allocates time based on prior knowledge of runtimes. In this case a comparison is

[4] While all data is available online, we obtained it directly from Streeter, who kindly saved us the time consuming task of formatting it. In his work, experiments are performed on 44 competitions, one of which was missing in the data we received: the Miscellanea category from QBFEVAL'07, which consisted of only 67 instances.

[5] 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/`.

obviously not fair. 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 9.4 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 19000, and clause-to-variable ratio between 2.67 (underconstrained) and 89 (heavily overconstrained). The winner on this category, `minisatSAT`, could solve 98 instances: its performance is compared to ORACLE in the top-left plot of Fig. 9.4. 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. Also from the RTD plots it can be seen that this algorithm more or less dominates the scene, together with SATZILLA, which ranked second in this case: SATZILLA is better on quantiles until 0.1, corresponding to instances at the top-left corner of the plot, it dominates until quantile 0.4 or so, but it solves 4 instances less. The other contestants lag far behind.

A similar situation can be observed in the industrial category (Fig. 9.5), consisting of 166 hard instances from real industrial applications, mostly hardware verification, 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. 9.5).

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. 9.5, bottom-right plot), and it it outperformed by UNIFORM (WTU) and ONG-EXP3 (WTG) on 5 and 16 runs respectively. In both cases, the overhead of GAMBLETA drops quickly after the first $20 - 30$ instances, and the overall performance is comparable to that of WINNER: with such short instance sequences, further performance improvements are arguably difficult.

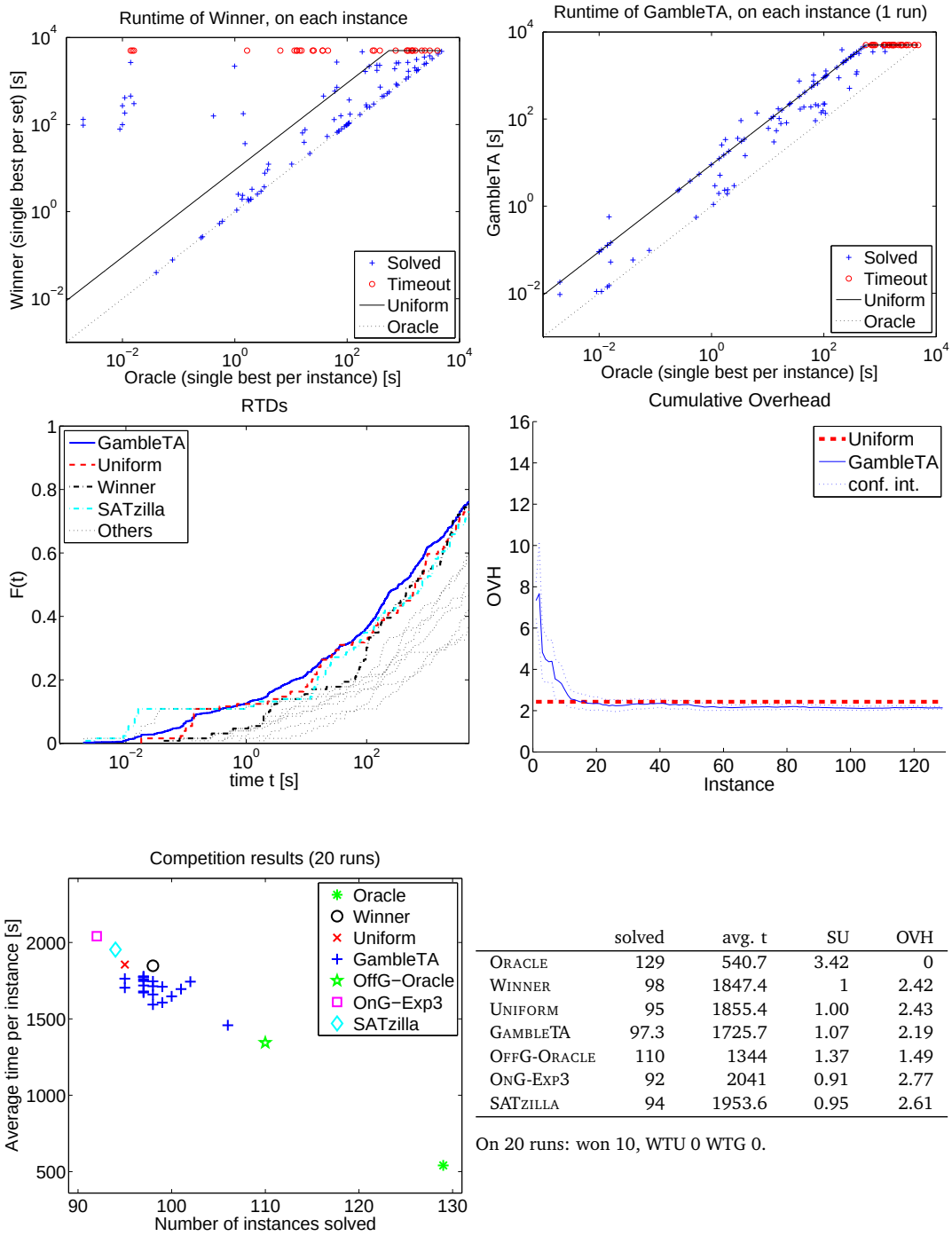
The situation changes in the random category (Fig. 9.6), which consists of 411 randomly generated instances, with a number of variables between 45 and 19000, and ratio between 2.67 and 89. We already saw the performance of the winner, `March KS`, in Figure 1.1: it can only solve 257 instances, and is WTU on many of them (the points above the continuous line in the top-left plot). The improvement obtained by GAMBLETA in this case 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 further improve the performance, solving between 302 and 319 instances: its RTD dominates that of UNIFORM.

In all three cases, there is a huge gap among the 4-approximation of the best per set schedule, OFFG-ORACLE, and the best per instance schedule, ORACLE, therefore we expect that these results could be improved using informative instance features.

Figure 9.7 reports the results of the special track on And-Inverter Graphs (AIG), which are used to encode formal software and hardware verification problems into SAT format: in this case GAMBLETA is always better than UNIFORM, and improves over WINNER on 7 runs over 20. We do not have data for the comparison terms on this and the following competitions.

Figures 9.8–9.10 report instead results of the three categories of the 2009 edition of the competition. In that occasion, the scoring system was simplified to the same criterion we used here, such that the winner was the algorithm which solved the most instances, in the least time.

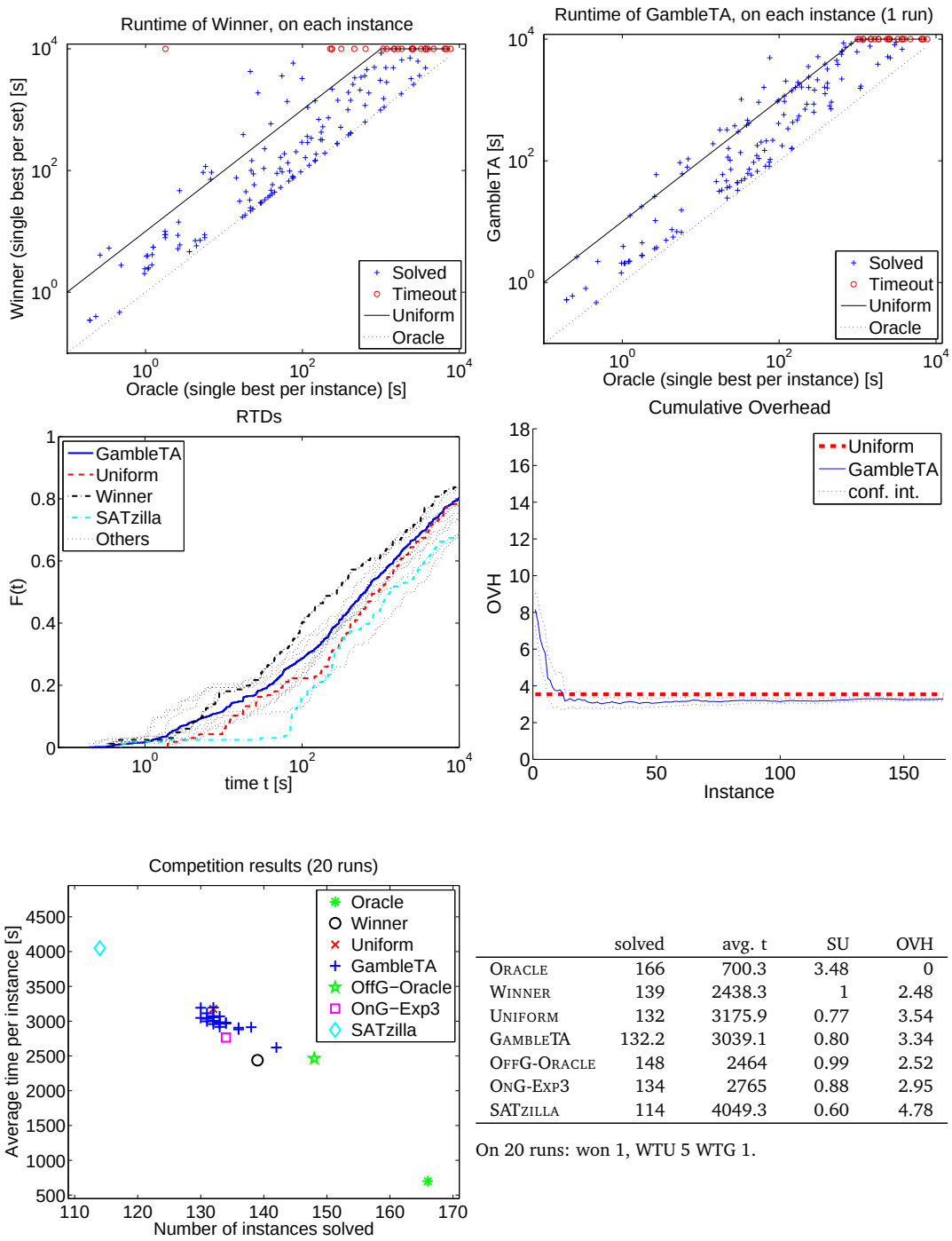
In this edition, UNIFORM would not have won in any of the category. On 20 runs, GAMBLETA would have won 10 and 19 in the crafted and random categories, respectively.



	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
SATzilla	94	1953.6	0.95	2.61

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

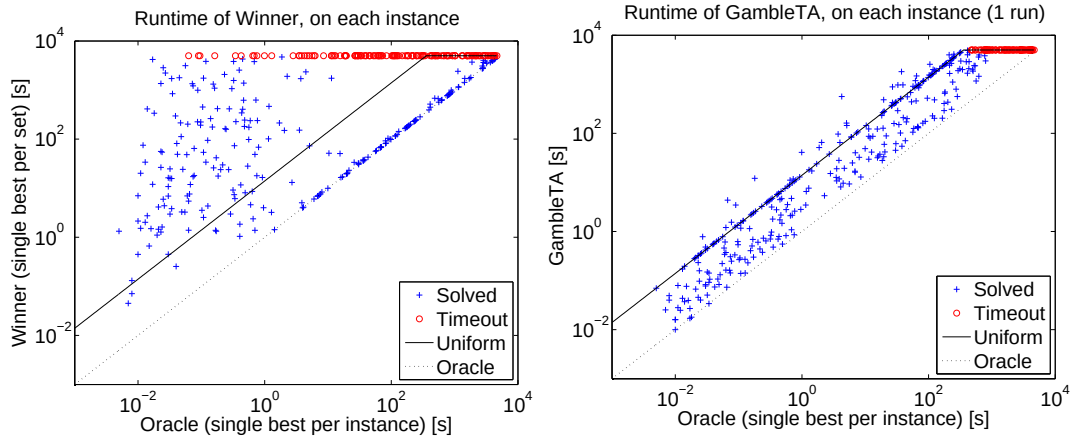
Figure 9.4. SAT'07, Hand-crafted, 9 algorithms, 129 instances, timeout 5000 s. Winner: `minisatSAT`.

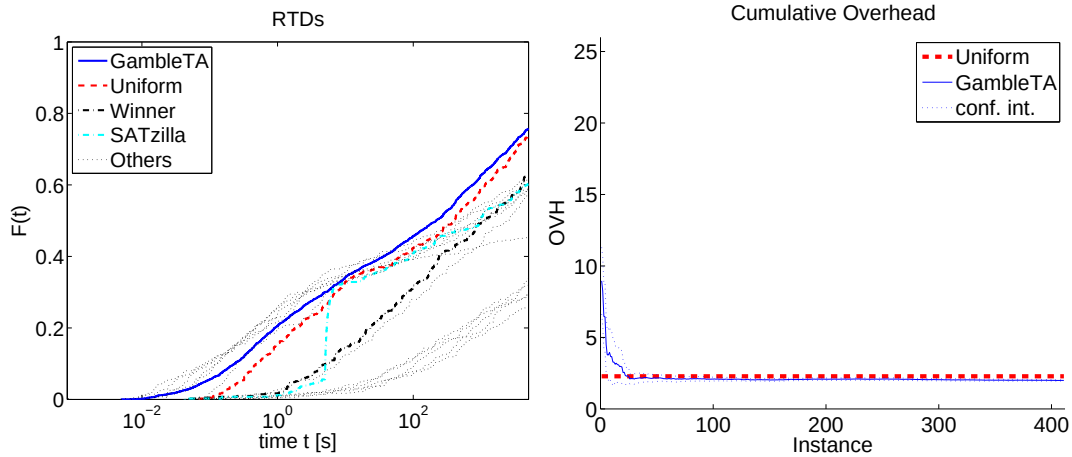


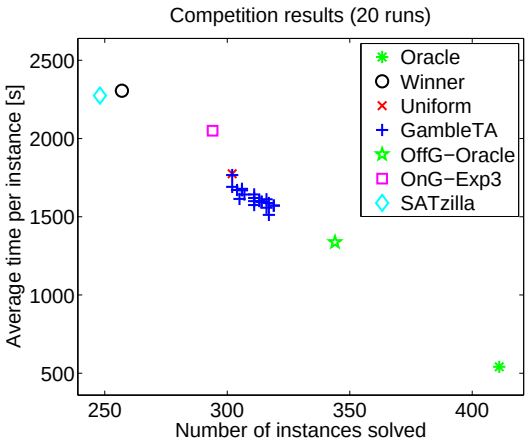
	solved	avg. t	SU	OVH
Oracle	166	700.3	3.48	0
Winner	139	2438.3	1	2.48
Uniform	132	3175.9	0.77	3.54
GambleTA	132.2	3039.1	0.80	3.34
OffG-Oracle	148	2464	0.99	2.52
OnG-Exp3	134	2765	0.88	2.95
SATzilla	114	4049.3	0.60	4.78

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

Figure 9.5. SAT'07, Industrial, 10 algorithms, 166 instances, timeout 10000 s. Winner: `Rsat`.



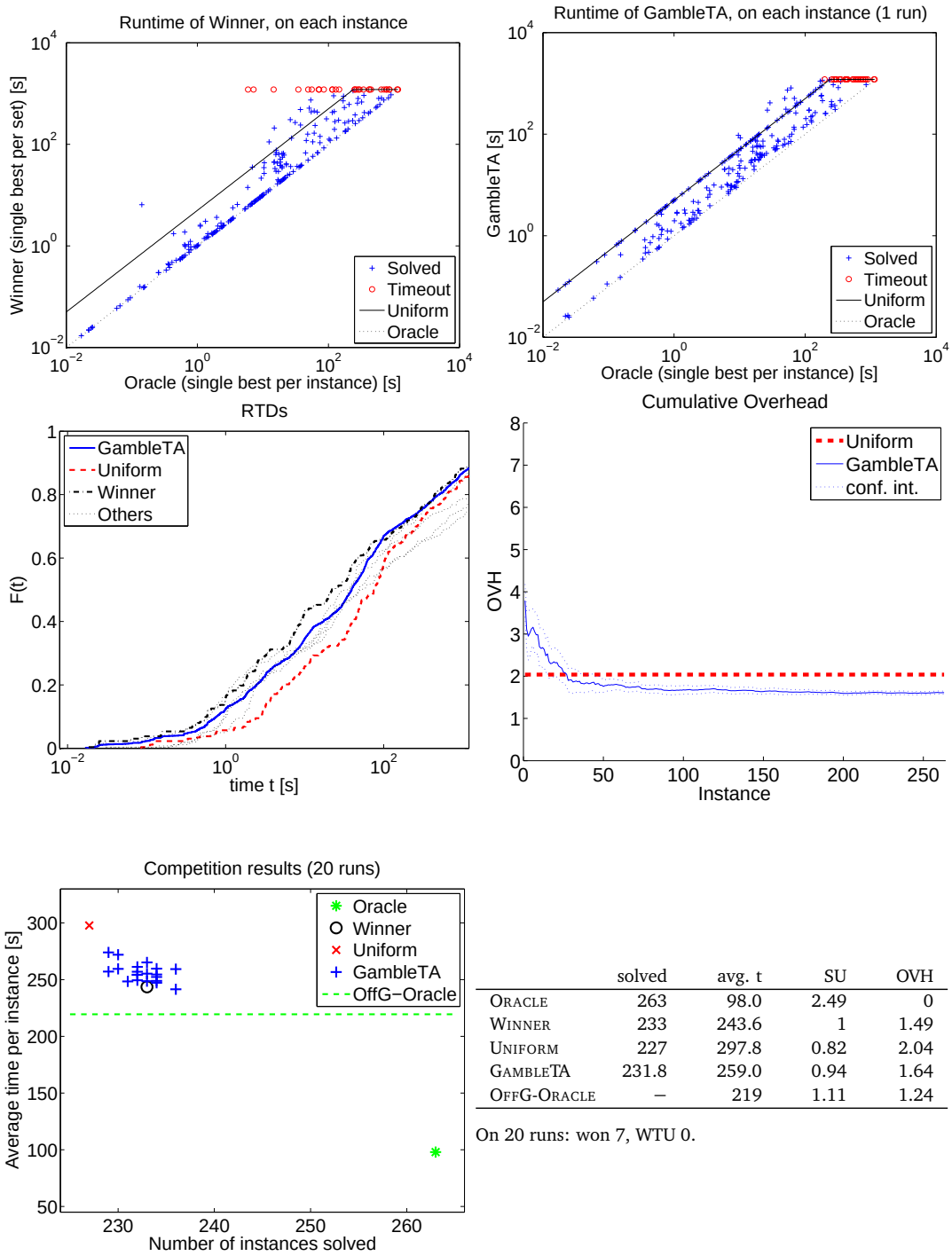




	solved	avg. t	SU	OVH
Oracle	411	540.8	4.26	0
Winner	257	2305.4	1	3.26
Uniform	302	1774.9	1.30	2.28
GambleTA	309.0	1639.6	1.41	2.03
OffG-Oracle	344	1337	1.72	1.47
OnG-Exp3	294	2050	1.12	2.79
SATzilla	248	2274.5	1.01	3.21

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

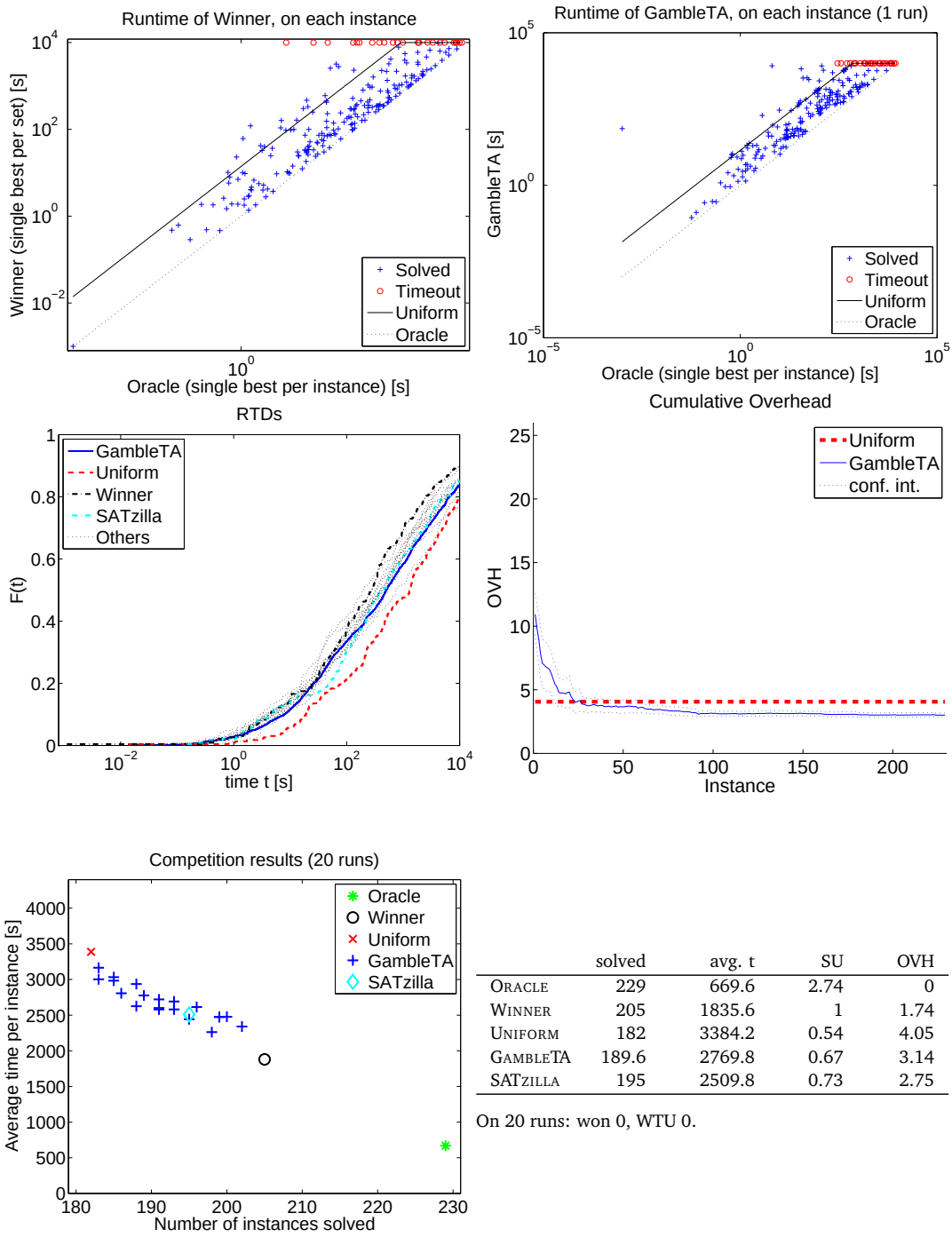
Figure 9.6. SAT'07, Random, 14 algorithms, 411 instances, timeout 5000 s. Winner: `March KS`. Uniform would have won.



	solved	avg. t	SU	OVH
Oracle	263	98.0	2.49	0
Winner	233	243.6	1	1.49
Uniform	227	297.8	0.82	2.04
GambleTA	231.8	259.0	0.94	1.64
OffG-Oracle	–	219	1.11	1.24

On 20 runs: won 7, WTU 0.

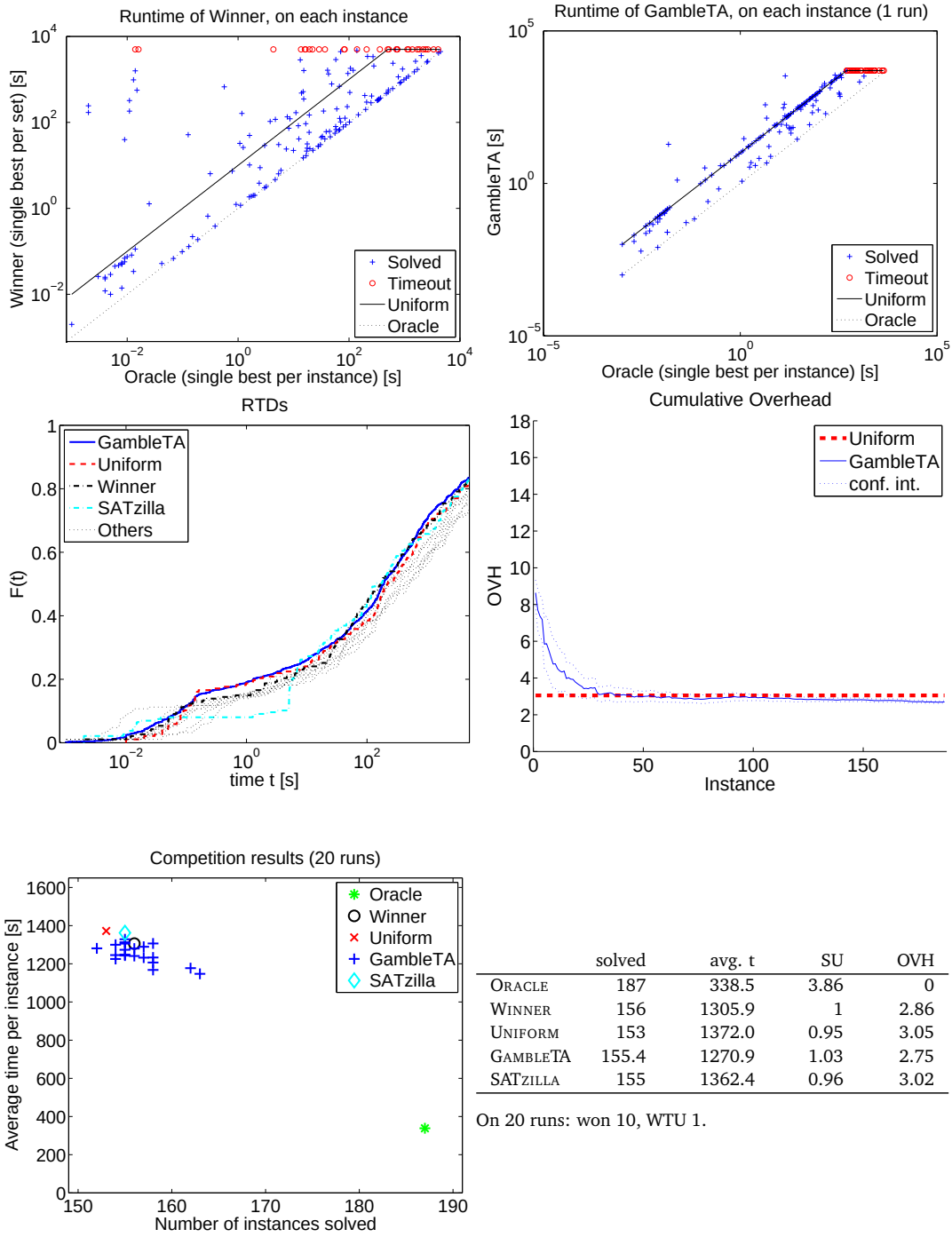
Figure 9.7. SAT'07, And-Inverter Graphs, 5 algorithms, 263 instances, timeout 1200 s. Winner: `aig-cmusat`.



	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
SATZILLA	195	2509.8	0.73	2.75

On 20 runs: won 0, WTU 0.

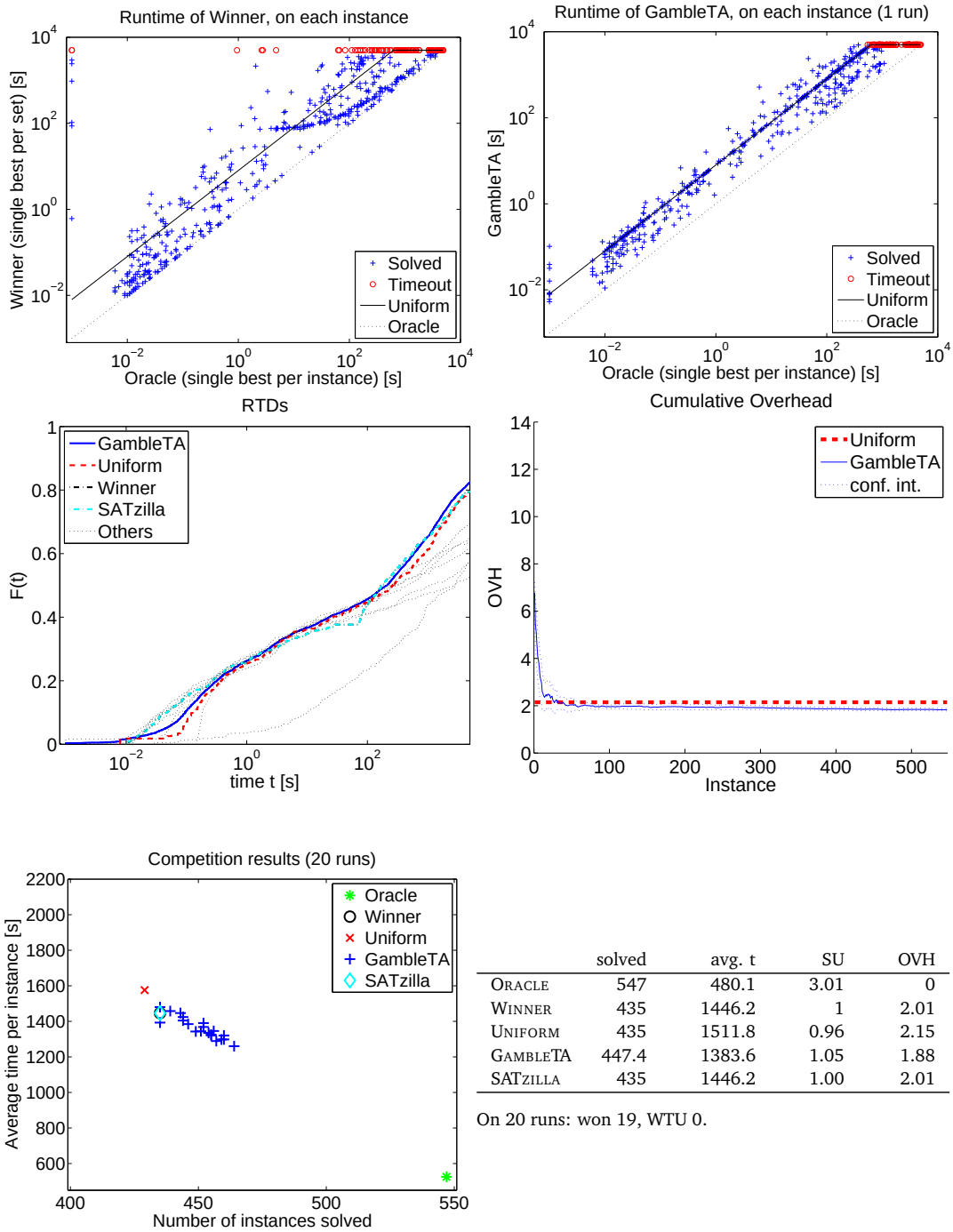
Figure 9.8. SAT'09, Application, 14 algorithms, 229 instances, timeout 10000 s. WINNER: `precosat`.



	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
SATZILLA	155	1362.4	0.96	3.02

On 20 runs: won 10, WTU 1.

Figure 9.9. SAT'09, Crafted, 10 algorithms, 187 instances, timeout 5000 s. WINNER: `clasp`.



	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
SATZILLA	435	1446.2	1.00	2.01

On 20 runs: won 19, WTU 0.

Figure 9.10. SAT'09, Random, 8 algorithms, 547 instances, timeout 5000 s. WINNER: `SATzilla`.

9.3.2 Quantified Boolean formulas (SAT 2007)

Another competition was held at the SAT 2007 conference among quantified Boolean formulas (QBF) solvers. This problem is a generalization of the SAT problem, where clauses can be formed using also the operators $\exists$ and $\forall$, in addition to the negation. In Figures 9.11 and 9.12 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. 9.11), GAMBLETA improves sensibly on the performance of the winner, which does not clearly dominate, and it 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, and it does not seem to further improve after the first 200 instances.

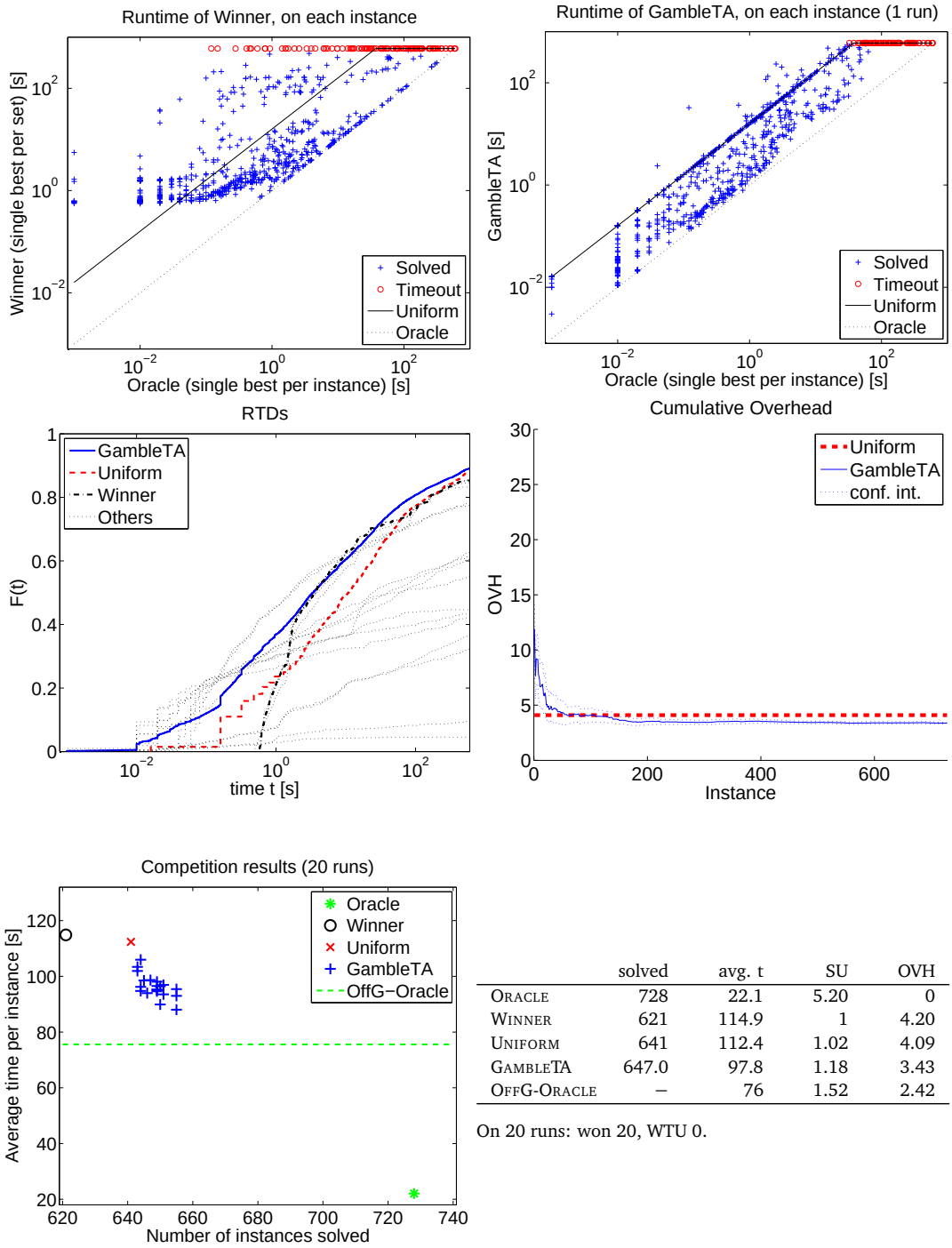
On the shorter sequence of Horn clause formulas, there are two algorithms which dominate the others: WINNER solves all 287 instances but one. GAMBLETA does not solve much more instances than UNIFORM, but it is much faster: its performance is more noisy in this case.

9.3.3 Max-SAT (SAT 2007)

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. Figures 9.13 to 9.16 report results for the four categories of the competition. GAMBLETA won on 5 runs in the partial Max-SAT category (Fig. 9.14), where WINNER is more often outperformed, was competitive in the weighted partial MS (Fig. 9.16), and was clearly inferior in the remaining two categories, where the performance of WINNER is closer to ORACLE: the actual overhead of WINNER was 0.25 in the Max-SAT category (Fig. 9.13), and 0.61 in the weighted MS (Fig. 9.15).

9.3.4 Pseudo Boolean optimization (SAT 2007)

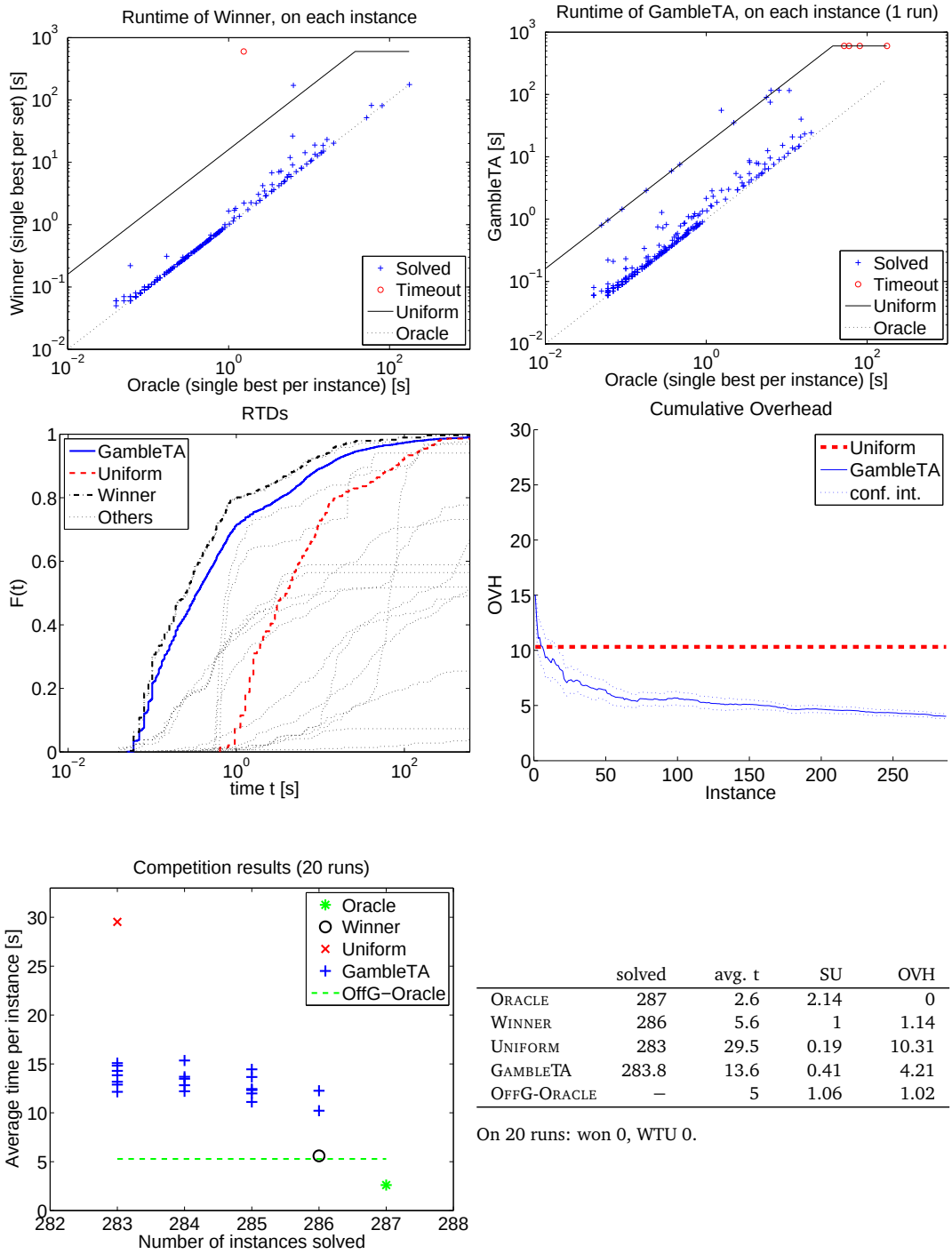
The pseudo-Boolean optimization problem (or zero-one integer programming) consists in minimizing a function of Boolean variables, subject to algebraic constraints. The PB'07 track at SAT 2007 consisted of five categories. In Figures 9.17 to 9.20 we present results for the four largest ones, three for optimization (Big integers, linear constraints; Small integers, linear and nonlinear constraints), one for the decision version of the problem (SAT/UNSAT, small integers, linear constraints), where the aim is only to decide whether an instance is satisfiable. GAMBLETA wins 18 runs in this latter category, where WINNER greatly outperforms UNIFORM (Fig. 9.20), and all 20 runs with small integers, linear constraints (Fig. 9.18), were also UNIFORM would have won. Its performance is similar to WINNER in the two remaining competitions, where it wins on a few runs.



	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.

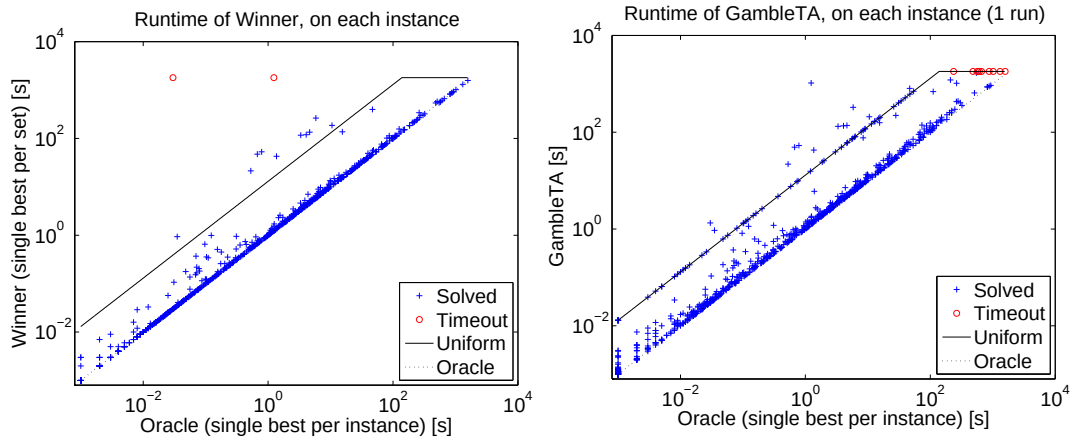
Figure 9.11. QBF'07, Formal verification, 16 algorithms, 728 instances, timeout 600 s. Winner: `AQME-C4.5`. Uniform would have won.

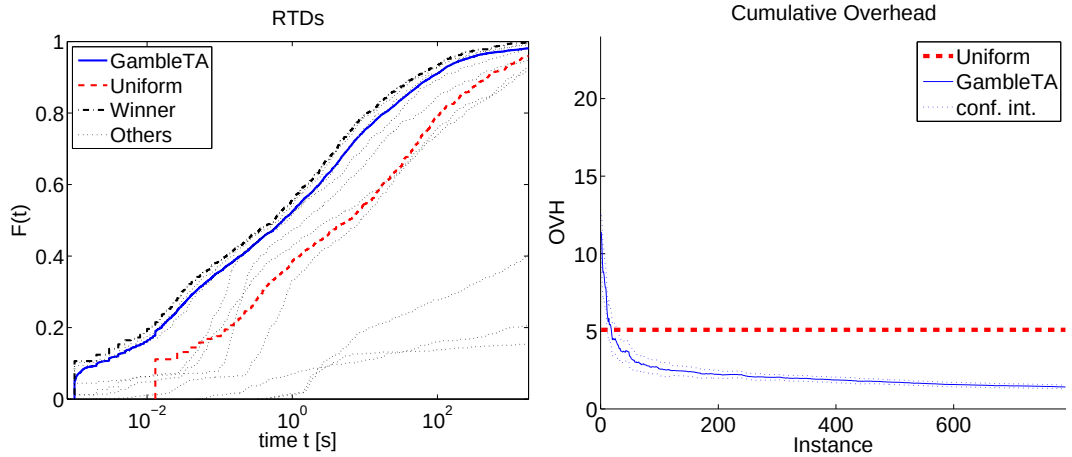


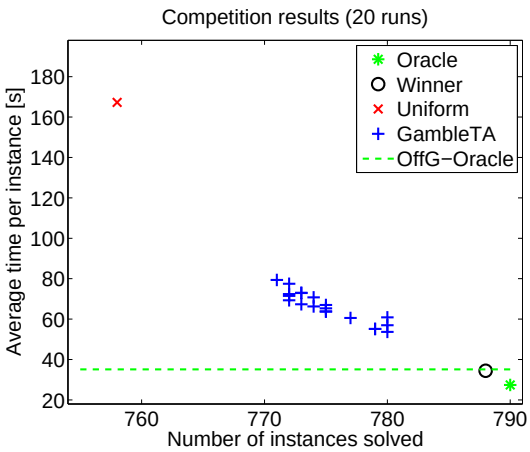
	solved	avg. t	SU	OVH
Oracle	287	2.6	2.14	0
Winner	286	5.6	1	1.14
Uniform	283	29.5	0.19	10.31
GambleTA	283.8	13.6	0.41	4.21
OffG-Oracle	–	5	1.06	1.02

On 20 runs: won 0, WTU 0.

Figure 9.12. QBF'07, Horn clause forms., 16 algorithms, 287 instances, timeout 600 s. Winner: `ncQuBE1.1`.



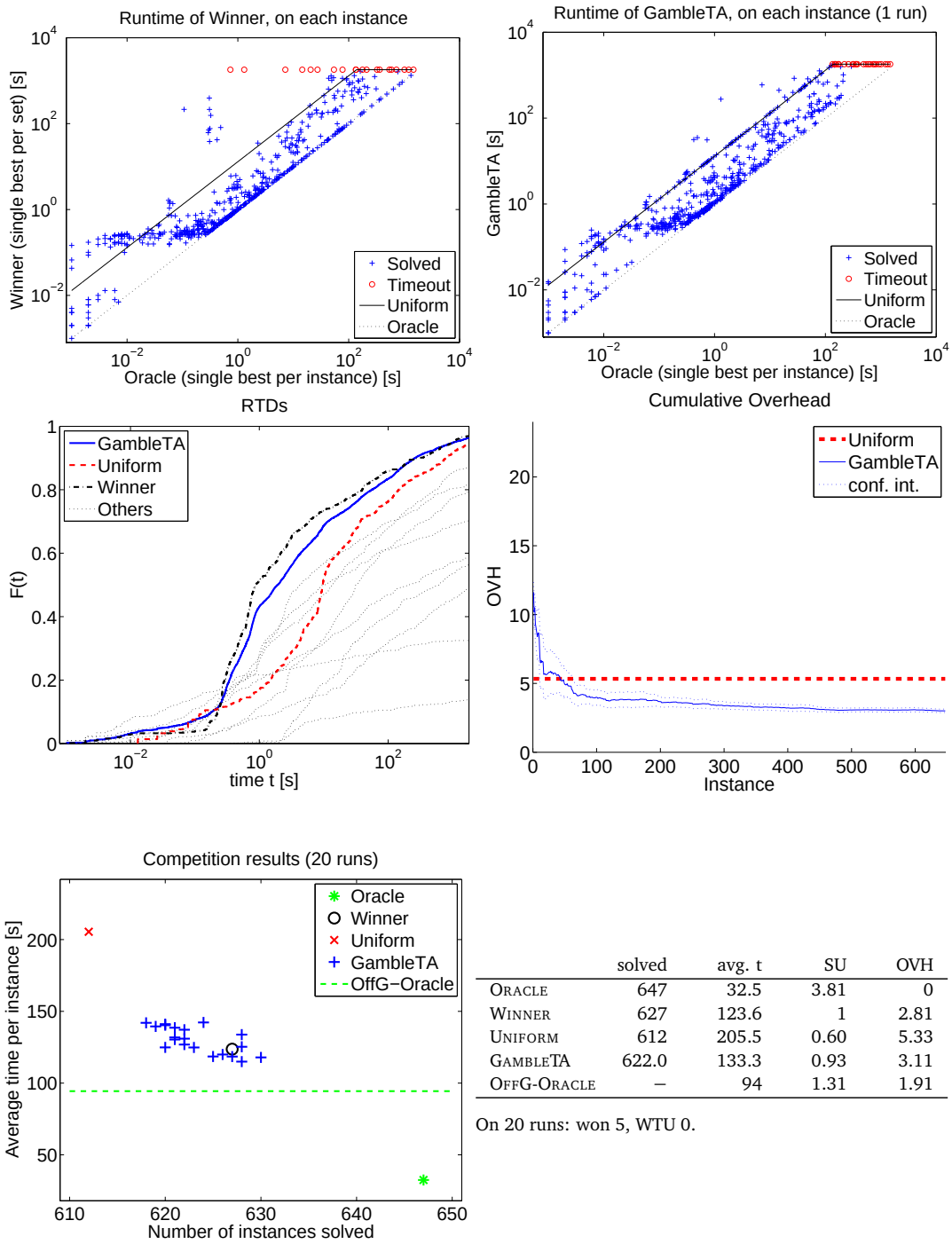




	solved	avg. t	SU	OVH
Oracle	790	27.4	1.26	0
Winner	788	34.4	1	0.26
Uniform	758	167.3	0.21	5.10
GambleTA	773.9	68.9	0.50	1.51
OffG-Oracle	–	35	0.98	0.28

On 20 runs: won 0, WTU 0.

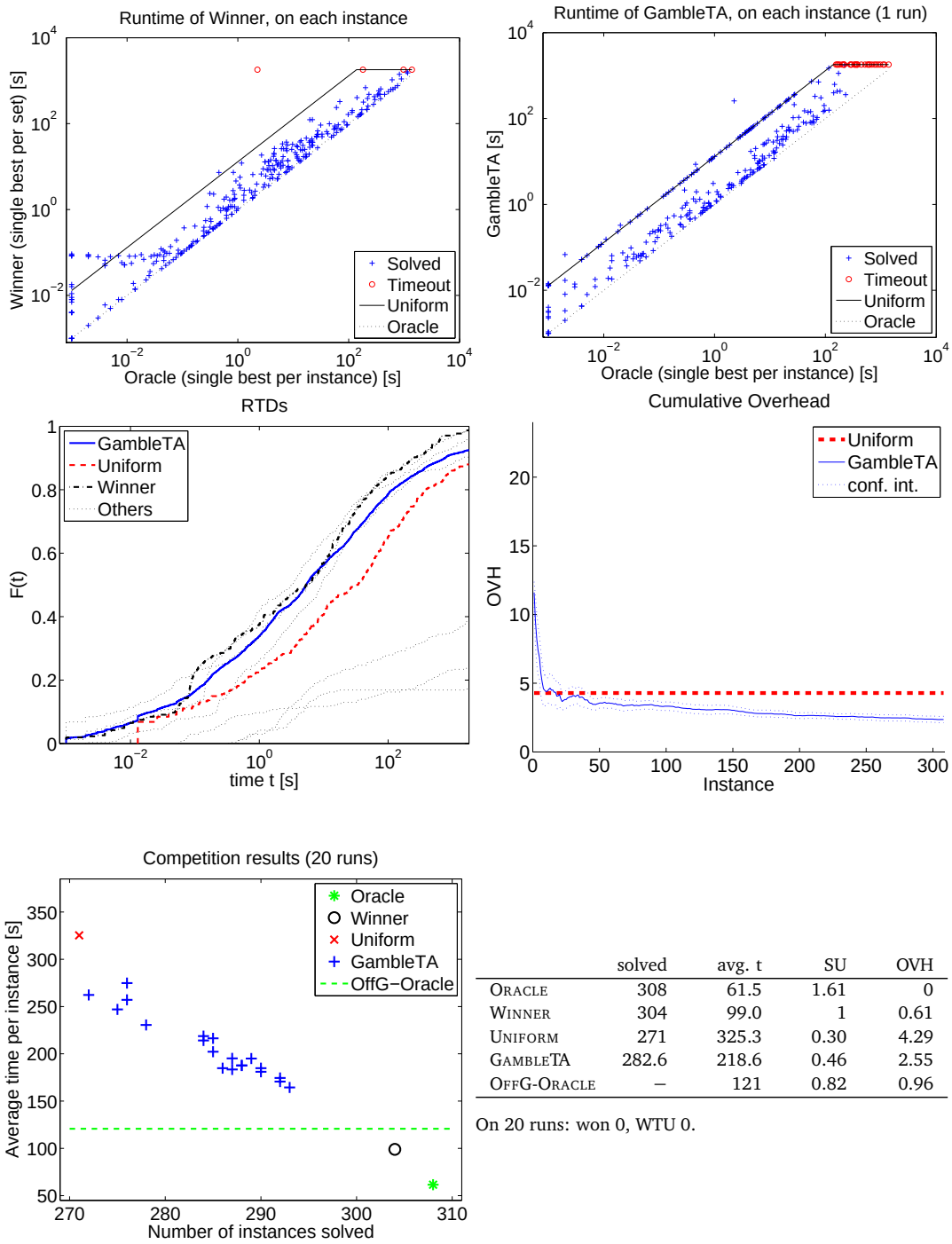
Figure 9.13. Max-SAT'07, Max-SAT, 13 algorithms, 790 instances, timeout 1800 s. Winner: `maxsatz`.



	solved	avg. t	SU	OVH
Oracle	647	32.5	3.81	0
Winner	627	123.6	1	2.81
Uniform	612	205.5	0.60	5.33
GambleTA	622.0	133.3	0.93	3.11
OffG-Oracle	–	94	1.31	1.91

On 20 runs: won 5, WTU 0.

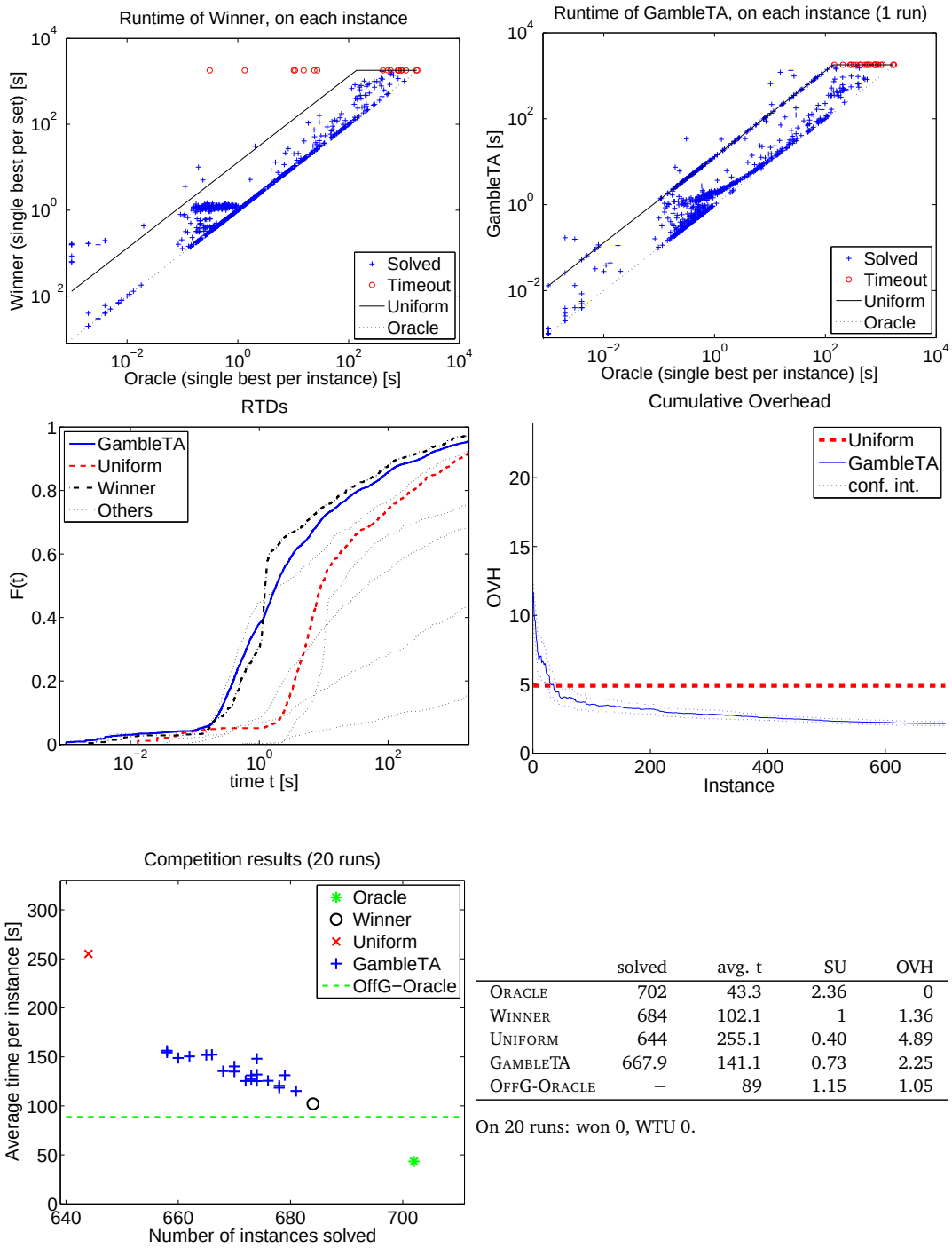
Figure 9.14. Max-SAT'07, Partial MS, 13 algorithms, 647 instances, timeout 1800 s. Winner: `minimaxsat`.



	solved	avg. t	SU	OVH
Oracle	308	61.5	1.61	0
Winner	304	99.0	1	0.61
Uniform	271	325.3	0.30	4.29
GambleTA	282.6	218.6	0.46	2.55
OffG-Oracle	–	121	0.82	0.96

On 20 runs: won 0, WTU 0.

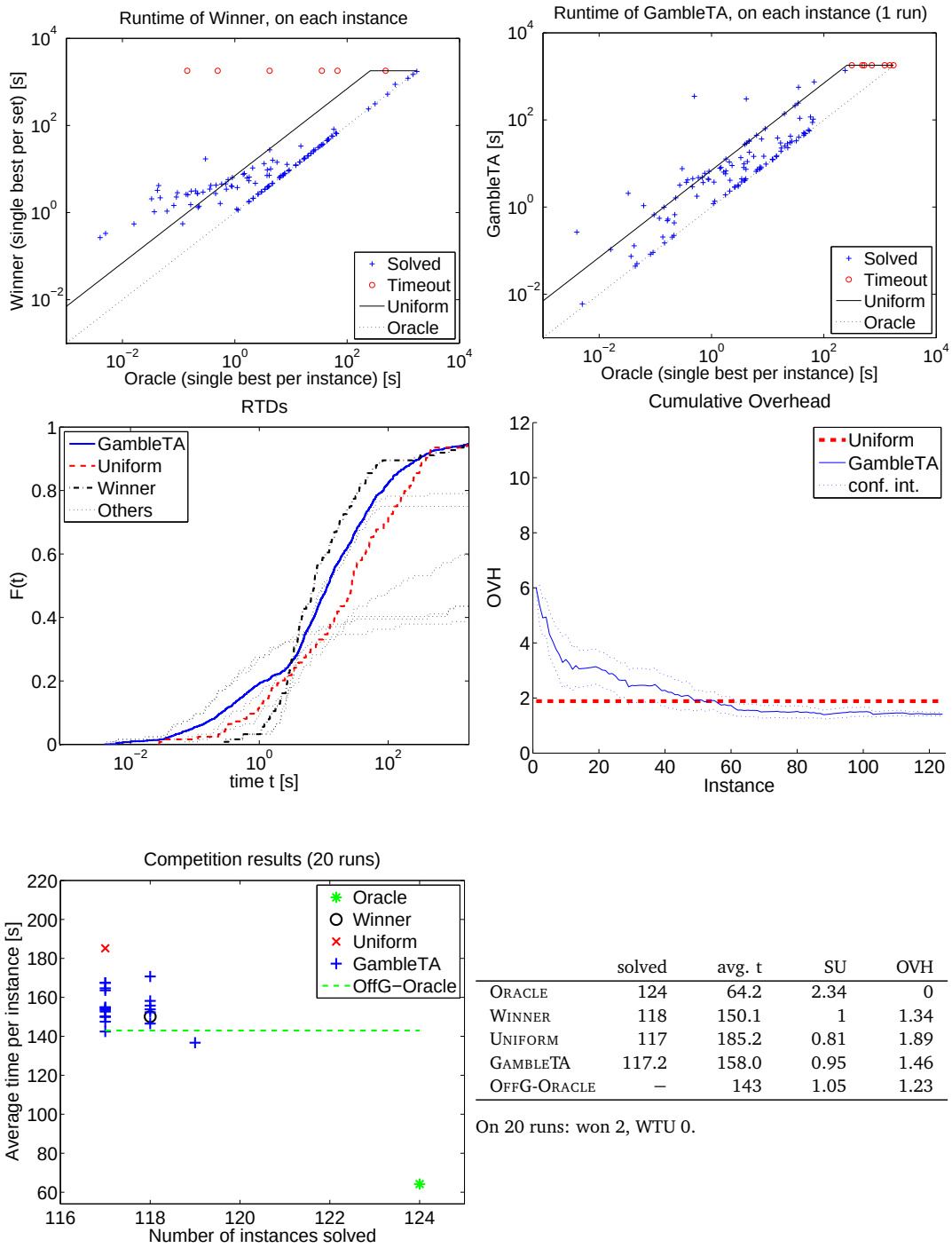
Figure 9.15. Max-SAT'07, Weighted MS, 13 algorithms, 308 instances, timeout 1800 s. Winner: LB-PSAT.



	solved	avg. t	SU	OVH
Oracle	702	43.3	2.36	0
Winner	684	102.1	1	1.36
Uniform	644	255.1	0.40	4.89
GambleTA	667.9	141.1	0.73	2.25
OffG-Oracle	–	89	1.15	1.05

On 20 runs: won 0, WTU 0.

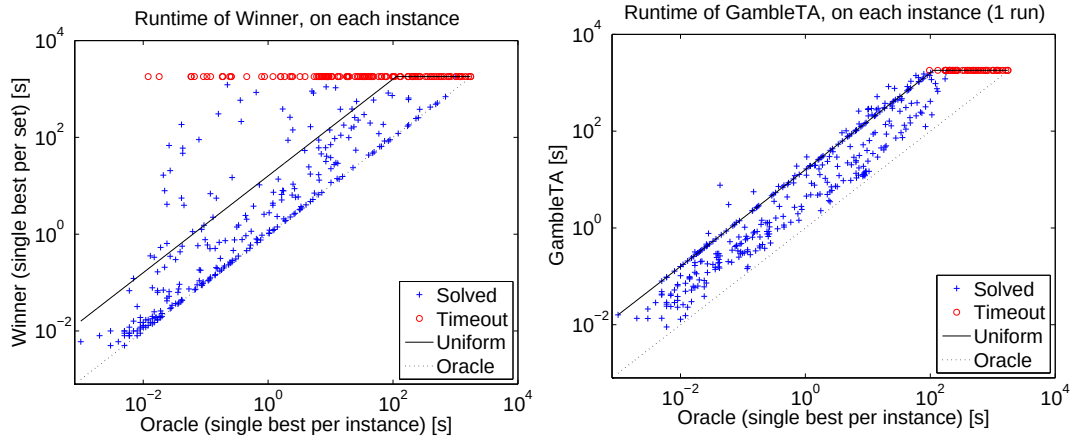
Figure 9.16. Max-SAT'07, Weight. Part., 13 algorithms, 702 instances, timeout 1800 s. Winner: `minimaxsat`.

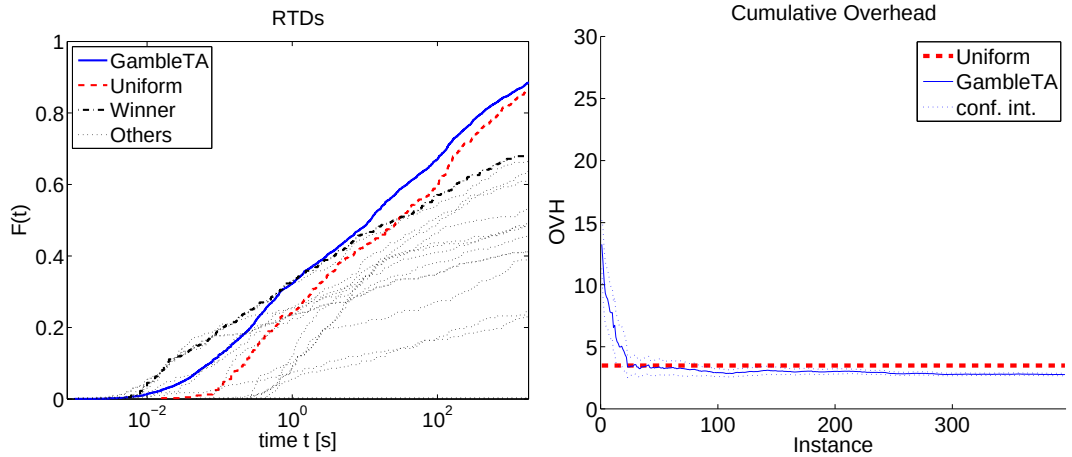


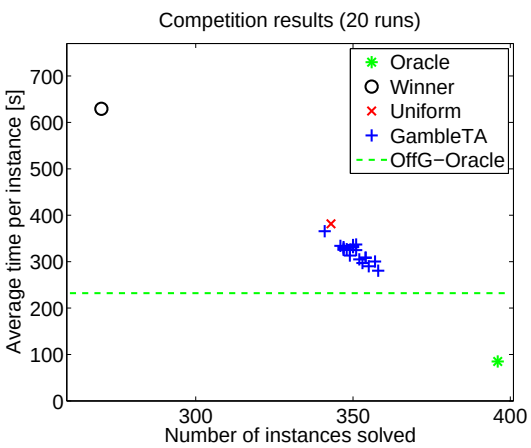
	solved	avg. t	SU	OVH
Oracle	124	64.2	2.34	0
Winner	118	150.1	1	1.34
Uniform	117	185.2	0.81	1.89
GambleTA	117.2	158.0	0.95	1.46
OffG-Oracle	–	143	1.05	1.23

On 20 runs: won 2, WTU 0.

Figure 9.17. PB'07, Opt. big ints., 7 algorithms, 124 instances, timeout 1800 s. Winner: `SAT4JPseudoResolution`.



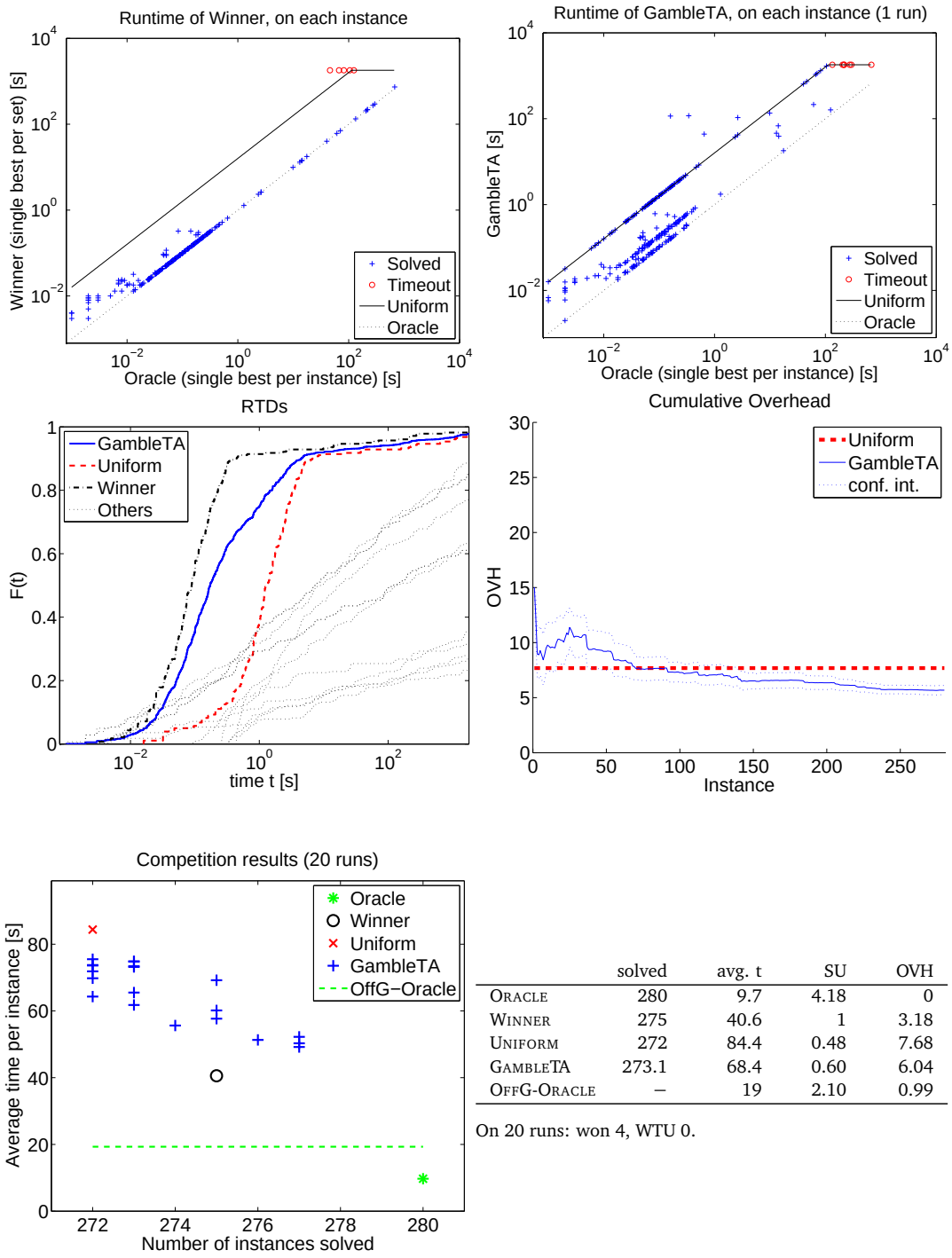




	solved	avg. t	SU	OVH
ORACLE	396	85.0	7.40	0
WINNER	270	629.3	1	6.40
UNIFORM	343	381.5	1.65	3.49
GAMBLETA	349.0	327.3	1.93	2.85
OFFG-ORACLE	–	232	2.71	1.73

On 20 runs: won 20, WTU 1.

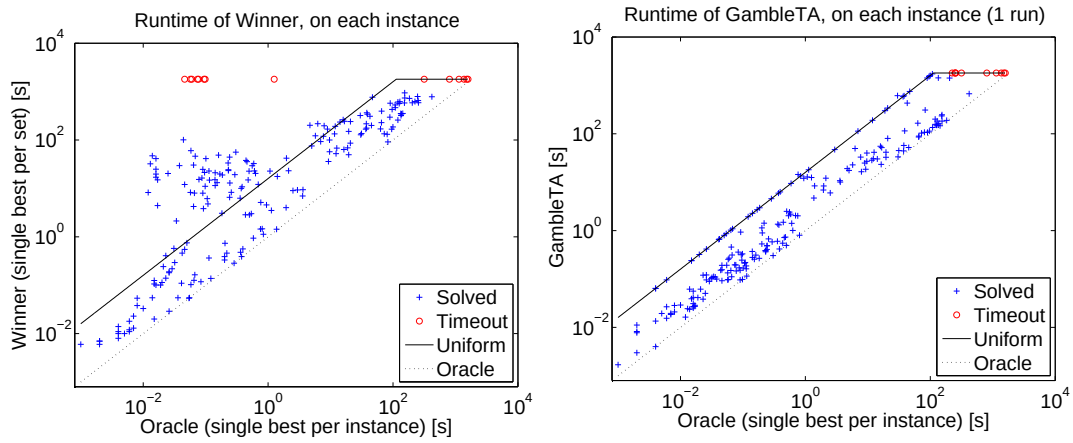
Figure 9.18. PB'07, Opt. small ints., 16 algorithms, 396 instances, timeout 1800 s. WINNER: `bsolo3.`. UNIFORM would have won.

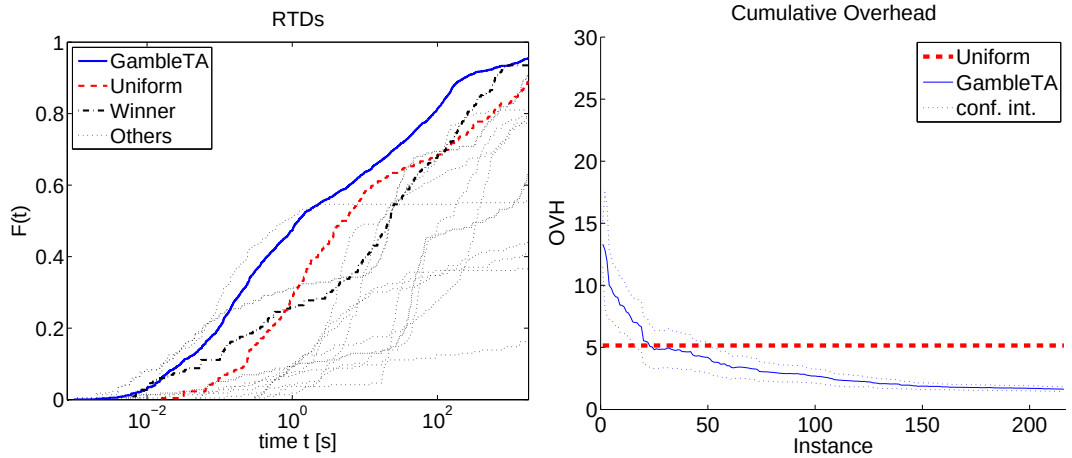


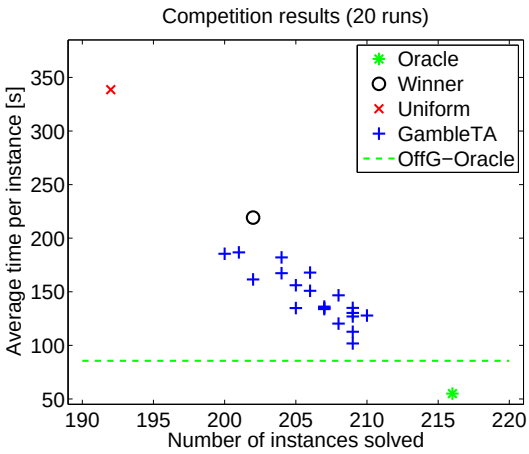
	solved	avg. t	SU	OVH
Oracle	280	9.7	4.18	0
Winner	275	40.6	1	3.18
Uniform	272	84.4	0.48	7.68
GambleTA	273.1	68.4	0.60	6.04
OffG-Oracle	–	19	2.10	0.99

On 20 runs: won 4, WTU 0.

Figure 9.19. PB'07, Opt. sm. ints. nonlin., 16 algorithms, 280 instances, timeout 1800 s. Winner: `minisat+1.14`.







	solved	avg. t	SU	OVH
ORACLE	216	55.0	3.98	0
WINNER	202	219.2	1	2.98
UNIFORM	192	338.6	0.65	5.16
GAMBLETA	205.2	153.8	1.46	1.80
OFFG-ORACLE	–	86	2.56	0.56

On 20 runs: won 18, WTU 0.

Figure 9.20. PB'07, SAT/UNS sm. ints. lin., 16 algorithms, 216 instances, timeout 1800 s. WINNER: `Pueblo1.4`.

9.3.5 Constraint satisfaction (CP 2006)

The CPAI'06 competition was held at the 2006 conference on Constraint Programming (CP 2006). Figures 9.21 to 9.24 present results for four categories of the decision version of the Constraint Satisfaction problem, while Figure 9.25 refers to the optimization version of the problem (Max-CSP), where the number of satisfied constraints is maximized. GAMBLETA systematically wins in three categories (Figs. 9.22, 9.24, 9.25), two of which would have also been won by UNIFORM, and behaves well on the remaining two.

9.3.6 Other competitions

CASC-J3 is an automated theorem proving competition, held during CADE 2007. Figure 9.26 reports results for the category with the longest instance sequence, which would have been won by UNIFORM. GAMBLETA wins on all runs, and is WTU on 6 of them.

Figure 9.27 reports results of the optimal planning track of the IPC-5 competition, held at ICAPS 2006. Planning [Lagoudakis and Koenig, 2004] is an optimization problem, in which solutions are represented by sequences of actions. In this case the solvers had to find a plan with minimum makespan, and prove its optimality. The benchmarks were obtained from real applications in biochemistry, logistics, job shop scheduling.

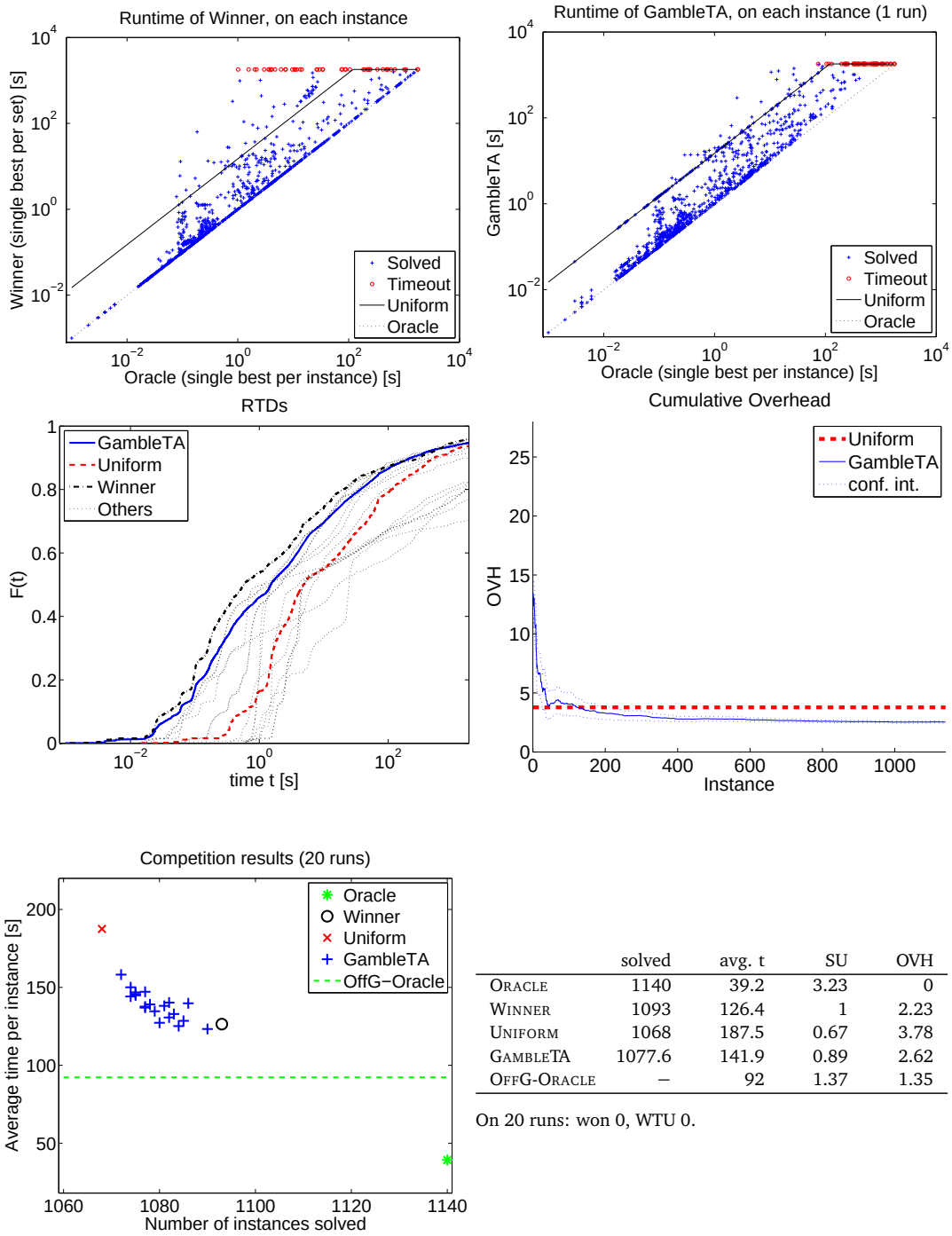
SMT-COMP'07 was held at CAV'07, and is a competition among solvers for satisfiability modulo theories. In this decision problem, the question to be answered is whether a given formula is true, given a first order logic representation of a theory. Typical applications are hardware and software verification. Figure 9.28 reports results for one of the categories, QF LRA (Unquantified linear real arithmetic), where the formula consists of Boolean combinations of inequalities between linear polynomials of real variables.

9.3.7 All competitions, summary of results

To summarize the overall results on all 43 competitions, in Figure 9.29 we report the performance of GAMBLETA, compared to WINNER, in each competition, in terms of average time (upper left plot), and number of instances solved (upper right plot). The results are always competitive with WINNER.

In the lower part of the figure, we report the results as the 43 competitions were a single one, with a total of 12642 instances. In this case WINNER does not correspond to a single algorithm, but to the aggregate results of the winners of each competition. Competitions characterized by more instances and larger runtimes have obviously a larger impact on these overall results. From this data we can see that UNIFORM is already competitive with WINNER, but GAMBLETA improves further. The comparison with OFFG-ORACLE, for which we know only the average time, is impressive if we consider that both this allocator and GAMBLETA are per set (as we do not use instance features in this case), but while OFFG-ORACLE is based on prior knowledge of the exact runtimes of all algorithms, GAMBLETA starts from scratch, solving each problem instance only one, thus observing a single uncensored runtime for each instance.

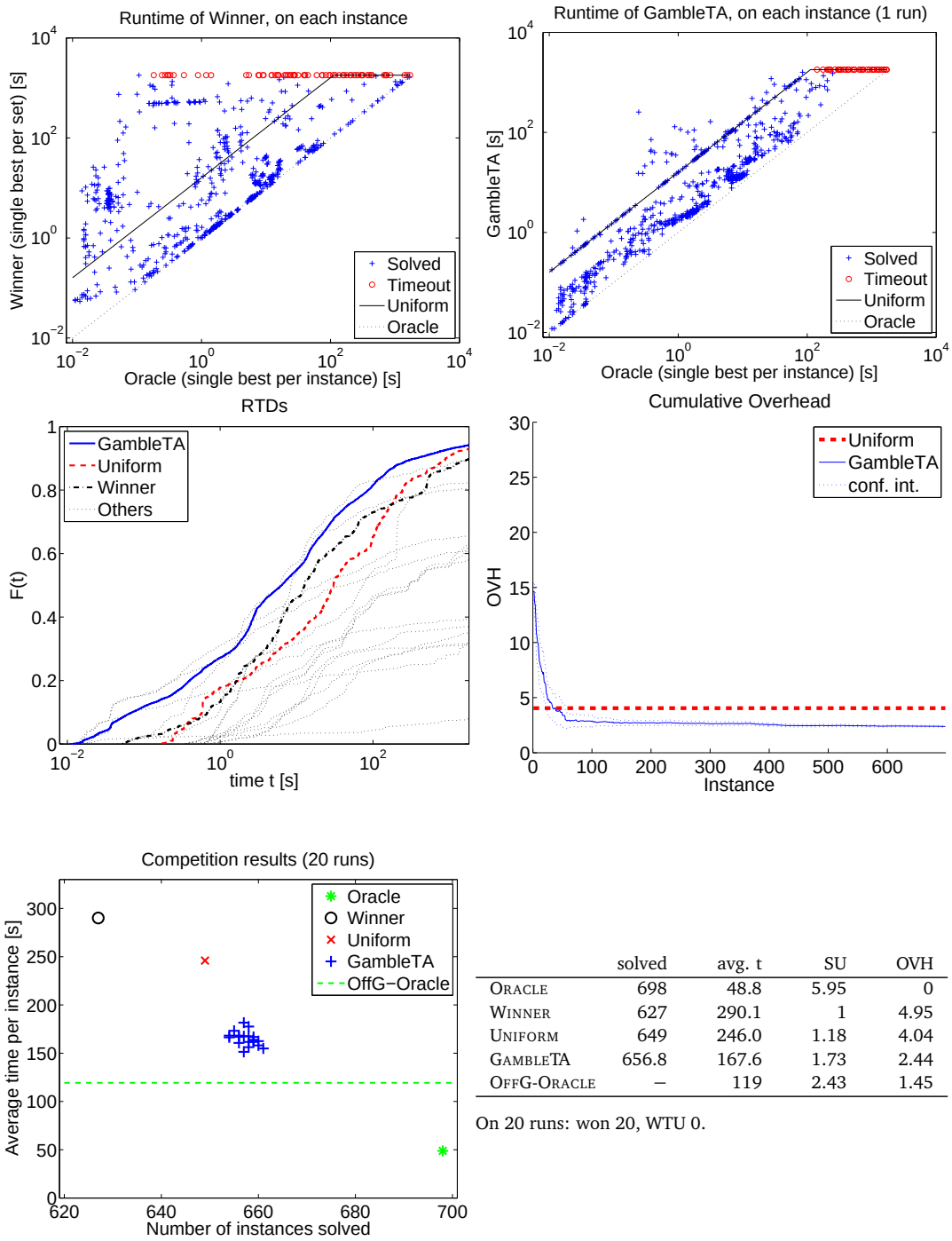
In Tables 9.1–9.8 we report the speedup obtained by GAMBLETA on each of the 43 competitions, comparing with ORACLE, UNIFORM, OFFG-ORACLE (and ONG-EXP3 when available).



	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.

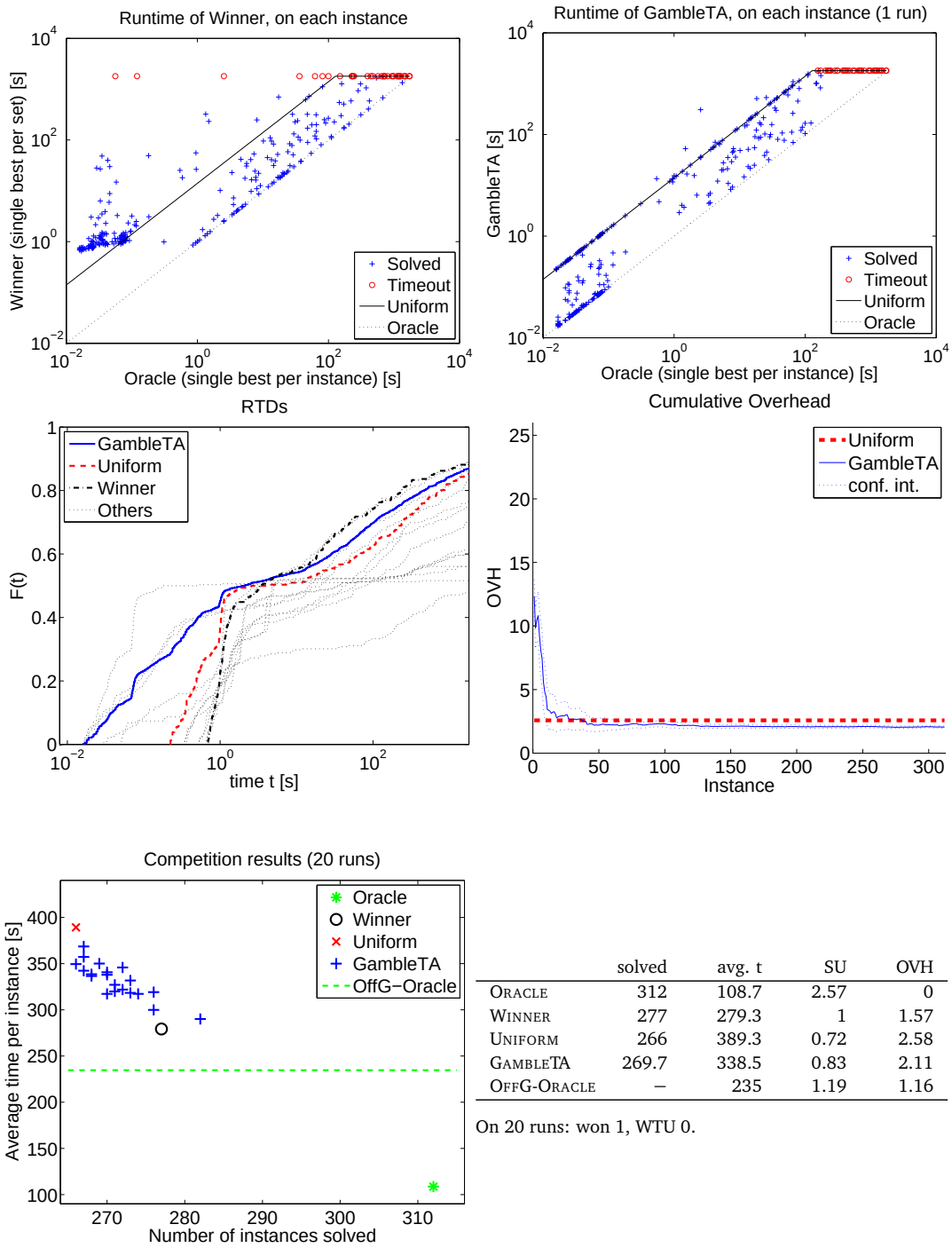
Figure 9.21. CPAI'06, Binary ext., 15 algorithms, 1140 instances, timeout 1800 s. Winner: `VALCSP3..`



	solved	avg. t	SU	OVH
Oracle	698	48.8	5.95	0
Winner	627	290.1	1	4.95
Uniform	649	246.0	1.18	4.04
GambleTA	656.8	167.6	1.73	2.44
OffG-Oracle	–	119	2.43	1.45

On 20 runs: won 20, WTU 0.

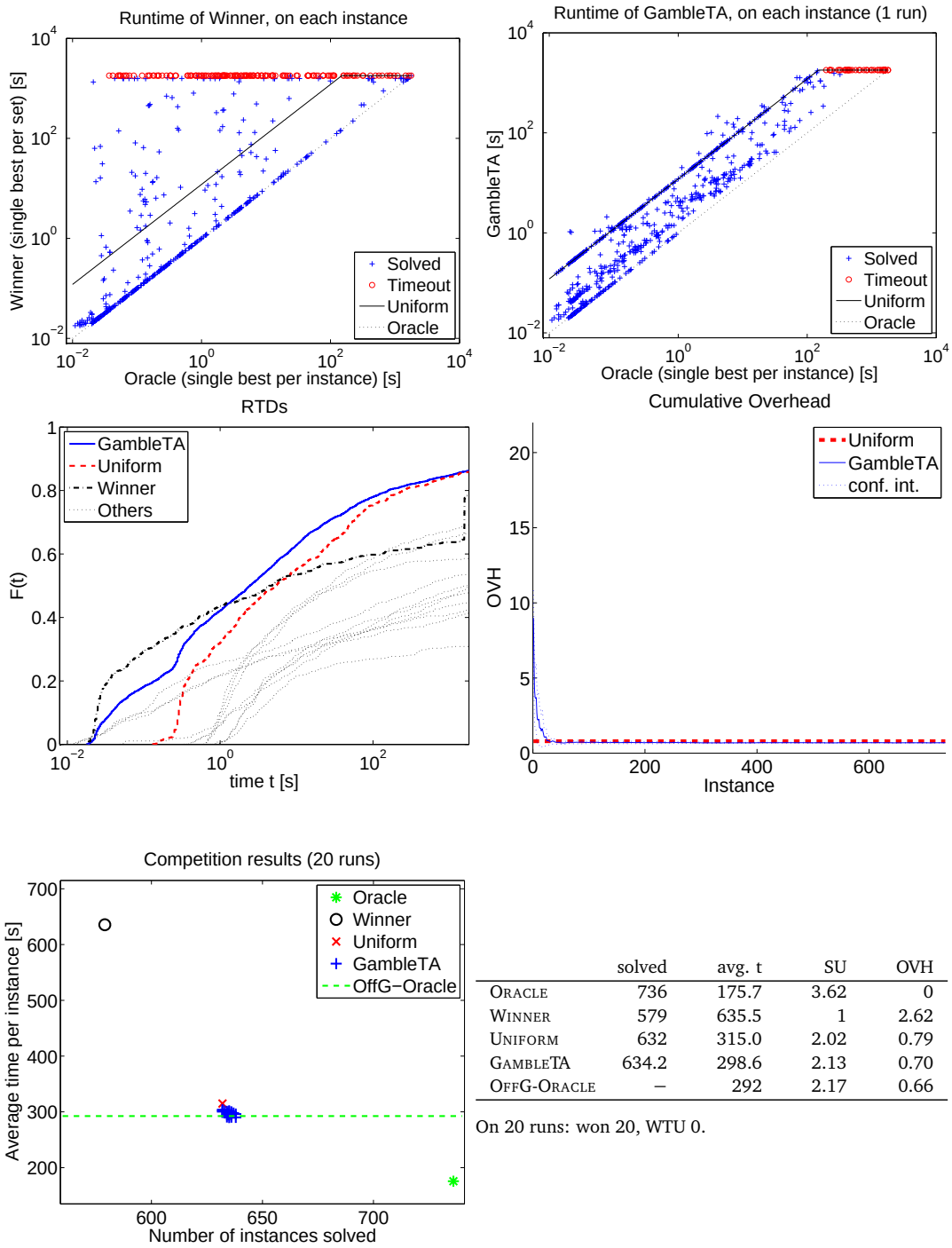
Figure 9.22. CPAI'06, Binary int., 16 algorithms, 698 instances, timeout 1800 s. Winner: buggy-. Uniform would have won.



	solved	avg. t	SU	OVH
Oracle	312	108.7	2.57	0
Winner	277	279.3	1	1.57
Uniform	266	389.3	0.72	2.58
GambleTA	269.7	338.5	0.83	2.11
OffG-Oracle	–	235	1.19	1.16

On 20 runs: won 1, WTU 0.

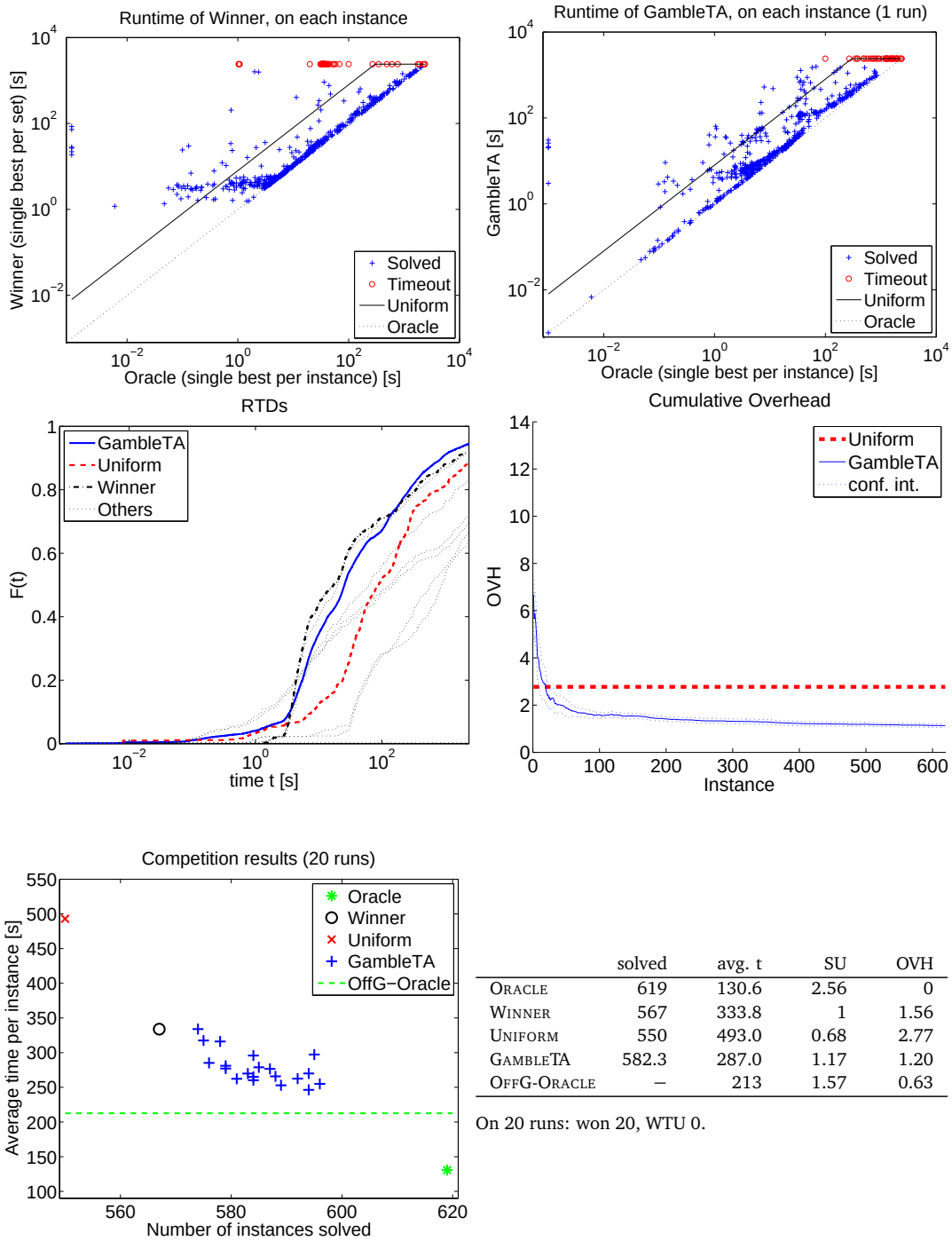
Figure 9.23. CPAI'06, N-ary ext., 14 algorithms, 312 instances, timeout 1800 s. Winner: `Abscon1`.



	solved	avg. t	SU	OVH
Oracle	736	175.7	3.62	0
Winner	579	635.5	1	2.62
Uniform	632	315.0	2.02	0.79
GambleTA	634.2	298.6	2.13	0.70
OffG-Oracle	–	292	2.17	0.66

On 20 runs: won 20, WTU 0.

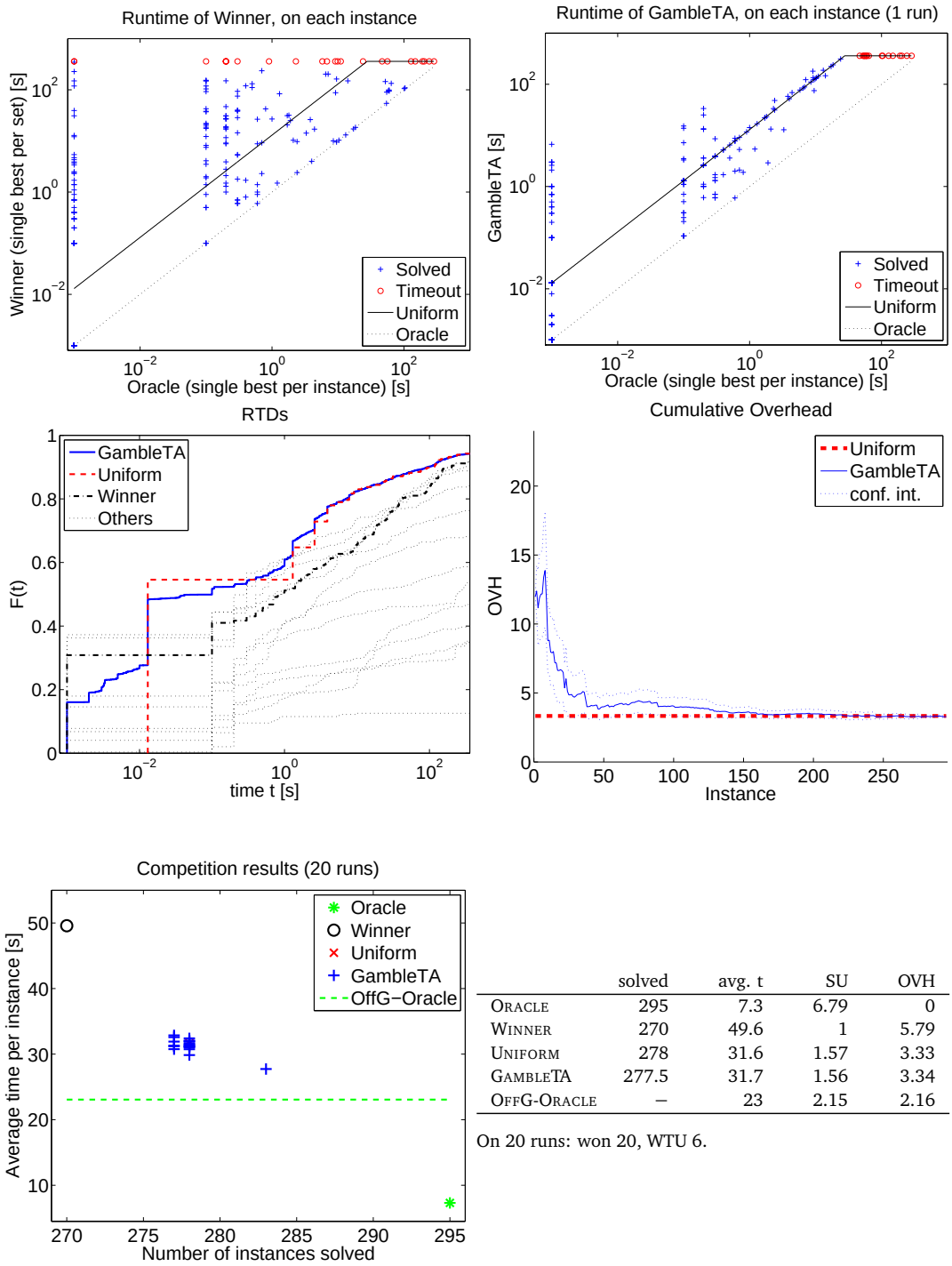
Figure 9.24. CPAI'06, N-ary int., 12 algorithms, 736 instances, timeout 1800 s. Winner: `BPrologCSPSolver7`. Uniform would have won.



	solved	avg. t	SU	OVH
Oracle	619	130.6	2.56	0
Winner	567	333.8	1	1.56
Uniform	550	493.0	0.68	2.77
GambleTA	582.3	287.0	1.17	1.20
OffG-Oracle	–	213	1.57	0.63

On 20 runs: won 20, WTU 0.

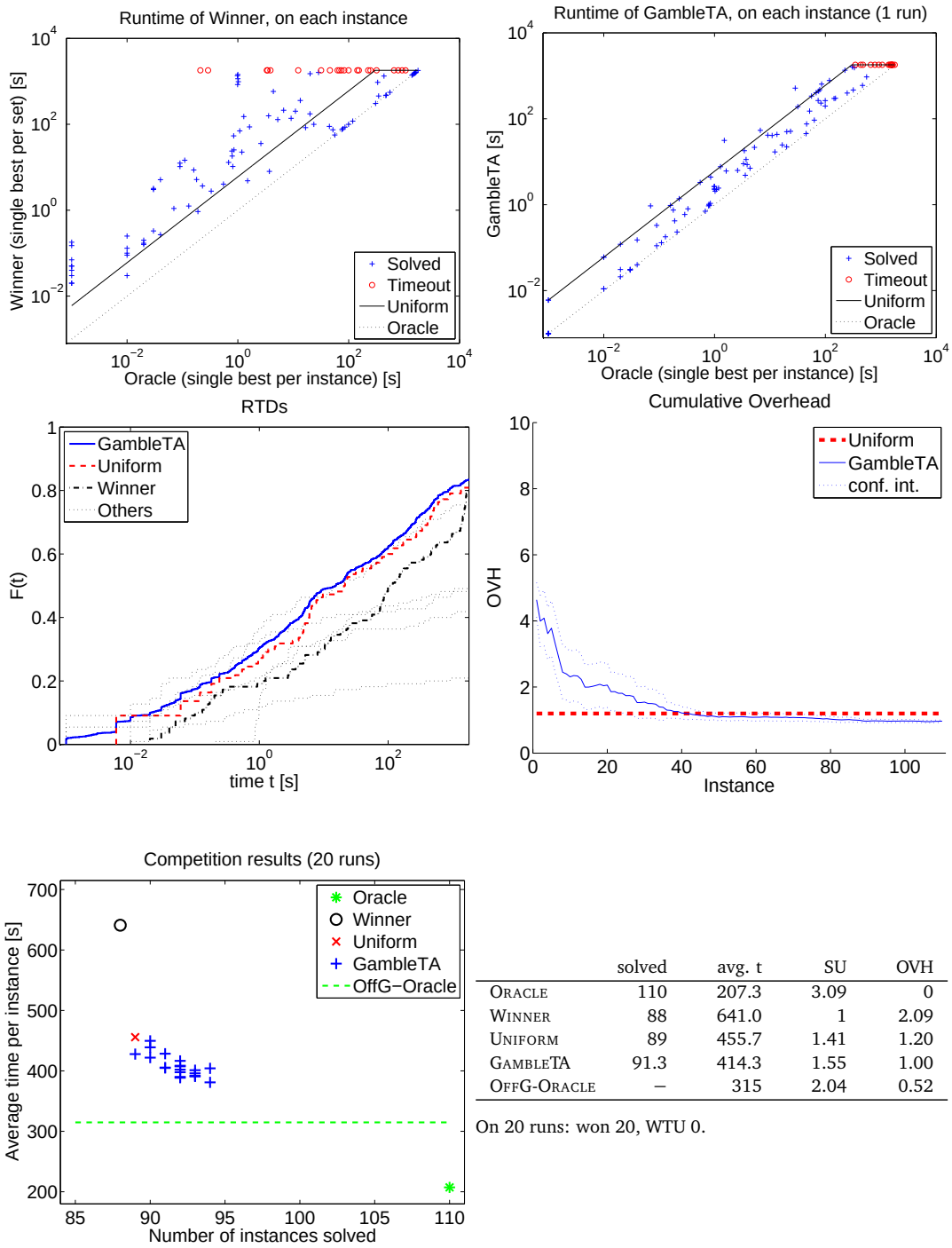
Figure 9.25. CPAI'06, Opt. binary ext., 8 algorithms, 619 instances, timeout 2400 s. Winner: `Toolbar-BTD`.



	solved	avg. t	SU	OVH
Oracle	295	7.3	6.79	0
Winner	270	49.6	1	5.79
Uniform	278	31.6	1.57	3.33
GambleTA	277.5	31.7	1.56	3.34
OffG-Oracle	–	23	2.15	2.16

On 20 runs: won 20, WTU 6.

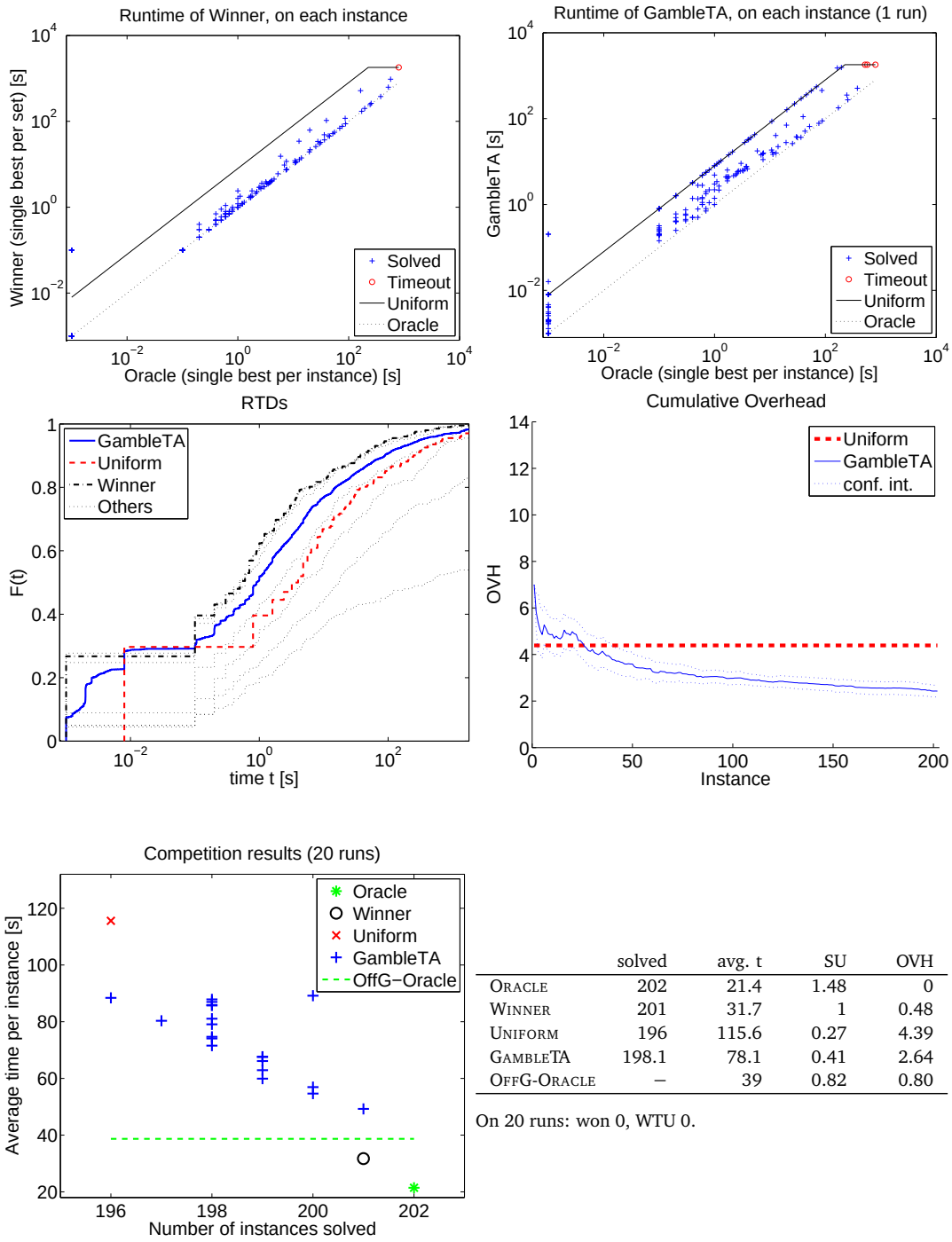
Figure 9.26. CASC-J3, FOF, 13 algorithms, 295 instances, timeout 360 s. Winner: `Vampire-9.`. Uniform would have won.



	solved	avg. t	SU	OVH
Oracle	110	207.3	3.09	0
Winner	88	641.0	1	2.09
Uniform	89	455.7	1.41	1.20
GambleTA	91.3	414.3	1.55	1.00
OffG-Oracle	–	315	2.04	0.52

On 20 runs: won 20, WTU 0.

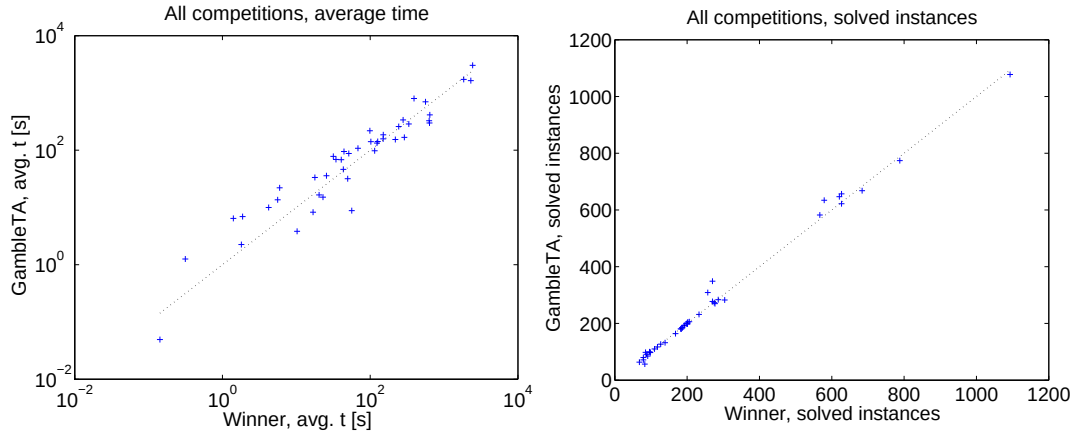
Figure 9.27. IPC-5, Optimal planning, 6 algorithms, 110 instances, timeout 1800 s. Winner: `maxplan`. Uniform would have won.

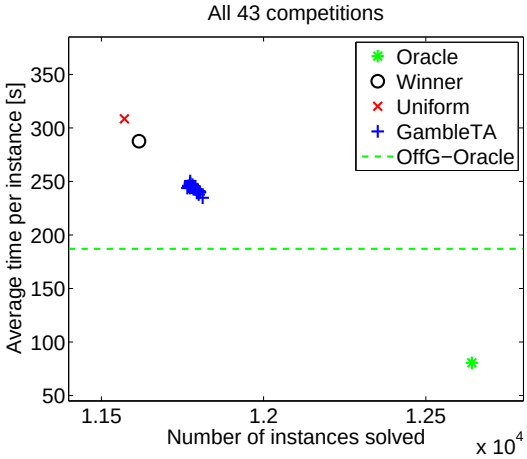


	solved	avg. t	SU	OVH
Oracle	202	21.4	1.48	0
Winner	201	31.7	1	0.48
Uniform	196	115.6	0.27	4.39
GambleTA	198.1	78.1	0.41	2.64
OffG-Oracle	–	39	0.82	0.80

On 20 runs: won 0, WTU 0.

Figure 9.28. SMT'07, QF LRA, 8 algorithms, 202 instances, timeout 1800 s. Winner: `Yices+1`.





	solved	avg. t	SU	OVH
ORACLE	12642	80.6	3.57	0
WINNER	11616	287.6	1	2.57
UNIFORM	11571	308.5	0.93	2.83
GAMBLETA	11781.5	244.5	1.18	2.03
OFFG-ORACLE	–	187.1	1.54	1.32

UNIFORM: won 12/43.
GAMBLETA: won 17.17/43, WTU 2.58/43.

Figure 9.29. All 43 competitions. Top: average time (left) and number of instances solved (right) by GAMBLETA (vertical axis), compared with WINNER (horizontal axis). In these plots, each + sign corresponds to a competition. Results for GAMBLETA are confidence bounds evaluated on 20 runs (upper for average time, lower for instances). Bottom: overall performance on all 43 competitions, considered as a single one with 12642 instances. UNIFORM is already competitive with WINNER, but GAMBLETA improves further.

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-ORACLE
CNF	10	191	3.70	0.69	1.04	1.45
EPR	10	98	1.88	0.19	0.32	0.56
FNT	6	100	10.98	6.05	6.50	3.47
FOF	13	295	6.79	1.57	1.61	2.15
SAT	7	100	6.95	0.99	2.36	5.49
UEQ	7	93	1.19	0.65	0.73	0.99

Table 9.1. Speedups for CADE 2007

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-ORACLE
Binary ext.	15	1140	3.23	0.67	0.94	1.37
Binary int.	16	698	4.82	0.96	1.45	1.97
Global	13	127	2.76	0.21	0.27	0.28
Opt. binary ext.	8	619	2.56	0.68	1.24	1.57
Opt. n-ary ext.	8	97	2.65	0.61	0.87	1.23
N-ary ext.	14	312	2.57	0.72	0.86	1.19
N-ary int.	12	736	3.50	1.95	2.08	2.10

Table 9.2. Speedups for CP 2006

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-O
Optimal planning	6	110	2.44	1.11	1.26	1.61

Table 9.3. Speedups for IPC 2006

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-ORACLE
Max-SAT	13	790	1.26	0.21	0.55	0.98
Partial MS	13	647	3.81	0.60	0.98	1.31
Weighted MS	13	308	1.61	0.30	0.52	0.82
Weight. Part.	13	702	2.36	0.40	0.78	1.15

Table 9.4. Speedups for MaxSAT 2007

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-O
Opt. big ints.	7	124	2.34	0.81	0.99	1.05
Opt. small ints.	16	396	7.40	1.65	2.02	2.71
Opt. sm. ints. nl.	16	280	4.18	0.48	0.68	2.10
Pure SAT	16	88	1.80	0.38	0.53	0.98
SAT/UNS sm. ints. nl.	16	216	3.98	0.65	1.65	2.56

Table 9.5. Speedups for PB 2007

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-O
Formal verification	16	728	5.20	1.02	1.21	1.52
Horn clause forms.	16	287	2.14	0.19	0.45	1.06
Non prenex non cnf	12	81	2.30	0.42	0.48	0.81
Planning	16	80	4.12	0.69	0.85	1.28

Table 9.6. Speedups for QBF 2007

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-O	ONG-E
AIG	5	263	2.49	0.82	0.96	1.11	–
Crafted	9	129	3.42	1.00	1.11	1.37	0.91
Industrial	10	166	3.48	0.77	0.83	0.99	0.88
Random	14	411	3.99	1.22	1.35	1.61	1.05

Table 9.7. Speedups for SAT 2007

Category	n.algs.	n.insts.	ORACLE	UNIFORM	GAMBLETA	OFFG-ORACLE
AUFLIA	5	192	11.65	2.33	2.83	2.64
AUFLIRA	5	193	16.31	3.26	3.98	15.10
QF AUFBV	3	187	1.00	0.46	0.56	1.00
QF AUFLIA	4	206	2.91	0.73	0.89	1.05
QF BV	5	200	3.41	0.68	1.34	1.97
QF IDL	6	186	1.46	0.54	0.63	1.00
QF LIA	6	186	1.02	0.18	0.35	0.95
QF LRA	8	202	1.48	0.27	0.48	0.82
QF RDL	6	168	2.39	0.47	0.67	0.70
QF UF	6	199	7.13	1.19	1.63	2.29
QF UFIDL	5	201	1.20	0.24	0.48	0.85
QF UFLIA	6	110	1.09	0.18	0.24	0.25

Table 9.8. Speedups for SMT 2007

9.4 Combinatorial auctions

The Auction Winner Determination Problem (WDP) Leyton-Brown et al. [2002] is an interesting combinatorial optimization problem, where a set of agents allocate money on n bids over m goods, and the winning subset of bids, that maximizes the sum of the amounts bidden, must be determined. The agents have limited amounts of money, and are allowed to specify XOR constraints over the bidden goods, and the selected winning subset has also to satisfy these constraints. The problem is NP-hard.

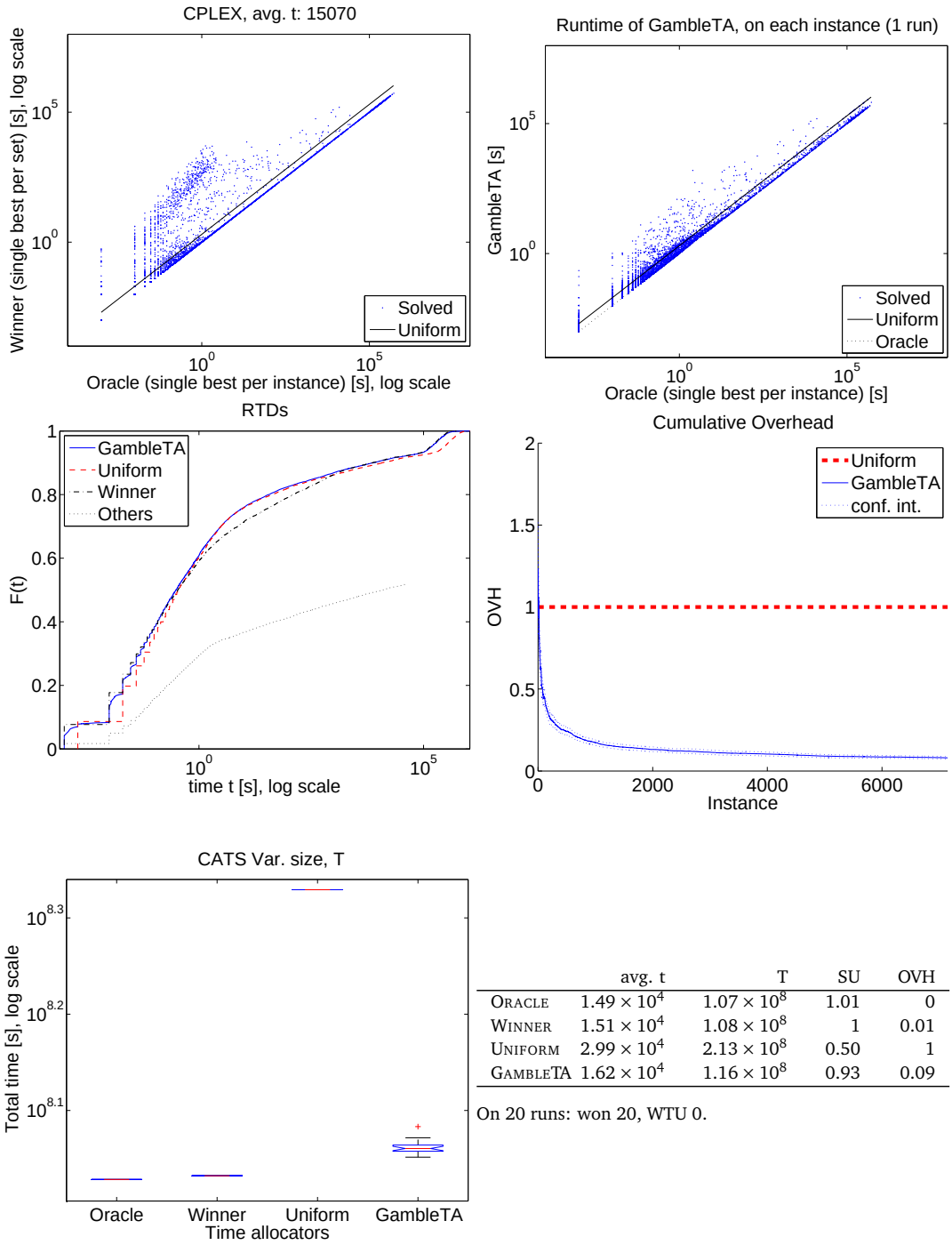
In [Leyton-Brown et al., 2002], to which we refer for more details and references, the hardness of randomly generated WDP instances is modeled, describing the performance of a Linear Programming software (CPLEX), and an ad-hoc solver (CASS). The runtime of these solvers is related to 28 instance features, including the size (n,m), and serves as target for a regression routine aimed at learning a predictive model of runtime value, conditioned on instance features. The performance of the models is assessed using the mean squared error on the logarithm of predicted values, which implies a parametric assumption of the run-time distribution being log-normal. Censored runtimes ("capped" runs in the terminology of the paper) are considered as the uncensored, and it is argued that the impact of this approximation on model precision is low. The resulting models are indeed quite precise in terms of the proposed error measure. The performance of CPLEX dominates CASS, but on about 1/4th of the instances this situation is inverted. In such a case, a per set selection technique would always select CPLEX. As an interesting example application of these models, the authors propose a per instance algorithm selection technique, in which the expected fastest algorithm is picked based on the model's predictions. In the original paper, the model is trained on runtime data obtained by solving a large number of instances, censoring runs that exceed a predetermined threshold of 12 hours for CASS. On a test set of unseen instances, the model performs efficient selection, detecting the instances on which CASS is faster, and allowing the portfolio to improve on the performance of CPLEX alone. The overhead (9.3) compared to the performance of the oracle, is reported to be 8%, excluding a small additional factor due to the cost of computing features.

The runtime data for the two algorithms were obtained online[6]. We report results for the largest set, with variable instance size. After discarding a few instances, for which the time values were censored for both algorithms, the set has 7145 instances. The runtimes in the data set sum to almost nine years.

Figure 9.30 reports results, using box-plots[7] to represent the cumulative time. Note that also in this case we only have one run for each algorithm and instance, so results for UNIFORM, WINNER and ORACLE consists of a single number, while results for GAMBLETA refer to 20 runs as usual. The WINNER in this case is `CPLEX`, which dominates the other algorithm in performance. The overhead is reduced quickly and drops to a final 9%. Most of the overhead is due to the initial portion of the instance sequence: limiting to the second half of the portion, the overhead would be 6%. Leyton-Brown et al. [2002] report that their algorithm selection method obtain an overhead of 8% on a test set of 15% of the instances, after training based on results of both algorithms on the other 85% of the instances.

[6] `http://ws.cs.ubc.ca/~kevinlb/dl.php?u=data.zip`

[7] See note 3 at page 122.



	avg. t	T	SU	OVH
ORACLE	1.49×10^4	1.07×10^8	1.01	0
WINNER	1.51×10^4	1.08×10^8	1	0.01
UNIFORM	2.99×10^4	2.13×10^8	0.50	1
GAMBLETA	1.62×10^4	1.16×10^8	0.93	0.09

On 20 runs: won 20, WTU 0.

Figure 9.30. Results for the CATS benchmark, 2 algorithms, 7145 instances. WINNER: CPLEX. In this case WINNER dominates. Performance of GAMBLETA is comparable. Limiting to the second portion of the sequence, the overhead drops down to 0.06.

9.5 SAT/UNSAT

In this benchmark, the algorithm set consists of two solvers for the SAT problem (Sec. 2.1.1). One is the complete solver Satz-Rand [Gomes et al., 2000], a randomized version of Satz [Li and Anbulagan, 1997] in which random noise influences the choice of the branching variable. Satz is a modified version of the complete DPLL procedure, in which the choice of the variable on which to branch next follows an heuristic ordering, based on first and second level unit propagation. Satz-Rand differs in that, after the list is formed, the next variable to branch on is randomly picked among the top h fraction of the list. All experiments were performed with the heuristic starting from the most constrained variables, as suggested also in [Li and Anbulagan, 1997], and the noise parameter set to 0.4. The second one is the local search solver G2-WSAT [Li and Huang, 2005]. For this algorithm, we set a high noise parameter (0.5), as advisable for problems at the phase threshold, and the diversification probability at the default 0.05.

As a benchmark for this set, we will use the complete set of `uf-n-m` and `uuf-n-m` instances from SATLIB [Hoos and Stützle, 2000]. These are randomly generated instances at the phase transition, with n ranging from 20 (resp. 50 for the unsat) to 250, and m varying accordingly. The instances are subdivided in groups of satisfiable (`uf*`) and unsatisfiable (`uuf*`) instances, there are 100 instances for each size, for a total of 1899 instances[8] in total.

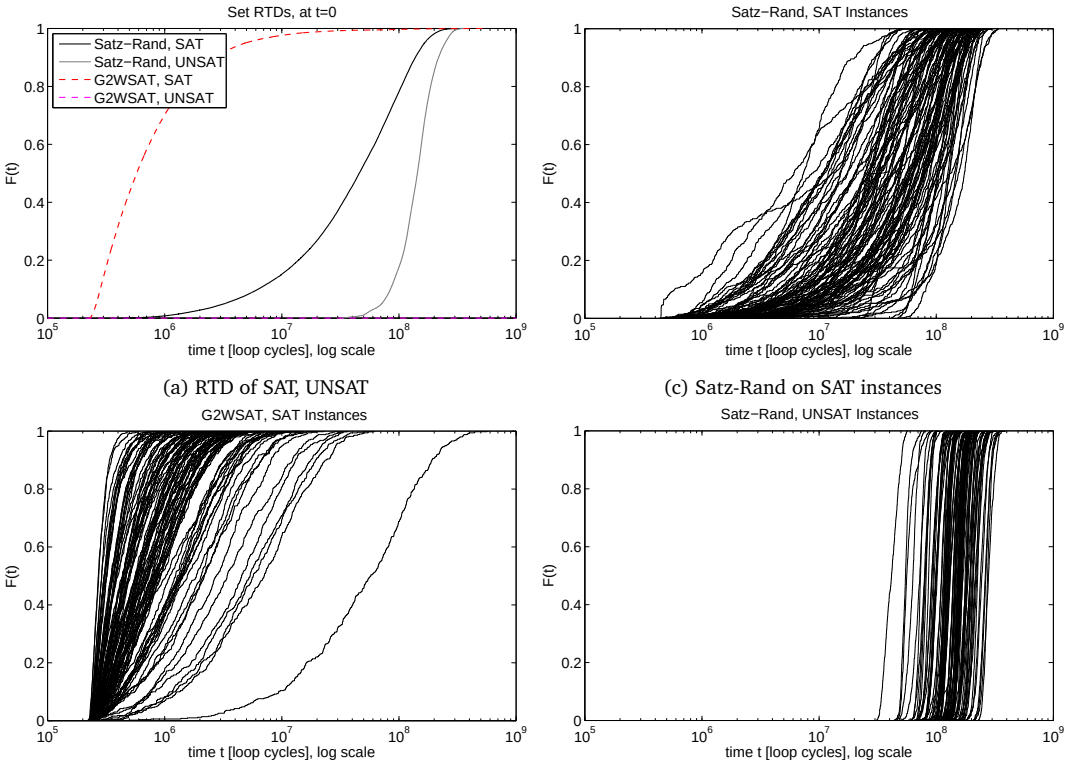
As we needed a common measure of time, and the CPU runtime measures are quite inaccurate [see also Hoos and Stützle, 2004, p. 169], we modified the original code of the two algorithms adding a counter, that is incremented at every loop in the code. The resulting time measure was consistent with the number of backtracks, for Satz-Rand, and the number of flips, for G2-WSAT. All runtimes reported for this benchmark are expressed in these loop cycles: on a 2.4 GHz machine, 10^9 cycles take about 1 minute.

From the point of view of the runtimes involved, this is clearly a "toy" benchmark: `Satz-Rand` can solve the whole set of instances in less than an hour. Nonetheless, this algorithm set/problem set combination poses an interesting time allocation problem, as G2-WSAT dominates the performance of Satz-Rand on satisfiable instances, while the latter is obviously the winner on all unsatisfiable ones, on which the runtime of G2-WSAT is infinite. As discussed already in Section 6.1.1, what makes the problem interesting is that the satisfiability of an instance cannot be inferred based on features only.

Let us now look with more detail at the variability of the RTDs within the set. Figure 9.31 (a) displays the RTDs of the two algorithms on the subsets of SAT and UNSAT instances of size 250. Figures 9.31 (c,d,e) display the RTDs of the instances, again estimated based on 100 runs for each instance, grouped based on the algorithm and on satisfiability. Note that there still is a huge variability of the RTD of the instances within each subset.

Figure 9.32 reports the performance of GAMBLETA, which is quite satisfactory in this case, obtaining an overhead of 20% over ORACLE.

[8] This odd number is due to the fact that instance `uuf-200-860` number 100 is missing in the online archive. Note also that the smallest n for the unsatisfiable instances is 50, so there are 1000 SAT and 899 UNSAT instances in total, making the SAT probability for the whole set slightly higher than 0.5.



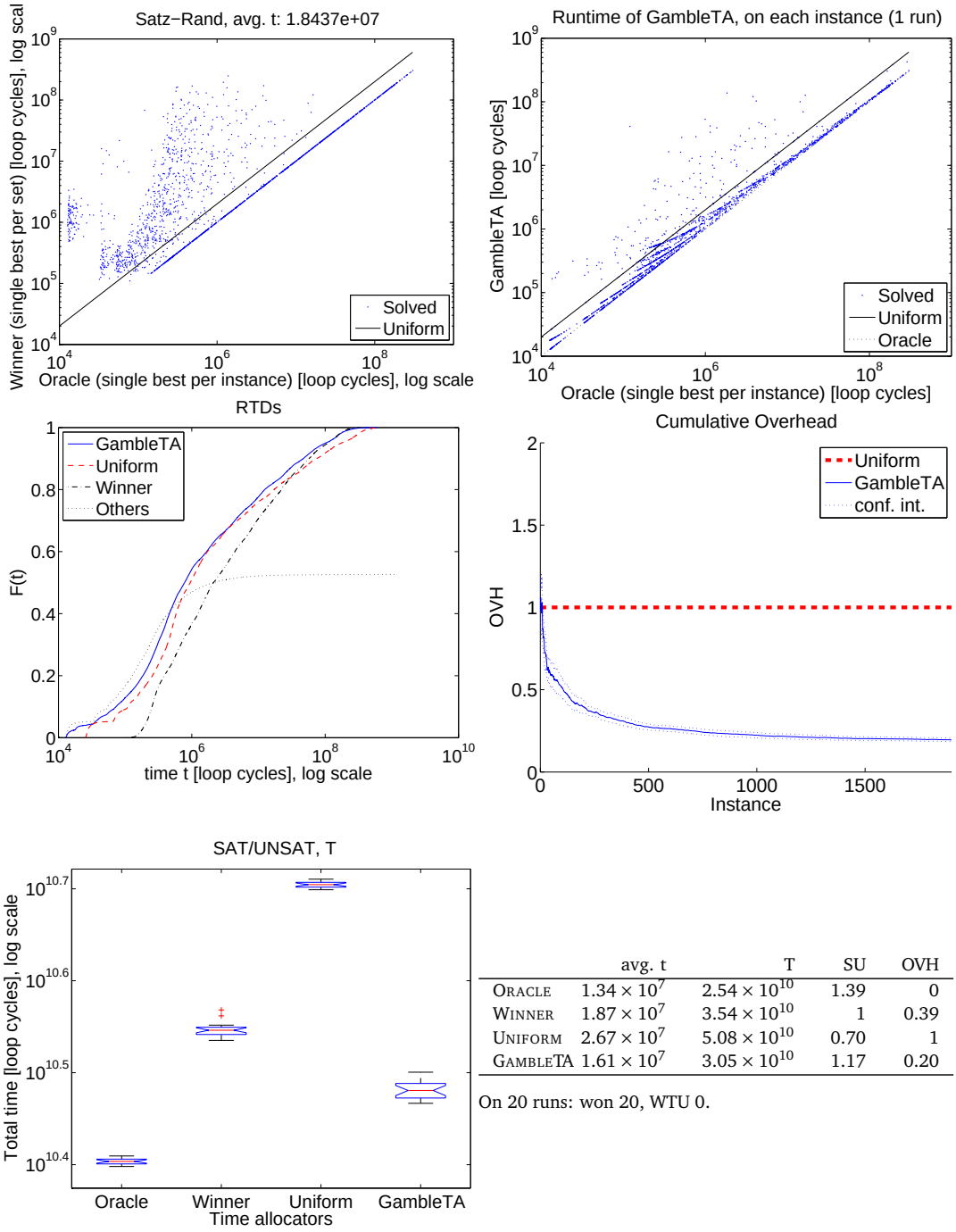
(a) RTD of SAT, UNSAT

(c) Satz-Rand on SAT instances

(b) G2WSAT on SAT instances

(d) Satz-Rand on UNSAT instances

Figure 9.31. (a) RTDs of the two algorithms on the subsets of SAT and UNSAT instances: the line for G2WSAT on UNSAT instances would be constant at 0, and is omitted — (b) RTDs of G2WSAT on each of the 100 satisfiable instances. Note the different time scale. The lower line leaving the plot refers to instance 24, and reaches 1 at time 1.6×10^8 — (c,d) RTDs of Satz-Rand on the satisfiable and unsatisfiable instances, respectively



	avg. t	T	SU	OVH
ORACLE	1.34×10^7	2.54×10^{10}	1.39	0
WINNER	1.87×10^7	3.54×10^{10}	1	0.39
UNIFORM	2.67×10^7	5.08×10^{10}	0.70	1
GAMBLETA	1.61×10^7	3.05×10^{10}	1.17	0.20

On 20 runs: won 20, WTU 0.

Figure 9.32. Results for the SAT/UNSAT benchmark, 2 algorithms, 1899 instances. WINNER: `Satz-Rand`.

9.6 Multiple processors

The main objective of these preliminary experiments is to analyze the speedup (the ratio between runtime with 1 and $Z > 1$ CPUs) and efficiency (the ratio between speedup and number of CPUs) of the proposed allocation method. Note that the notion of efficiency assumes a different connotation in the context of algorithm portfolios: traditionally one does not expect to achieve an efficiency larger than 1, as it is assumed that all computations performed on a single CPU have to be carried out on Z CPUs as well. This is not the case for algorithm portfolios, as in this case we can stop the computation as soon as the fastest algorithm solves the problem, so we will see efficiencies greater than 1.

In the first experiment we apply GAMBLETA to the SAT/UNSAT benchmark (Sec. 9.5). In this case the set of time allocators includes the uniform one, and nine quantile allocators, with α ranging from 0.1 to 0.9.

In a second experiment we use `Satz-Rand` alone on the morphed graph-coloring benchmark (Sec. 8.1). In this case, the time allocators decide only how many parallel copies of the algorithm should be run for each problem: as the share is 1 on each CPU, we allowed the TAs to dynamically shrink and also grow the number of CPUs used.

Both experiments were repeated for different numbers of CPUs $(1, 5, 10, 15, 20)$. Results reported are upper confidence bounds obtained from 20 runs, each time using fresh random seeds, and a different random reordering of the problem instances.

The results of the experiments on the graph coloring benchmark show that, when `Satz-Rand` displays heavy tails, GAMBLETA can exploit this opportunity, obtaining also an efficiency much larger than 1. When no heavy tails are present, and on the SAT/UNSAT benchmark, the results are less convincing, as the efficiency is less than 1.

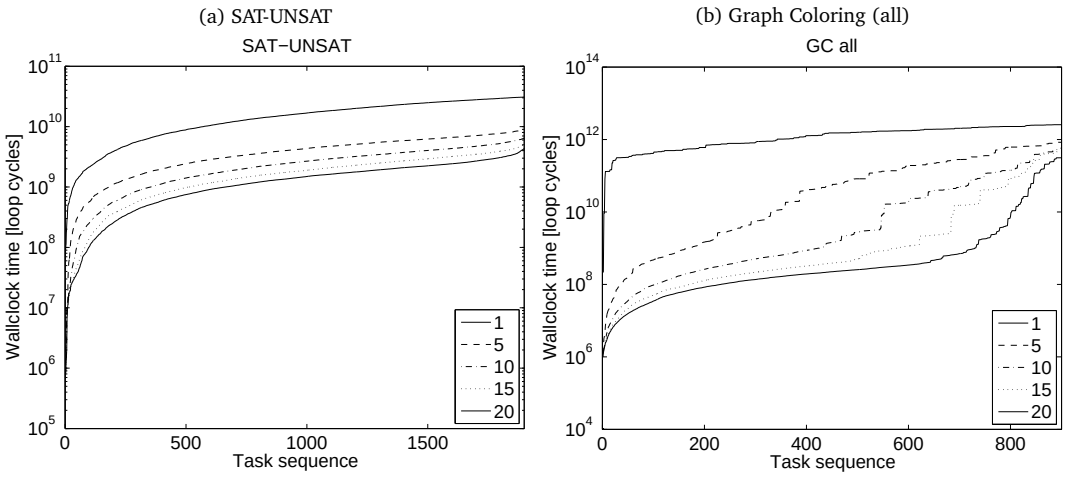


Figure 9.33. (a): Wall-clock time for the SAT-UNSAT benchmark, for different numbers of CPUs ($10^9 \approx 1$ min.). (b): Wall-clock time for the Graph Coloring benchmark (all problems), for different numbers of CPUs ($10^9 \approx 1$ min.).

	Speedup				Efficiency			
#CPUs	5	10	15	20	5	10	15	20
GC 0	4.34	7.19	9.23	11.19	0.87	0.72	0.62	0.56
GC 1	1.00	279.26	83.55	85.87	0.20	27.93	5.57	4.29
GC 2	2.65	18.44	35.98	97.74	0.53	1.84	2.40	4.89
GC 3	2.59	6.55	9.80	21.28	0.52	0.66	0.65	1.06
GC 4	3.97	6.18	6.94	8.81	0.79	0.62	0.46	0.44
GC 5	3.36	3.47	7.69	7.83	0.67	0.35	0.51	0.39
GC 6	3.55	5.13	5.79	8.54	0.71	0.51	0.39	0.43
GC 7	5.93	8.27	12.65	11.95	1.19	0.83	0.84	0.60
GC 8	5.06	9.02	11.62	12.01	1.01	0.90	0.77	0.60
GC all	3.06	4.46	5.50	8.22	0.61	0.45	0.37	0.41
SAT-UNSAT	3.46	4.81	6.02	7.08	0.69	0.48	0.40	0.35

Table 9.9. Speedup (on the left) and efficiency (on the right) for the SAT-UNSAT and the different subgroups of Graph Coloring (GC) benchmarks. Note the dramatic speed-up obtained on GC sets 1 and 2, the effect of the heavy-tailed RTD of Satz-Rand on these problem sets.

9.7 Deletion experiments

In this section we report the results of several deletion experiments, in which a single aspect of GAMBLETA is changed, in order to evaluate its impact on performance. These results were obtained on the SAT/UNSAT benchmark, and are reported in Table 9.10. Each line in the table corresponds to a set of 20 repetition of the same experiment, with different random seeds, and different random reordering of the instances, but such that the same sequence of instances and runtimes are observed in each experiments: in this way the difference in the results depends only on the corresponding deletions.

The table is divided in three blocks, the last one reporting the usual comparison terms. The first block contains experiments which can be implemented in practice: the first line reports the baseline performance of GAMBLETA (Sec. 9.5). The second line (CORRECT) corresponds to the more correct RTD estimation method proposed in Section 6.1.2, which takes into account the satisfiability of previously solved instances and estimates the RTDs on the current instance using a mixture. In this case the performance degrades slightly, rather than improving as we would have expected.

The following line (STATIC) reports results obtained suppressing the dynamic updates of the share (Sec. 5.4). For these experiments, we mixed the shares evaluated by the time allocators with the uniform share, in order to satisfy the hypothesis that each instances would eventually be solved[9]. Here the impact on performance was dramatic, and negative: GAMBLETA with static allocation becomes similar to UNIFORM.

The second block of experiments was aimed at studying the impact of the incompleteness of information faced by GAMBLETA, and could only be implemented as we were working with a pre-collected runtime sample: in this case the results do not account for the additional runtime that would have been spent to perform the experiments in a realistic situation.

In the UNCENSORED experiment, when an instance is solved by one of the algorithm, also the runtime of the remaining algorithms is revealed exactly, with no censoring. The aim of this experiment was to study the impact of the bias induced by competing risks (Sec. 6.3). The result is practically the same as the original GAMBLETA, indicating that the impact of competing risks is negligible in this case.

In the FORESIGHT experiment, all runtimes are revealed beforehand, and the model is updated before beginning the instance sequence. Note that such data would allow to implement ORACLE, and obtain the best possible performance: the aim of this experiment is rather to compare, based on allocation performance, the RTD models obtained with online learning and censoring, to those that could be obtained knowing the uncensored runtime samples in advance. In this case we do see a marginal improvement over the baseline performance. This result can be better understood if we look at the evolution of cumulative overhead (Fig. 9.32), which decreases rapidly during the first few instances. This seems to suggest that the small advantage of such foresight derives from these initial instances, after which the RTD models become competitive.

In the following experiment (INSTANCE RTD), we simulated the performance of the ideal allocators of Section 5.2: rather than using regression models, the instance RTDs were estimated directly from a sample of 100 uncensored runtimes. In this case the improvement in performance is more evident, as we are using a very good approximation of the actual RTD. Note that in this as in the previous two experiments, an actual implementation would obviously be much worse

[9]Time was allocated according to the share $(0.9\mathbf{s} + 0.1\mathbf{s}_{\mathcal{U}})$, where $\mathbf{s}$ is the output of the TA, and $\mathbf{s}_{\mathcal{U}}$ is the uniform share. Note that this already increments the overhead by 0.1, but the resulting difference in performance is much more relevant.

than UNIFORM: each instance should be solved N times for UNCENSORED and FORESIGHT, and $100N$ times for Instance RTD. GAMBLETA obtains comparable results solving each instance only once.

Excluding INSTANCE RTD, for which we did not have enough data, the other deletion experiments were performed also for some of the competitions, with similar results, except that in some cases also CORRECT and UNCENSORED improved slightly over GAMBLETA. STATIC was systematically worse.

9.8 Bandit problem solver performance

In this section we study the behavior of the BPS, on the SAT/UNSAT benchmark (Sec. 9.5), as well as on some of the competitions (Sec. 9.3).

Figure 9.34 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.

To further investigate the impact of each allocator, in Figures 9.35 and 9.36 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 (Sec. 9.1). 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. As in the previous section, also in this case 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[10].

[10]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 more likely be another model-based allocator.

TA	T ($\times 10^{10}$)	SU	OVH
GambleTA	3.05	1.17	0.20
Correct	3.12	1.15	0.23
Static	5.08	0.73	1.01
Uncensored	3.06	1.17	0.21
Foresight	2.99	1.20	0.18
Instance RTD	2.80	1.28	0.11
Oracle	2.54	1.39	0
Winner	3.54	1	0.39
Uniform	5.08	0.70	1

Table 9.10. Results from various deletion experiments with GambleTA on the SAT/UNSAT benchmark: each column reports the cumulative time (T), speed-up (SU) compared to the per set best algorithm (in this case `Satz-Rand`), and overhead (OVH) compared to Oracle. The line GambleTA reports the baseline performance (Sec. 9.5), and is listed as a comparison term, as the lines for Oracle, Winner and Uniform. The remaining lines report results for variations of GambleTA. The first two can be implemented. Correct: the RTD is estimated accounting for the satisfiability of solved instances, as described in Section 6.1.2. Static: no dynamic update is performed. The following three would require some form of foresight of runtimes. Uncensored: when an instance is solved, the exact runtimes of all algorithms are revealed. Foresight: uncensored runtimes are revealed in advance, for all instances. Instance RTD: the KM estimate from 100 runs is available in advance for each instance. Each line reports upper/lower confidence bounds estimated on 20 runs, with different random reordering of the instances. These experiments allow to conclude that dynamic allocation has an important impact on performance, while the correctness and precision of the RTD is less relevant, as only the instance RTD allows to sensibly improve the performance.

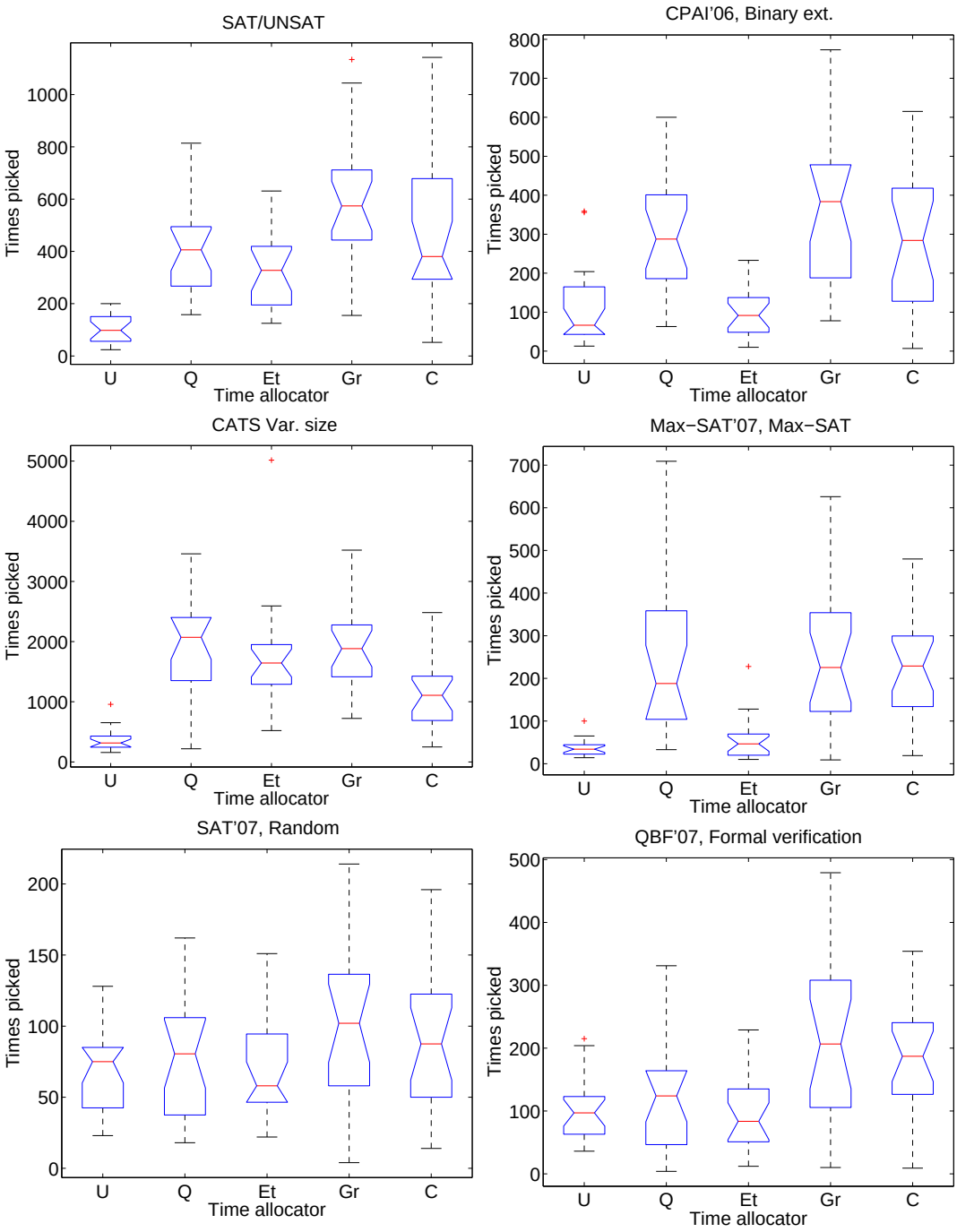


Figure 9.34. BPS: Number of pulls for each arm (number of instances solved with each TA, on 20 runs).

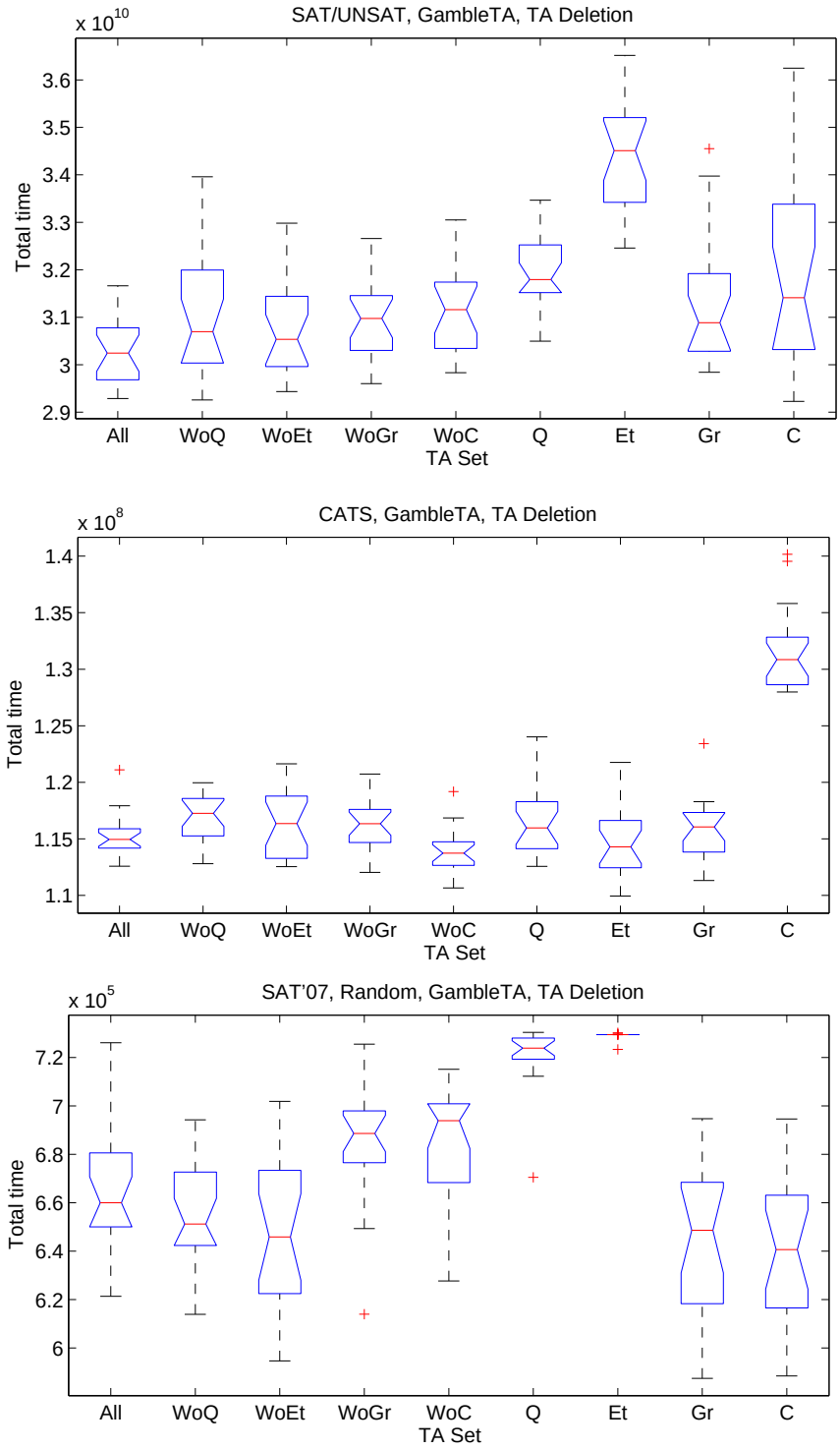


Figure 9.35. BPS: Deletion experiments with different TA sets (20 runs). All: GAMBLETA with 5 allocators. WoQ: without the quantile allocator. Q: only with UNIFORM and quantile allocators. Analogous for expected time (Et), greedy (Gr) and contract (C) allocators.

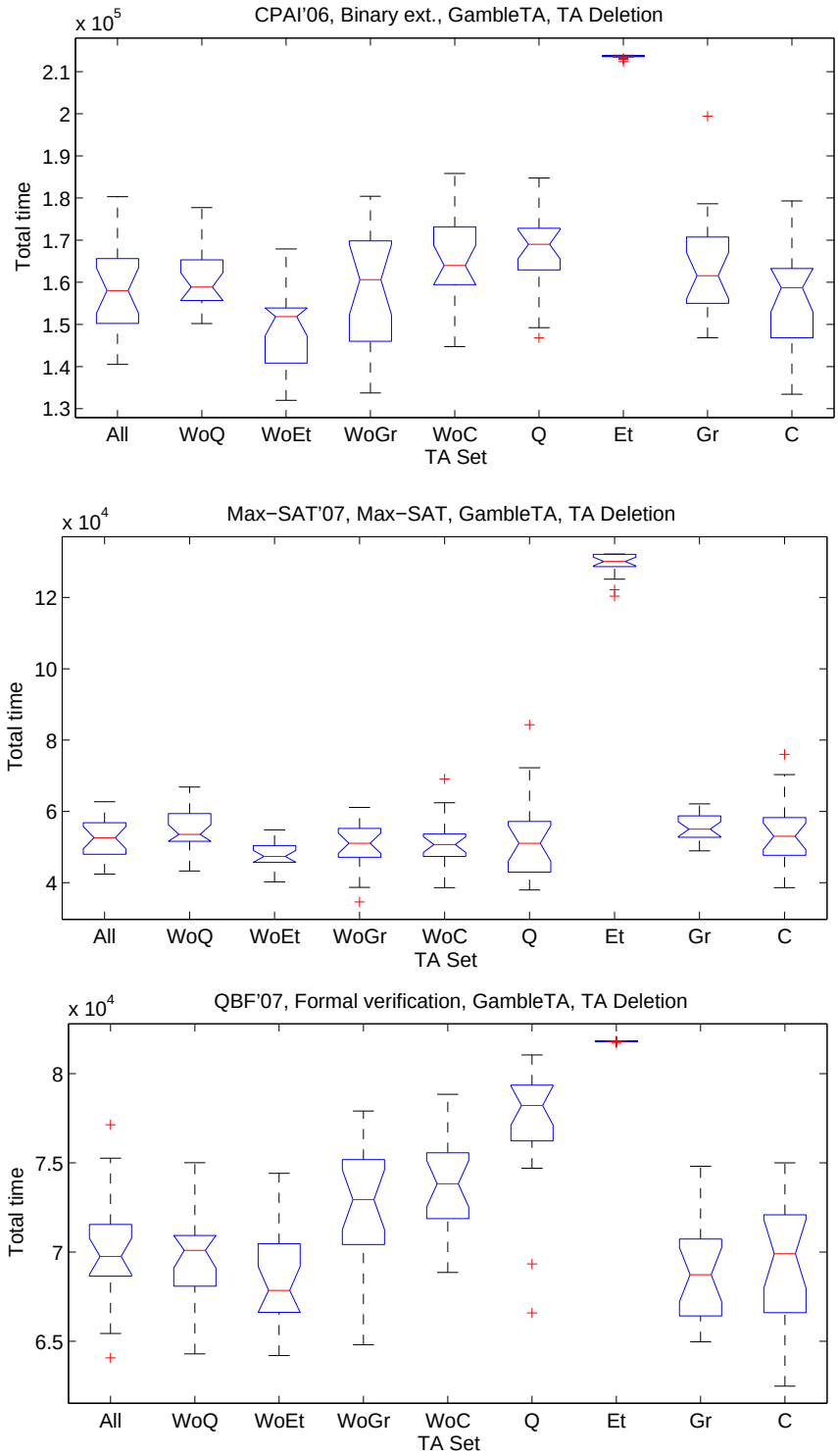


Figure 9.36. BPS: Deletion experiments with different TA sets (20 runs). All: GambleTA with 5 allocators. WoQ: without the quantile allocator. Q: only with Uniform and quantile allocators. Analogous for expected time (Et), greedy (Gr) and contract (C) allocators.

9.9 Discussion

This chapter presented experiments with GAMBLETA in several diverse time allocation scenarios. The standard comparison terms for all experiments were the UNIFORM portfolio of all algorithms, the single best per set algorithm (WINNER), and an ideal ORACLE, predicting and choosing the fastest algorithm independently for each instance.

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 summarize 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 on the CATS data, Fig. 9.30, and in some of the competitions, e.g. the Horn Clause formulas of QBFEVAL'07, Fig. 9.12, Max-SAT, Fig. 9.13), 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 the SAT/UNSAT benchmark (Fig. 9.32), as well as in many competitions, as SAT'07 Random (Fig. 9.6), QBFEVAL'07 Formal Verification (Fig. 9.11), PB'07 Optimization, small integer constraints (Fig. 9.18), several CPAI'06 categories (Figs. 9.21–9.24), etc. In these cases, GAMBLETA improves greatly over WINNER.

Moreover, excluding a few competitions with few instances, and small runtimes, the performance of GAMBLETA is consistently better than UNIFORM: GAMBLETA wins all the competitions where UNIFORM would have won, and some 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. As it can be seen from the evolution of cumulative overhead, this improvement is often 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 confirm our initial intuition that a rough RTD model already allows to allocate time efficiently.

The results of deletion experiments (Sec. 9.7) go in the same direction: relevant improvements in the model, as correcting for the correlation, map to marginal advantages in performance, or even disadvantages. Another important aspect is that the dynamic updates of the allocation are confirmed to be essential to performance.

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 (Sec. 9.8). 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 (see Sec. 9.1). The greedy allocator, based on [Streeter et al., 2007], 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.

Part V

Conclusion

Chapter 10

Conclusion

The variety in algorithm performance allows to speed up problem solving by combining multiple algorithms. This thesis contributed several practical examples of such speed-up. This closing chapter is organized as follows: Section 10.1 contains some concluding remarks. Section 10.2 summarizes our original contributions, referring to the corresponding publications. Section 10.3 sketches directions for future work.

10.1 Discussion

We introduced GAMBLETA, a novel framework for learning to perform time allocation online, and GAMBLER, a similar method for learning model-based restart strategies. Both frameworks are general and modular: they can be applied to arbitrary combinations of time allocators, and algorithms. Once these elements are chosen, no additional parameter setting is required.

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) [Hoos and Stützle, 2004], 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 both frameworks is to alternate a "default" way of allocating time, not requiring any prior knowledge on algorithm performance (an *oblivious* allocator) with one or more methods which learn to allocate time, based on runtimes observed so far (*non-oblivious* allocators). 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 in solving it represents the loss. The BPS used should deal with unbounded losses, as it is difficult to predict a maximum runtime. We contributed one such solver, EXP3LIGHT-A, proving a bound on its regret.

For GAMBLETA, the default allocator is the uniform portfolio (UNIFORM): given N algorithms, this allocator will be a factor N slower than the fastest algorithm. As we have shown in our

experiments, this trivial allocator already performs remarkably well, to the point that it would have won several competitions (Sec. 9.3). For GAMBLER, the default allocator is the universal strategy of Luby et al. [1993], whose performance is bounded w.r.t. the optimal strategy.

We also contributed example instantiations of the two frameworks, combining existing methods from the literature with our own research. We looked in particular at allocators based on the runtime distribution (RTD) of the algorithms, the most general performance model for gLVAs. Such methods include restart strategies and algorithm portfolios.

Restart strategies consist in a sequence of runs of the same randomized algorithm, with different random seeds, each run bounded by a restart threshold: when the runtime reaches the threshold, the current run is aborted, and the algorithm is restarted with a different random seed. The instance RTD of the algorithm allows to evaluate an optimal strategy [Luby et al., 1993].

Algorithm portfolios run several algorithms independently, on one or more processors, solving the same instance: once one algorithm solves it, the whole portfolio halts. Time is allocated according to a static or dynamic schedule. The RTD of the portfolio as a whole can be evaluated based on the schedule, and on the RTDs of the algorithms on the current instance. Based on this RTD, the schedule can be optimized, for example minimizing expected time [Finkelstein et al., 2002, 2003] and variance [Huberman et al., 1997; Gomes and Selman, 2001]. We considered minimization of expected time, along with two novel criteria, minimizing a quantile, and maximizing solution probability at a contract time.

During preliminary research on oblivious allocation [Gagliolo et al., 2004] we had realized the potential advantage of dynamic over static schedules, which was later proved by Sayag et al. [2006]. We therefore designed dynamic portfolios, updating the schedule at runtime, after conditioning the RTDs on the time already spent.

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. Such estimation can be carried out using regression models, conditioned on features of the instance. This is analogous to the scalar regression of expected runtime in single algorithm selection (Sec. 2.3), but here the whole RTD is predicted. Our allocators can be implemented using an arbitrary regression model: in our experiments, we used a simple non-parametric estimator [Wichert and Wilke, 2005]. 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. Using estimates of the instance RTDs results in suboptimal allocations, but comparative experiments with more correct models showed that the impact of model precision on allocation performance is small.

The performance obtained is more than satisfactory: both methods obtain competitive results while starting from scratch, always comparable to, or better than, the default allocator. For GAMBLETA, several experiments were performed in a variety of hard practical settings, including solver competitions in different fields. The performance was quite robust, often improving over the best algorithm. We compared our results with those of OFFG-ORACLE [Streeter, 2007], an offline method based on a priori knowledge of the runtimes. As this method is per set, we did not use instance features, finding that the performance of GAMBLETA, starting from scratch, was often surprisingly close to that of OFFG-ORACLE. For three of the competitions we could also compare our results with those of another online method, ONG-EXP3 [Streeter, 2007], finding the performance of GAMBLETA to be similar. ONG-EXP3 is a per set method which can be adapted

to deal with a few categorical instance features; GAMBLETA can deal in a principled manner with discrete and continuous features, and is inherently per instance.

In this work we considered the use of several algorithms to solve a sequence of problem instances, but our methods can be applied in general to allocate computation time to several "primitives" in order to speed up a sequence of "tasks", for example in database search, data transmission over computer networks, etc.

10.2 Original contributions

Here we summarize the original contributions of this thesis, referring the reader to the corresponding section, and to the publication where each idea was first presented.

Online time allocation: GAMBLETA**.** We addressed the problem of allocating time to a set of generalized Las Vegas algorithms, whose performance is completely unknown a priori, under the very mild hypothesis that each instance can be solved by least one of the algorithms (Sec. 5.1). We adopted an online approach, updating the information on the algorithms (a runtime sample) while solving a sequence of instances (Sec. 7.1, Gagliolo and Schmidhuber [2005]). The uniform portfolio is used in alternative to the portfolios evaluated by one (or more) non-oblivious time allocators, based on the runtime sample observed so far. The two (or more) allocators are viewed as arms of a bandit (Ch. 4): for each subsequent instance, a bandit problem solver (BPS) selects the time allocator to be used. Thanks to the optimal regret of the BPS, the performance of GAMBLETA approaches the performance of the best allocator (Sec. 7.3, Gagliolo and Schmidhuber [2006c]).

Unbounded losses: EXP3LIGHT-A**.** In GAMBLETA, the total runtime spent by the algorithms corresponds to the loss to be attributed to the time allocator used. Using a BPS for bounded losses would have the disadvantage of requiring prior information on the runtimes involved. We therefore devised a BPS that can deal with an unknown bound on losses, via a doubling trick, proving a bound on its regret (Sec. 7.5, Gagliolo and Schmidhuber [2008a]).

Static portfolios. Three static TAs were proposed, optimizing a resource sharing portfolio based on the survival functions of the instance RTDs of the algorithms. The expected time, quantile, and solution probability at a contract time are optimized respectively (Sec. 5.2, Gagliolo and Schmidhuber [2006c]). These three allocators can be evaluated also for improper RTDs (Sec. 2.4.1), and can thus be applied to a set of generalized LVAs, not all guaranteed to solve the instance.

Allocation of multiple processors. The static portfolios described above can be adapted to allocate time on multiple processors. For the quantile and contract allocator, we proved that this can be performed without increasing the size of the schedule space (Sec. 5.3, Gagliolo and Schmidhuber [2008b]).

Dynamic time allocation. A dynamic schedule can be obtained updating a static schedule periodically, while solving an instance (Sec. 5.4, Gagliolo and Schmidhuber [2005], Gagliolo and Schmidhuber [2008b] for multiple processors). The same approach can be taken to evaluate the task switching schedule of [Streeter et al., 2007]. For this and our RTD based portfolios, the update can be performed simply conditioning the RTD on the time spent (Sec. 5.4, Gagliolo and Schmidhuber [2006c]).

Per instance allocation. Research on per instance portfolios and restarts was so far theoretical, as the instance RTDs were assumed to be available a priori [Luby et al., 1993; Huberman et al., 1997; Gomes and Selman, 2001; Finkelstein et al., 2002], (Sec. 2.4). More recently,

model-free methods have been proposed for evaluating per set portfolios [Petrik, 2005a; Sayag et al., 2006; Streeter et al., 2007] (Sec. 2.5). Our research provides the first practical implementation of a per instance portfolio, which is in principle more efficient than per set allocation. Also the greedy task-switching portfolio of [Streeter et al., 2007] can be evaluated based on the instance RTD, effectively turning it into a per instance method (Sec. 5.4). The instance RTD can be approximated using a regression RTD model (Sec. 3.4), conditioned on discrete or continuous instance features (Sec. 6.1, Gagliolo and Schmidhuber [2005]). We also provided an in depth discussion of the implications of this approximation, suggesting a more correct evaluation (Sec. 6.1), but experimental results do not seem to encourage this direction of research.

Censored RTD sampling. We made extensive use of survival analysis methods for modeling the runtime distributions of the algorithms (Ch. 3). We exploited censoring (Sec. 3.2) in order to reduce the computational cost of sampling RTDs. Censoring enabled us to use a portfolio also for collecting a runtime sample: when one of the algorithms solves the current instance, its runtime can be observed, while the runtimes of the remaining ones can be considered as censored observations, and correctly contribute to the model (Sec. 6.2, Gagliolo and Schmidhuber [2005]). We showed experimentally that the impact of censored sampling on restart performance is low (Sec. 8.2, Gagliolo and Schmidhuber [2006b]). We also discussed the issue of competing risks (Sec. 3.6) and its impact on model precision (Sect 6.3, Gagliolo and Legrand [2010]).

Online restart strategy: GAMBLER**.** A method for learning a restart strategy online (Sec. 7.2, Gagliolo and Schmidhuber [2007]). As in GAMBLETA, also in this case two (or more) strategies are viewed as arms of a bandit, and a BPS is used to choose among them. One arm is the universal strategy of Luby et al. [1993], which will eventually try restart thresholds of an arbitrary order of magnitude. The other arm is the uniform strategy of Luby et al., based on an RTD model, learned incrementally as more instances are solved. Additional strategies can be added as further arms.

10.3 Future work

In this section we enumerate several possible directions for future research, discussing their potential impact.

Extension to optimization problems. This is the most promising and challenging direction for future research. The most general performance model for optimization algorithms is a bivariate distribution, relating runtime to solution quality. This bivariate distribution can be analyzed considering runtime as a dependent variable, modeling the *solution quality distribution* (SQD) for an arbitrary runtime value [Hoos and Stützle, 2004]. In statistical terminology, this is an example of *longitudinal data*, which can be described using *mixed effects* models [Fitzmaurice et al., 2008]. In [Gagliolo et al., 2009] 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.

Time-varying covariates. The integration of time varying covariates in our dynamic portfolios is straightforward, as it was already envisioned (Alg. 5.4, Sec. 5.4): it suffices to condition the RTD also on the current value of the covariate. The main difficult in this sense is a practical one, as the modeling methods which we are aware of require to model such covariates as longitudinal data, which is computationally expensive (see above).

Feature selection. One limitation of using a non-parametric regression RTD model is that such models suffer from a curse of dimensionality (Sec. 3.4). It then becomes crucial to identify a small set of features which have the largest impact on the runtime. An alternative route is to adopt different models, for example semi-parametric regression models as (3.18), which were shown in medical applications to be able to cope well with high dimensional covariates [Li, 2006]. A disadvantage of such models is that they require a much stronger hypothesis on the data, that individual displays the same underlying hazard: in our case, this hypothesis would would only be satisfied by an algorithm whose hazard was the same on different instances, apart from a constant multiplicative factor which is a function of covariates of the instance. It is difficult to imagine a similar situation, but also in this case the model should be evaluated based on the time allocation it allows to perform, rather than on its precision.

Integration of portfolios and restarts. This would only be useful for heavy-tailed algorithms (Sec. 2.4.1). Restarts can be integrated with the ideal portfolios (Sec. 5.2) in a straightforward manner: for each algorithm, the optimal strategy can be evaluated based on the instance RTD (Sec. 2.4.2), and the RTDs of the algorithms with restarts can be used in place of the original ones when evaluating the RTD of the portfolio (2.16). In practice, integration in GAMBLETA is more problematic as it could happen that, in the beginning of the problem sequence, restart thresholds are incorrectly evaluated as too short, such that none of the algorithms can ever solve the current instance. Note that this cannot happen in GAMBLER, where the BPS is used repeatedly during the solution of a single instance, alternating the threshold of the model-based strategy and those of the universal strategy, such that the instance will eventually be solved. A possible turnaround could be to replace each algorithm in GAMBLETA with a copy of GAMBLER.

Extension to non-stationary settings. The main motivation for online learning in GAMBLETA is reducing the computational cost of runtime sampling to the bare minimum which allows to profit from the RTD models, but the sample is kept indefinitely. Another motivation could be to adapt to a non-stationary setting, where the instances characteristics change over time. When this is the case, a possible approach could be to discard old data, predicting RTDs based on a heuristically set number of recent observations. Alternatively, in a stationary setting, one could eventually stop learning, and continue using only the time allocator which is assigned the highest probability by the BPS. While this would violate the worst-case bounds on regret, it would be more practical in terms of the implementation, as growing the runtime sample will eventually have an impact on the computational overhead of GAMBLETA (see note 3, p. 129).

Algorithm set selection. The number of algorithms present in the set does not pose a practical problem during allocation, as the share to redundant algorithms can equal 0. It does determine the size of the schedule space, which can have an impact on the computational cost of our time allocators, as they are based on gradient descent (Sec. 5.4). Moreover, it increases the cost of the uniform portfolio, which can always be chosen by the BPS. An heuristic for "pruning" the algorithm set could therefore reduce the cost of GAMBLETA further, especially during the initial portion of the instance sequence. Such heuristic could be inspired by the preliminary algorithm set selection performed by SATZILLA [Xu, Hutter, Hoos and Leyton-Brown, 2007]. A more principled approach could be inspired by the novel framework of *combinatorial bandits* [Cesa-Bianchi and Lugosi, 2009], where multiple arms can be played simultaneously at each trial.

Approximated share evaluation. In our current implementation of the resource sharing portfolios (Sec. 5.2), the share is evaluated doing gradient descent, which makes its worst case computational complexity exponential in the number of algorithms N. Preliminary experiments suggested that the computational cost could be reduced recurring to a very rough approximation

of the share, with a small impact on performance. The practical utility of such an optimized implementation is questionable, as the overhead of time allocation is negligible for reasonably sized algorithm sets ($10 - 15$ algorithms). Moreover, the greedy task-switching portfolio of [Streeter et al., 2007], whose complexity is only linear in N, achieves a similar performance. While a dynamic resource-sharing schedule can in principle be more powerful than a task switching schedule, where only one algorithm is active at a given time, the simplicity of implementation and satisfactory performance of the latter diminishes the potential advantage of further research on the former.

Multiprocessor greedy task-switching portfolios. One aspect for which resource sharing portfolios maintain an advantage is that they can be easily parallelized on multiple processors (Sec. 5.3). In this sense, a more useful direction for research may be to parallelize the GREEDY allocator of Streeter et al. [2007]. This should be straightforward as this allocator depends on the CDF, as CONTRACT (see Sec. 5.3): we therefore expect a similar reduction of search space size to be possible.

Parallel and distributed implementations. Being for research purposes, our implementation of GAMBLETA uses pre-collected runtime values, simulating the actual execution of the portfolios. For a practical application, GAMBLETA could be easily implemented and run on a cluster of machines. In this case, the front-end could be used to allocate time on the nodes, communicating via the network. The amount of communication required would be minimal: the front-end would send the corresponding share to each node, and receive the runtimes observed when an instance is solved. If necessary, GAMBLETA can be easily implemented in a fully distributed fashion. In such case, runtime data can be broadcast, in order to allow each node to update a local copy of the RTD models; the time allocation algorithm being pseudo-random, it can be reproduced deterministically, such that each node can independently evaluate the same allocation, and execute the job(s) assigned to itself. Existing distributed computing techniques can be used at a lower level, to deal with message losses and node failures.

Appendix A

Proofs of theorems

A.1 Distributed time allocators

Homogeneous share: Be $\mathbf{s}_j$ the share for CPU j (i.e., the j-th column of $\mathbf{\Theta}$). A share $\mathbf{\Theta} = \{s_{kj}\}$ is *homogeneous* iff $s_{kj} = s_k \ \forall (k, j)$, i.e., $\mathbf{s}_j = \mathbf{s} \ \forall j$.

Theorem 1. *For each contract-optimal share $\mathbf{\Theta}^*(t_u)$ there is an homogeneous equivalent $\mathbf{\Theta}^*_h(t_u)$ such that $S_{\mathcal{A}}(t_u; \mathbf{\Theta}^*) = S_{\mathcal{A}}(t_u; \mathbf{\Theta}^*_h)$.*

Proof. Consider a non homogeneous contract-optimal share $\mathbf{\Theta}^*(t_u)$, with $\mathbf{s}^*_j \neq \mathbf{s}^*_i$ for a pair of columns (i, j):

$$\mathbf{\Theta}^*(t_u) = \arg\min_{\mathbf{\Theta}} S_{\mathcal{A},\mathbf{\Theta}}(t_u) = \arg\min_{\{\mathbf{s}_j\}} \prod_{j=1}^{Z} S_{\mathcal{A},\mathbf{s}_j}(t_u). \tag{A.1}$$

If $S_{\mathcal{A},\mathbf{s}_i}(t_u) > S_{\mathcal{A},\mathbf{s}_j}(t_u)$, then replacing $\mathbf{s}_i$ with $\mathbf{s}_j$ produces a better share, violating the hypothesis of optimality of $\mathbf{\Theta}^*$. As this must hold for any i, j, then $S_{\mathcal{A},\mathbf{s}_i}(t_u)$ must be the same for all i. Setting all $\mathbf{s}_j$ to a same $\mathbf{s}_i$ will therefore produce a homogeneous optimal share $\mathbf{\Theta}^*_h(t_u)$. □

Theorem 2. *For each quantile-optimal share $\mathbf{\Theta}^*(\alpha)$ there is an homogeneous equivalent $\mathbf{\Theta}^*_h(\alpha)$ such that $F^{-1}_{\mathcal{A}}(\alpha; \mathbf{\Theta}^*) = F^{-1}_{\mathcal{A}}(\alpha; \mathbf{\Theta}^*_h)$.*

Proof. From its definition, a quantile t_α is

$$t_\alpha = \min\{t | F(t) = \alpha\} = \min\{t | S(t) = (1 - \alpha)\}; \tag{A.2}$$

this, together with the monotonicity of the survival function, and of the logarithm function, implies:

$$\ln(1 - \alpha) = \ln S(t_\alpha) < \ln S(t) \ \ \forall \ \ t < t_\alpha. \tag{A.3}$$

Consider a non homogeneous optimal share $\mathbf{\Theta}^*$, with $\mathbf{s}^*_j \neq \mathbf{s}^*_i$ for a pair $(i, j), i \neq j$. We can write:

$$\ln(1 - \alpha) = \ln S_{\mathcal{A},\mathbf{\Theta}}(t_\alpha) = \sum_{j=1}^{Z} \ln S_{\mathcal{A},\mathbf{s}_j}(t_\alpha) \tag{A.4}$$

Pick now an arbitrary column $\mathbf{s}_i^*$ of $\mathbf{\Theta}^*$, and set all other columns $\mathbf{s}_j^*$ to $\mathbf{s}_i^*$, obtaining a homogeneous share with quantile t_i

$$\ln(1-\alpha) = Z \ln S_{\mathcal{A},\mathbf{s}_i}(t_i) \tag{A.5}$$

While $t_i < t_\alpha$ violates the hypothesis of optimality of $\mathbf{\Theta}^*(\alpha)$, $t_\alpha < t_i$ would imply for (A.3) (with t_i in place of t_α):

$$\ln S_{\mathcal{A},\mathbf{s}_i}(t_i) = \frac{\ln(1-\alpha)}{Z} < \ln S_{\mathcal{A},\mathbf{s}_i}(t_\alpha). \tag{A.6}$$

Summing over i gives a contradiction ($\ln(1-\alpha) < \ln(1-\alpha)$), so the only possibility left is $t_i = t_\alpha$. As this must hold for any i, then t_i must be constant $\forall i$. Setting all $\mathbf{s}_j$ to a same $\mathbf{s}_i$ will then produce a homogeneous optimal share $\mathbf{\Theta}_h^*(\alpha)$. Note that, different from the contract, in this case $\mathbf{\Theta}_h^*$ depends on Z. □

Theorem 3. *Expected-value-optimal shares do not necessarily have an homogeneous equivalent.*

Proof. Consider a simple example: $Z = N = 2$, $S_1(t) = 1$ for $t \in [0,1]$, $S_1(t) = 0.5$ for $t \in [1,4]$, $S_1(t) = 0$ for $t > 4$; $S_2(t) = 1$ for $t \in [0,2]$, $S_2(t) = 0$ for $t > 2$. The optimal share ($E\{t\} = 1.5$) allocates one CPU per algorithm, so it is not homogeneous, and there is no homogeneous share giving the same expected time. □

A.2 Worst-case performance of GambleR

Theorem 4. *On a given problem instance, if R_U is a bound on the number of runs of the universal restart strategy, t_U the bound on its runtime (2.13), and the BPS plays according to a $\mathbf{p}$ such that $p_k \geq p_{min}$ for all arms k, and for all trials, then the runtime t_G of* GambleR *on the instance is bounded in expectation as*

$$E\{t_G\} \leq E\{t_U + R_U \hat{\tau} \frac{1-p_{min}}{p_{min}}\} \tag{A.7}$$

Proof. In the following, t_k represents the solution time of strategy k on a single problem, R_k the number of restarts performed, $p_k \in [0,1]$ the probability of the BPS picking the k-th restart strategy, and indexes τ^*, U, $\hat{\tau}$, label quantities related to the unknown optimal uniform, universal, and estimated optimal uniform strategies, respectively, while G will identify the GambleR strategy.

On a given problem, for which the RTD of our algorithm a is $F(t)$, a restart strategy with thresholds $\tau(r)$ can be viewed as sequence of independent Bernoulli processes, with success probabilities $F(\tau(r))$. The number of restarts R required to solve the problem is then distributed according to the discrete pdf

$$p(R) = \prod_{r=1}^{R-1} (1 - F(\tau(r)))F(\tau(R)). \tag{A.8}$$

For a uniform strategy $\tau(r) = \tau$, $p(R_\tau)$ is geometric, with $E\{R_\tau\} = 1/F(\tau)$. For any deterministic strategy $\tau(r)$, the expected value of the total runtime $t_{\tau(r)}$ is

$$E\{t_{\tau(r)}\} = \sum_{r=1}^{E\{R\}} \tau(r), \tag{A.9}$$

a monotonic function of $E(R)$. For U this implies that the bound (2.13) can be translated into a bound on R_U, with the same high probability.

Given our simple reward scheme (Alg. 7), the BPS will keep a constant $\mathbf{p} = (p_U, p_{\hat{\tau}})$ during solution of a single problem. If the BPS always keeps its probabilities above a lower threshold $p_{\min} > 0$, as it is often the case, then $p_U \geq p_{\min}$.

Consider now the worst-case setting in which $\hat{\tau}$ is such that $F(\hat{\tau}) = 0$, i.e., the uniform restart $\hat{\tau}$ will *never* solve the problem. If U spends R_U restarts, in the meantime another $R_{\hat{\tau}}$ restarts will have been wasted on $\hat{\tau}$, with

$$E\{R_{\hat{\tau}}\}) = E\{R_U\}\frac{1-p_U}{p_U} \leq E\{R_U\}\frac{1-p_{\min}}{p_{\min}}. \tag{A.10}$$

The expected runtime of GAMBLER will then be bounded as (A.7) with high probability, and upper bounds on t_U and R_U (2.13) will also guarantee an upper bound on t_G.

□

A.3 Unbounded losses

Theorem 5. 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:*

$$\begin{aligned} & E\{L_E(M)\} - L^*(M) \leq \\ \leq\ & 2\sqrt{6\mathcal{L}(\log K + K\log M)KL^*(M)} \\ +\ & \mathcal{L}[2\sqrt{2\mathcal{L}(\log K + K\log M)K} \\ +\ & (2K+1)(1+\log_4(3M+1))] \end{aligned} \tag{A.11}$$

Proof. The proof is trivially based on the regret for the original EXP3LIGHT, with $\mathcal{L} = 1$, which according to [Cesa-Bianchi, Mansour and Stoltz, 2005, Theorem 5] (proof obtained from Cesa-Bianchi [2008]) can be evaluated using the optimal values (4.2) for η_r :

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

Playing a game with losses in $[0, \mathcal{L}]$, simply dividing all losses by $\mathcal{L}$, the following will hold for the actual losses observed:

$$\frac{E\{L_E(M)\} - L^*(M)}{\mathcal{L}} \leq \tag{A.13}$$

$$2\sqrt{2(\log K + K\log M)K(1+3L^*(M)/\mathcal{L})} \tag{A.14}$$

$$+\ (2K+1)(1+\log_4(3M+1)). \tag{A.15}$$

Multiplying both sides for $\mathcal{L}$ and rearranging produces (A.11). □

Theorem 6. Regret of EXP3LIGHT-A.

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

$$\begin{aligned} & E\{L_E(M)\} - L^*(M) \leq \\ & 4\sqrt{3\lceil \log_2 \mathcal{L} \rceil \mathcal{L}(\log K + K \log M) K L^*(M)} \\ + \quad & 2\lceil \log_2 \mathcal{L} \rceil \mathcal{L}[\sqrt{4\mathcal{L}(\log K + K \log M)K} \\ + \quad & (2K+1)(1+\log_4(3M+1)) + 2] \end{aligned} \tag{A.16}$$

Proof. This follows the proof technique employed in [Cesa-Bianchi, Mansour and Stoltz, 2005, 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 (A.11) 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:

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

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

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

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

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

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

[1] 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, (A.17) is zero, and (A.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 (A.19) can be bounded using Jensen's inequality

$$\sum_{u=0}^{U} \sqrt{a_u} \leq \sqrt{(U+1)\sum_{u=0}^{U} a_u}, \tag{A.19}$$

with

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

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

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

which implies

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

In this way we can bound the sum

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

We conclude by noting that

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

Inequality (A.19) then becomes:

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

Plugging in (A.21, A.22) and rearranging, we obtain (A.16).

□

Appendix B

Competing risks

Consider the *joint survival function* of the times $\{t_k\}$

$$Z(t_1, \ldots, t_K) = \Pr\{T_1 > t_1, \ldots, T_K > t_k\}. \tag{B.1}$$

The probability of the event $\{T_k > t_k\}$ is the *marginal* survival function

$$\Pr\{T_k > t_k\} = Z_k(t_k) = Z(0, \ldots, t_k, \ldots, 0). \tag{B.2}$$

If the event times $\{T_k\}$ are independent, the joint probability (B.1) equals the product of the probabilities of the single events (B.2):

$$Z(t_1, \ldots, t_K) = \prod_{j=1}^{K} Z_j(t_j). \tag{B.3}$$

Unfortunately, if the data is gathered as described in Section 6.2, with at most one uncensored event time per individual, the independence assumption cannot be tested. Neither the joint survival function (B.1), nor the marginals (B.2) can be identified, and Kaplan-Meier estimates of the marginals for each risk will be biased [Tsiatis, 1975; Putter et al., 2006]: in practice, the survival function related to the slower risks will be overestimated.

One can always estimate the overall survival probability for an individual,

$$S(t) = \Pr\{T_1 > t, \ldots, T_K > t\} = Z(t_1 = t, t_2 = t, \ldots, t_K = t), \tag{B.4}$$

with a product limit estimate (3.16) from the data, considering all recorded events as they were the same event, with $C = 1$. Other quantities that can be estimated [Pintilie, 2006; Putter et al., 2006] are the *cause-specific hazard*, defined as the hazard of failing for a specific cause k:

$$\lambda_k(t) = \lim_{\Delta t \to 0} \frac{\Pr\{t \leq T < t + \Delta t, C = k \mid T \geq t\}}{\Delta t}, \tag{B.5}$$

from which one can obtain the cumulative cause-specific hazard,

$$\Lambda_k(t) = \int_0^t \lambda_k(\tau) d\tau, \tag{B.6}$$

and the *subsurvival* function,

$$R_k(t) = \exp(-\Lambda_k(t)), \tag{B.7}$$

with

$$S(t) = \prod_k R_k(t) = \exp(-\sum_k \Lambda_k(t)). \tag{B.8}$$

The *cumulative incidence function* or *subdistribution function* for the cause k is defined as the probability of failing from cause k before time t:

$$I_k(t) = \Pr\{T \leq t, C = k\} = \int_0^t \lambda_k(\tau) S(\tau) d\tau. \tag{B.9}$$

It is an improper distribution, as $I(\infty) = \Pr\{C = k\} = \Pr\{T_j > T_k \forall j \neq k\}$, which can be smaller than 1; note also that $I_k(t) + R_k(t) = \Pr\{C = k\}\ \forall t$.

It is interesting to note that the "naive" approach to competing risks, which consists in estimating the probability of failing from cause k before time t based on the Kaplan-Meier estimate, censoring other causes of failure, actually estimates

$$1 - \hat{S}_k(t) = \hat{F}_k(t) = \int_0^t \lambda_k(\tau) R_k(\tau) d\tau. \tag{B.10}$$

which differs from (B.9) in the presence of the subsurvival function $R_k(t)$ in place of the overall survival probability $S(t)$. In other words, the naive KM estimate only takes into account the probability of failing from cause k, while the cumulative incidence curve takes into account the fact that an individual must survive *all* causes of failure up to t to fail from cause k at t. This explains why the Kaplan-Meier estimates obtained while censoring all other events leads to a biased estimate of the marginal survival function for a single cause k.

Bibliography

Aalen, O. [1978]. Nonparametric inference for a family of counting processes, *Annals of Statistics* **6**: 701–726. 3.3.2

Akritas, M. [1994]. Nearest neighbor estimation of a bivariate distribution under random censoring, *Annals of Statistics* **22**: 1299–1327. 3.4

Allenberg, C., Auer, P., Györfi, L. and Ottucsák, G. [2006]. Hannan consistency in on-line learning in case of unbounded losses under partial monitoring, *in* J. L. Balcázar, P. M. Long and F. Stephan (eds), *Algorithmic Learning Theory — ALT*, Vol. 4264 of *LNCS*, Springer, pp. 229–243. 4.3, 7.5

Alt, H., Guibas, L. J., Mehlhorn, K., Karp, R. M. and Wigderson, A. [1996]. A method for obtaining randomized algorithms with small tail probabilities, *Algorithmica* **16**(4/5): 543–547. 2.4.2

Applegate, D. [2006]. *The Traveling Salesman Problem*, Westview, Boulder. 2.1

Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R. E. [1995]. Gambling in a rigged casino: the adversarial multi-armed bandit problem, *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, pp. 322–331. 4.2, 7.4

Auer, P., Cesa-Bianchi, N., Freund, Y. and Schapire, R. E. [2002]. The nonstochastic multiarmed bandit problem, *SIAM Journal on Computing* **32**(1): 48–77. 1.5, 4.2, 4.5, 7.4, 9.3.1

Babai, L. [1979]. Monte-Carlo algorithms in graph isomorphism testing., *Technical Report 79-10*, Univ. de Montréal, Dép. de mathématiques et de statistique. 1.1

Baños, A. [1968]. On pseudo-games, *Annals of Mathematical Statistics* **39**: 1932–1945. 4.2

Battiti, R., Brunato, M. and Mascia, F. [2008]. *Reactive Search and Intelligent Optimization*, Vol. 45 of *Operations Research/Computer Science Interfaces*, Springer Verlag, Berlin. 2.2

Battiti, R. and Protasi, M. [1997]. Reactive search, a history-sensitive heuristic for MAX-SAT, *Journal of Experimental Algorithmics* **2**: 2. 5.5

Beck, C. J. and Freuder, E. C. [2004]. Simple rules for low-knowledge algorithm selection., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems — CPAIOR*, Vol. 3011 of *LNCS*, Springer, pp. 50–64. 2.2, 2.6

Beran, R. [1981]. Nonparametric regression with randomly censored survival data, *Technical report*, University of California, Berkeley, CA. 3.4

Birattari, M. [2004]. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*, PhD thesis, Université Libre de Bruxelles, Brussels, Belgium.
URL: `http://iridia.ulb.ac.be/~mbiro/thesis/` 4.5

Birattari, M. [2009]. *Tuning Metaheuristics: A Machine Learning Perspective*, Springer-Verlag, Berlin. 2.1

Birattari, M., Stützle, T., Paquete, L. and Varrentrapp, K. [2002]. A racing algorithm for configuring metaheuristics, *in* W. B. Langdon et al. (eds), *Proceedings of the Genetic and Evolutionary Computation Conference — GECCO*, Morgan Kaufmann Publishers, pp. 11–18. 4.5

Bishop, C. M. [1995]. *Neural networks for pattern recognition*, Oxford University Press. 2.4.2, 3.3.1, 3.4

Bishop, C. M. [2006]. *Pattern Recognition and Machine Learning*, Springer, Berlin. 2.3.2, 2.3.3, 6.4

Boisvert, R. F. [2000]. Mathematical software: Past, present, and future, *Technical report*, National Institute of Standards and Technology.
URL: `http://arxiv.org/abs/cs/0004004` 2.3.1

Borrett, J. E., Tsang, E. P. K. and Walsh, N. R. [1996]. Adaptive constraint satisfaction: The quickest first principle, *in* W. Wahlster (ed.), *European Conference on Artificial Intelligence — ECAI*, John Wiley and Sons, pp. 160–164. 5.5

Carchrae, T. and Beck, J. C. [2005]. Applying machine learning to low knowledge control of optimization algorithms, *Computational Intelligence* **21**(4): 373–387. 2.6, 4.5

Cesa-Bianchi, N. [2008]. Personal communication. A.3

Cesa-Bianchi, N. and Lugosi, G. [2006]. *Prediction, learning, and games*, Cambridge University Press. 4, 4.2

Cesa-Bianchi, N. and Lugosi, G. [2009]. Combinatorial bandits, *Conference on Learning Theory — COLT*, pp. 237–246. 10.3

Cesa-Bianchi, N., Lugosi, G. and Stoltz, G. [2005]. Minimizing regret with label efficient prediction, *IEEE Transactions on Information Theory* **51**(6): 2152–2162. 4.5

Cesa-Bianchi, N., Mansour, Y. and Stoltz, G. [2005]. Improved second-order bounds for prediction with expert advice, *in* P. Auer and R. Meir (eds), *18th Annual Conference on Learning Theory — COLT*, Vol. 3559 of *LNCS*, Springer, pp. 217–232. 4.2, 4.2, 4.3, 7.5, 7.5, A.3, A.3

Cesa-Bianchi, N., Mansour, Y. and Stoltz, G. [2007]. Improved second-order bounds for prediction with expert advice, *Machine Learning* **66**(2-3): 321–352. 4.3, 7.5

Cherkassky, V. and Mulier, F. [1998]. *Learning from Data — Concepts, Theory and Methods*, John Wiley & Sons, New York. 2.3.2

Cicirello, V. A. and Smith, S. F. [2005]. The max k-armed bandit: A new model of exploration applied to search heuristic selection., *in* M. Veloso and S. Kambhampati (eds), *Twentieth National Conference on Artificial Intelligence — AAAI*, AAAI Press, pp. 1355–1361. 4.5

Collet, D. [2003]. *Modeling survival data in medical research*, Chapman & Hall/CRC, Boca Raton, Florida. 3

Cox, D. [1972]. Regression models and life-tables, *Journal of the Royal Statistics Society, Series B* **34**: 187–220. 3.4

Cox, D. and Oakes, D. [1984]. *Analysis of survival data*, Chapman & Hall, London. 3.4

Cristianini, N., Campbell, C. and Taylor, S. J. [1999]. Dynamically adapting kernels in support vector machines, *in* S. A. Solla, T. K. Leen and K. R. Müller (eds), *Advances in Neural Information Processing Systems — NIPS*, Vol. 11, The MIT Press, pp. 204–210. 2.3.2

Davis, M., Logemann, G. and Loveland, D. [1962]. A machine program for theorem-proving, *Communications of the ACM* **5**(7): 394–397. 2.1.1

Fayolle, G., Gelenbe, E. and Pujolle, G. [1978]. An analytic evaluation of the performance of the "send and wait" protocol, *IEEE Transactions on Communications* **26**(3): 313–319. 2.4.2

Fink, E. [2004]. Automatic evaluation and selection of problem-solving methods: theory and experiments, *Journal of Experimental & Theoretical Artificial Intelligence* **16**(2): 73–105. 6.4

Finkelstein, L., Markovitch, S. and Rivlin, E. [2002]. Optimal schedules for parallelizing anytime algorithms: the case of independent processes, *Eighteenth national conference on Artificial intelligence — AAAI*, AAAI Press, pp. 719–724. 2.2, 2.4.3, 5.5, 10.1, 10.2

Finkelstein, L., Markovitch, S. and Rivlin, E. [2003]. Optimal schedules for parallelizing anytime algorithms: The case of shared resources, *Journal of Artificial Intelligence Research* **19**: 73–138. 1.2, 2.4.3, 5.5, 10.1

Fitzmaurice, G., Davidian, M., Verbeke, G. and Molenberghs, G. [2008]. *Longitudinal Data Analysis*, Chapman & Hall/CRC Press. 3.5, 10.3

Fleming, T. R. and Harrington, D. P. [1991]. *Counting processes and survival analysis*, John Wiley & Sons, Ltd., New York. 3.3.1

Frost, D., Rish, I. and Vila, L. [1997]. Summarizing CSP hardness with continuous probability distributions, *in* B. Kuipers and B. Webber (eds), *Fourteenth National Conference on Artificial Intelligence — AAAI*, AAAI Press, pp. 327–333. 2.4.1, 3.3.1, 1, 8.2

Fürnkranz, J. [2001]. On-line bibliography on meta-learning.
URL: `http://www.ofai.at/research/impml/metal/metal-bib.html` 2.3.2

Fürnkranz, J., Petrak, J., Brazdil, P. and Soares, C. [2002]. On the use of fast subsampling estimates for algorithm recommendation, *Technical Report TR-2002-36*, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien. 2.3.2

Gagliolo, M. [2007]. Universal search, *Scholarpedia* **2**(11): 2575.
URL: `http://www.scholarpedia.org/article/Universal_Search` 2.2

Gagliolo, M. and Legrand, C. [2010]. Algorithm survival analysis, *in* T. Bartz-Beielstein, M. Chiarandini, L. Paquete and M. Preuss (eds), *Empirical Methods for the Analysis of Optimization Algorithms*, Natural Computing, Springer, Berlin, pp. 159–182. To appear. 10.2

Gagliolo, M., Legrand, C. and Birattari, M. [2009]. Mixed-effects modeling of optimisation algorithm performance., *in* T. Stützle, M. Birattari and H. H. Hoos (eds), *Engineering Stochastic Local Search Algorithms — SLS*, Vol. 5752 of *LNCS*, Springer, pp. 150–154. 3.5, 10.3

Gagliolo, M. and Schmidhuber, J. [2005]. A neural network model for inter-problem adaptive online time allocation, *in* W. Duch et al. (eds), *Artificial Neural Networks: Formal Models and Their Applications — ICANN, Proceedings, Part 2*, Vol. 3697 of *LNCS*, Springer, Berlin, pp. 7–12. 3.4, 5.5, 6.4, 7.6, 10.2

Gagliolo, M. and Schmidhuber, J. [2006a]. Dynamic algorithm portfolios, *Ninth International Symposium on Artificial Intelligence and Mathematics — AI&MATH*.
URL: `http://anytime.cs.umass.edu/aimath06/proceedings/P37.pdf` 3.4

Gagliolo, M. and Schmidhuber, J. [2006b]. Impact of censored sampling on the performance of restart strategies, *in* F. Benhamou (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 4204 of *LNCS*, Springer, pp. 167–181. 10.2

Gagliolo, M. and Schmidhuber, J. [2006c]. Learning dynamic algorithm portfolios, *Annals of Mathematics and Artificial Intelligence* **47**(3-4): 295–328. 5.5, 7.4, 7.6, 10.2

Gagliolo, M. and Schmidhuber, J. [2007]. Learning restart strategies, *in* M. M. Veloso (ed.), *Twentieth International Joint Conference on Artificial Intelligence — IJCAI, vol. 1*, AAAI Press, pp. 792–797.
URL: `http://www.ijcai.org/papers07/Papers/IJCAI07-127.pdf` 10.2

Gagliolo, M. and Schmidhuber, J. [2008a]. Algorithm selection as a bandit problem with unbounded losses, *Technical Report IDSIA-07-08*, IDSIA, Lugano, Switzerland. To appear in LION4 Proceedings, Springer LNCS, 2010.
URL: `http://arxiv.org/abs/0807.1494` 10.2

Gagliolo, M. and Schmidhuber, J. [2008b]. Towards distributed algorithm portfolios, *in* J. M. Corchado and Others (eds), *International Symposium on Distributed Computing and Artificial Intelligence — DCAI*, Vol. 50 of *Advances in Soft Computing*, Springer, pp. 634–643. 10.2

Gagliolo, M., Zhumatiy, V. and Schmidhuber, J. [2004]. Adaptive online time allocation to search algorithms, *in* J. F. Boulicaut, F. Esposito, F. Giannotti and D. Pedreschi (eds), *Machine Learning: ECML 2004. Proceedings of the 15th European Conference on Machine Learning*, Vol. 3201 of *LNCS*, Springer, pp. 134–143. 2.6, 5.5, 10.1

Garey, M. R. and Johnson, D. S. [1990]. *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA. 1.1, 2.1

Gent, I. P., Hoos, H. H., Prosser, P. and Walsh, T. [1999]. Morphing: Combining structure and randomness, *in* J. Hendler and D. Subramanian (eds), *Sixteenth National Conference on Artificial Intelligence — AAAI*, AAAI Press, pp. 654–660. 2.4.1, 8.1

Gent, I. and Walsh, T. [1999]. The search for satisfaction, *Technical report*, Dept. of Computer Science, University of Strathclyde. 1.3, 2.1.1

Giraud-Carrier, C., Vilalta, R. and Brazdil, P. [2004]. Introduction to the special issue on meta-learning, *Machine Learning* **54**(3): 187–193. 2.3.2

Gomes, C. P., Fernandez, C., Selman, B. and Bessiere, C. [2005]. Statistical regimes across constrainedness regions, *Constraints* **10**: 317–337. 2.4.1

Gomes, C. P. and Selman, B. [2001]. Algorithm portfolios, *Artificial Intelligence* **126**(1-2): 43–62. 1.2, 2.4.2, 2.4.3, 2.7, 5.5, 10.1, 10.2

Gomes, C. P., Selman, B., Crato, N. and Kautz, H. [2000]. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, *Journal of Automated Reasoning* **24**(1-2): 67–100. 2.4.1, 2.4.1, 6.1.1, 8.1, 9.5

Hansen, E. A. and Zilberstein, S. [2001]. Monitoring and control of anytime algorithms: A dynamic programming approach, *Artificial Intelligence* **126**(1–2): 139–157. 5.5

Harick, G. R. and Lobo, F. G. [1999]. A parameter-less genetic algorithm, *in* W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela and R. E. Smith (eds), *Proceedings of the Genetic and Evolutionary Computation Conference — GECCO*, Vol. 2, Morgan Kaufmann. 2.6

Hogg, T. and Williams, C. P. [1994]. The hardest constraint problems: a double phase transition, *Artificial Intelligence* **69**(1-2): 359–377. 2.4.1

Holland, J. H. [1975]. *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor. 2.6

Hoos, H. H. [2002]. A mixture-model for the behaviour of SLS algorithms for SAT, *in* J. A. Hendler (ed.), *Eighteenth National Conference on Artificial Intelligence — AAAI*, AAAI Press, pp. 661–667. 2.4.1

Hoos, H. H. and Stützle, T. [1998a]. Evaluating Las Vegas algorithms: Pitfalls and remedies, *in* G. F. Cooper and S. Moral (eds), *Fourteenth Conference on Uncertainty in Artificial Intelligence — UAI*, Morgan Kaufmann, pp. 238–245. 2.4.1

Hoos, H. H. and Stützle, T. [1999]. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT, *Artificial Intelligence* **112**: 213–232. 2.4.1

Hoos, H. H. and Stützle, T. [2000]. SATLIB: An online resource for research on SAT, *in* I. Gent et al. (eds), *SAT 2000 — Highlights of Satisfiability Research in the Year 2000*, IOS press, pp. 283–292.
URL: `http://www.satlib.org` 6.1.1, 9.5

Hoos, H. H. and Stützle, T. [2004]. *Stochastic Local Search : Foundations & Applications*, Morgan Kaufmann. 1.1, 1.4, 2.1, 2.1.1, 2.4.1, 5.1, 8.1, 9.5, 10.1, 10.3

Hoos, H. and Stutzle, T. [1998b]. Characterizing the run-time behavior of stochastic local search, *Technical Report AIDA-98-1*, FG Intellektik, TU Darmstadt. 2.4.1

Horvitz, E. J. and Zilberstein, S. [2001]. Computational tradeoffs under bounded resources (editorial), *Artificial Intelligence* **126**(1–2): 1–4. 2.2

Horvitz, E., Ruan, Y., Gomes, C. P., Kautz, H. A., Selman, B. and Chickering, D. M. [2001]. A Bayesian approach to tackling hard computational problems, *17th Conference on Uncertainty in Artificial Intelligence — UAI*, Morgan Kaufmann Publishers Inc., pp. 235–244. 2.4.2, 5.5

Huberman, B. A., Lukose, R. M. and Hogg, T. [1997]. An economics approach to hard computational problems, *Science* **27**: 51–53. 1.2, 2.4.3, 2.7, 5.5, 10.1, 10.2

Hutter, F. [2009]. *Automated Configuration of Algorithms for Solving Hard Computational Problems*, PhD thesis, University Of British Columbia, Department of Computer Science, Vancouver, Canada.
URL: `http://people.cs.ubc.ca/~hutter/papers/Hutter09PhD.pdf` 2.3.3, 7.6

Hutter, F. and Hamadi, Y. [2005]. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver, *Technical Report MSR-TR-2005-125*, Microsoft Research, Cambridge, UK.
URL: `http://www.cs.ubc.ca/~hutter/papers/msr_tr05-autoparam.pdf` 1.2, 2.2, 2.3.3

Hutter, F., Hamadi, Y., Hoos, H. H. and Leyton-Brown, K. [2006]. Performance prediction and automated tuning of randomized and parametric algorithms, *in* F. Benhamou (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 4204 of *LNCS*, Springer, pp. 213–228. 2.3.3

Hutter, F., Hoos, H. H. and Stützle, T. [2007]. Automatic algorithm configuration based on local search, *in* R. C. Holte and A. Howe (eds), *Twenty-Second AAAI Conference on Artifical Intelligence*, AAAI Press, pp. 1152–1157. 2.3.3

Ibrahim, J. G., Chen, M.-H. and Sinha, D. [2001]. *Bayesian Survival Analysis*, Springer, Berlin. 3.3.2

Janakiram, V. K., Agrawal, D. P. and Mehrotra, R. [1988]. A randomized parallel backtracking algorithm, *IEEE Transactions on Computers* **37**(12): 1665–1676. 2.4.3

Kaplan, E. L. and Meyer, P. [1958]. Nonparametric estimation from incomplete samples, *Journal of The American Statistical Association* **73**: 457–481. 3.3.2, 3.3.2

Kautz, H., Horvitz, E., Ruan, Y., Gomes, C. and Selman, B. [2002]. Dynamic restart policies. 2.4.2, 5.5, 6.4

Keller, J. and Giraud-Carrier, C. [2000]. ECML 2000 workshop on meta-learning: Building automatic advice strategies for model selection and method combination. 2.3.2

Klein, J. P. and Moeschberger, M. L. [2003]. *Survival Analysis: Techniques for Censored and Truncated Data*, second edn, Springer, Berlin. 1.5, 3, 3.3.1, 3.3.1, 3.4

Kolen, J. F. [1988]. Faster learning through a probabilistic approximation algorithm, *IEEE International Conference on Neural Networks*, Vol. 1, pp. 449–454. 2.4.2

Lagoudakis, M. G. and Koenig, S. [2004]. Planning, *in* C. Ghaoui (ed.), *Encyclopedia of Human Computer Interaction*, Berkshire Publishing, pp. 554–560. 9.3.6

Lagoudakis, M. G. and Littman, M. L. [2000]. Algorithm selection using reinforcement learning, *in* P. Langley (ed.), *Seventeenth International Conference on Machine Learning — ICML*, Morgan Kaufmann, pp. 511–518. 5.5

Leyton-Brown, K., Nudelman, E., Andrew, G., Mcfadden, J. and Shoham, Y. [2003]. Boosting as a metaphor for algorithm design, *in* F. Rossi (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 2833 of *LNCS*, Springer, pp. 899–903. 2.3.3

Leyton-Brown, K., Nudelman, E. and Shoham, Y. [2002]. Learning the empirical hardness of optimization problems: The case of combinatorial auctions, *in* P. Van Hentenryck (ed.), *ICCP: International Conference on Constraint Programming (CP)*, Vol. 2470 of *LNCS*, Springer, pp. 556–572. 1.2, 2.3.3, 9, 9.4, 9.4

Li, C. M. and Anbulagan [1997]. Heuristics based on unit propagation for satisfiability problems, *in* M. P. Georgeff et al. (eds), *Fifteenth International Joint Conference on Artificial Intelligence — IJCAI*, Morgan Kaufmann, pp. 366–371. 6.1.1, 8.1, 9.5

Li, C. M. and Huang, W. [2005]. Diversification and determinism in local search for satisfiability, *in* F. Bacchus et al. (eds), *Theory and Applications of Satisfiability Testing*, Springer, pp. 158–172. 9.5

Li, G. and Doss, H. [1995]. An approach to nonparametric regression for life history data using local linear fitting, *Annals of Statistics* **23**: 787–823. 3.5

Li, H. [2006]. Censored data regression in high dimension and low sample size settings for genomic applications, *Technical Report 9*, University of Pennsylvania. 10.3

Liang, K., Self, S., Bandeen-Roche, K. and Zeger, S. [1995]. Some recent developments for regression analysis of multivariate failure time data, *Lifetime Data Analysis* **1**: 403–415. 3.3.1

Littlestone, N. and Warmuth, M. K. [1994]. The weighted majority algorithm, *Information and Computation* **108**(2): 212–261. 4.2

Luby, M. and Ertel, W. [1994]. Optimal parallelization of Las Vegas algorithms, *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science — STACS*, Springer-Verlag, London, UK, pp. 463–474. 5.5

Luby, M., Sinclair, A. and Zuckerman, D. [1993]. Optimal speedup of Las Vegas algorithms, *Information Processing Letters* **47**(4): 173–180. 1.2, 2.4.2, 2.4.3, 2.6, 10, 7.2, 1, 2, 8.2, 10.1, 10.2

Machin, D., Cheung, Y. and Parmar, M. [2006]. *Survival Analysis. A practical approach.*, John Wiley & Sons, West Sussex, England. Second edition. 3

MacKay, D. C. [1992]. Comparison of approximate methods for handling hyperparameters, *Neural Computation* **11**(5): 1035–1068. 2.3.2

MacKay, D. C. [2003]. *Information Theory, Inference and Learning Algorithms*, Cambridge University Press.
URL: `http://www.inference.phy.cam.ac.uk/mackay/itila/book.html` 3.3.1

Martinussen, T. and Scheike, T. H. [2006]. *Dynamic Regression Models for Survival Data (Statistics for Biology and Health)*, Springer, Berlin. 3.4

Maturana, J., Fialho, A., Saubion, F., Schoenauer, M. and Sebag, M. [2009]. Extreme compass and dynamic multi-armed bandits for adaptive operator selection, *in* A. Tyrrell (ed.), *2009 IEEE Congress on Evolutionary Computation — CEC*, IEEE Computational Intelligence Society, IEEE Press, Trondheim, Norway. 4.5

McCracken, M. O., Snavely, A. and Malony, A. D. [2003]. Performance modeling for dynamic algorithm selection, *in* P. M. A. Sloot et al. (eds), *International Conference on Computational Science*, Vol. 2660 of *LNCS*, Springer, pp. 749–758. 2.3.1

Mitchell, D., Selman, B. and Levesque, H. [1992]. Hard and easy distributions of SAT problems, *in* P. Rosenbloom and P. Szolovits (eds), *Tenth National Conference on Artificial Intelligence — AAAI*, AAAI Press, pp. 459–465. 2.1.1

Mitchell, T. M. [1997]. *Machine Learning*, McGraw-Hill Science/Engineering/Math. 2.5, 7.6

Mollick, E. [2006]. Establishing moore's law, *IEEE Annals of the History of Computing* **28**(3): 62–75. 1.1

Moore, A. W. and Lee, M. S. [1994]. Efficient algorithms for minimizing cross validation error, *Proceedings of the 11th International Conference on Machine Learning — ICML*, Morgan Kaufmann, pp. 190–198. 4.5

Muselli, M. and Rabbia, M. [1991]. Parallel trials versus single search in supervised learning, *in* O. Simula (ed.), *Second International Conference on Artificial Neural Networks — ICANN*, Elsevier, pp. 24–28. 2.4.2, 2.4.3

Nelson, W. [1972]. Theory and applications of hazard plotting for censored failure data, *Technometrics* **14**: 945–965. 3.3.2

Nelson, W. [1982]. *Applied Life Data Analysis*, John Wiley, New York. 3, 3.1

Nielsen, J. and Linton, O. [1995]. Kernel estimation in a nonparametric marker dependent hazard model, *Annals of Statistics* **23**: 1735–1748. 3.5

Nudelman, E., Andrew, G., Mcfadden, J., Brown, K. L. and Shoham, Y. [2003]. A portfolio approach to algorithm selection, *in* G. Gottlob and T. Walsh (eds), *Eighteenth International Joint Conference on Artificial Intelligence — IJCAI*, Morgan Kaufmann, pp. 1542–1543. 2.3.3

Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A. and Shoham, Y. [2004]. Understanding random SAT: Beyond the clauses-to-variables ratio, *in* M. Wallace (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 3258 of *LNCS*, Springer, pp. 438–452. 1.2, 2.3.3

Papadimitriou, C. H. and Steiglitz, K. [1998]. *Combinatorial Optimization : Algorithms and Complexity*, Dover Publications. 1.1, 2.1

Petrik, M. [2005a]. *Learning parallel portfolios of algorithms*, Master's thesis, Comenius University. 2.2, 2.5, 5.5, 6.4, 10.2

Petrik, M. [2005b]. Statistically optimal combination of algorithms. Presented at SOFSEM. **URL:** `http://www.cs.umass.edu/~petrik/adaptive-pub.pdf` 2.5, 5.5

Petrik, M. and Zilberstein, S. [2006]. Learning parallel portfolios of algorithms, *Annals of Mathematics and Artificial Intelligence* **48**(1-2): 85–106. 1.2, 1.3, 2.5, 6.4, 7.6

Pfahringer, B., Bensusan, H. and Carrier, C. G. [2000]. Meta-learning by landmarking various learning algorithms, *Proceedings of the Seventeenth International Conference on Machine Learning — ICML*, Morgan Kaufmann, pp. 743–750. 2.3.2

Pintilie, M. [2006]. *Competing Risks. A practical perspective.*, John Wiley & Sons, Ltd., New York. 3.3.2, 3.6, B

Puterman, M. L. [1994]. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley-Interscience, New York. 2.5

Putter, H., Fiocco, M. and Geskus, R. [2006]. Tutorial in biostatistics: Competing risks and multi-state models, *Statistics in Medicine* **26**: 2389–2430. 3.6, B, B

Reed, W. J. and Jorgensen, M. [2004]. The double Pareto-lognormal distribution — A new parametric model for size distribution, *Communications in Statistics — Theory and Methods* **33**(8): 1733–1753. 3.3.1

Rice, J. R. [1976]. The algorithm selection problem, *in* M. Rubinoff and M. C. Yovits (eds), *Advances in computers*, Vol. 15, Academic Press, pp. 65–118. 1.1, 1.2, 2.3.1, 2.3.3, 6.5

Robbins, H. [1952]. Some aspects of the sequential design of experiments, *Bulletin of the AMS* **58**: 527–535. 4

Sayag, T., Fine, S. and Mansour, Y. [2006]. Combining multiple heuristics, *STACS*, pp. 242–253. 1.2, 1.5, 2.4.3, 2.5, 2.7, 3.5, 5, 5.4, 6.4, 7.6, 10.1, 10.2

Schmee, J. and Hahn, G. J. [1979]. A simple method for regression analysis with censored data, *Technometrics* **21**(4): 417–432. 6.4

Schmidhuber, J. [2004]. Optimal ordered problem solver, *Machine Learning* **54**: 211–254. 7.6

Schmidhuber, J., Zhao, J. and Wiering, M. [1997]. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement, *Machine Learning* **28**: 105–130. 5.5

Seeger, M. [2000]. Bayesian model selection for support vector machines, Gaussian processes and other kernel classifiers, *in* S. A. Solla, T. K. Leen and K. R. Müller (eds), *Advances in Neural Information Processing Systems — NIPS*, Vol. 12, The MIT Press, pp. 603–609. 2.3.2

Shonkwiler, E. and van Vleck, E. [1994]. Parallel speed-up of Monte Carlo methods for global optimization, *Journal of Complexity* **10**(1): 64–95. 2.4.2

Soares, C., Brazdil, P. B. and Kuba, P. [2004]. A meta-learning method to select the kernel width in support vector regression, *Machine Learning* **54**(3): 195–209. 2.3.2

Solomonoff, R. J. [2003]. Progress in incremental machine learning, *Technical Report IDSIA-16-03*, IDSIA, Lugano, Switzerland. 5.5

Spierdijk, L. [2005]. Nonparametric conditional hazard rate estimation: a local linear approach, *Technical Report TW Memorandum*, University of Twente. 3.4

Streeter, M. [2007]. *Using Online Algorithms to Solve NP-Hard Problems More Efficiently in Practice*, PhD thesis, Carnegie Mellon University.
URL: `http://www.cs.cmu.edu/~matts/thesis/index.html` 1, 4.6, 6.4, 7.6, 9, 9.2, 9.2, 9.3, 9.3.1, 4, 10.1

Streeter, M. J., Golovin, D. and Smith, S. F. [2007]. Combining multiple heuristics online, *in* R. C. Holte and A. Howe (eds), *Twenty-Second AAAI Conference on Artificial Intelligence*, AAAI Press, pp. 1197–1203. 1.2, 1.3, 2.4.3, 2.5, 2.5, 2.7, 4.5, 5.4, 5.4, 5.6, 6.4, 7.6, 9.9, 10.2, 10.3

Streeter, M. J. and Smith, S. F. [2006]. An asymptotically optimal algorithm for the max k-armed bandit problem., *in* Y. Gil and R. J. Mooney (eds), *Twenty-First National Conference on Artificial Intelligence — AAAI*, AAAI Press. 4.5

Streeter, M. and Smith, S. F. [2008]. New techniques for algorithm portfolio design, *in* D. A. McAllester and P. Myllymäki (eds), *24th Conference on Uncertainty in Artificial Intelligence — UAI*, AUAI Press, pp. 519–527. 2.5, 2.7, 4.5, 6.4, 7.6, 5, 9.3, 9.3.1

Sutton, R. and Barto, A. [1998]. *Reinforcement learning: An introduction*, MIT Press, Cambridge, MA. 4, 4.1, 4.5

Tsiatis, A. [1975]. A nonidentifiability aspect of the problem of competing risks, *Proceedings of the National Academy of Sciences of the USA* **72**(1): 20–22. 3.6, B

Van Keilegom, I., Akritas, M. and Veraverbeke, N. [2001]. Estimation of the conditional distribution in regression with censored data : a comparative study, *Computational Statistics & Data Analysis* **35**: 487–500. 3.4

van Moorsel, A. and Wolter, K. [2004a]. Analysis and algorithms for restart, *Proceedings of the First International Conference on The Quantitative Evaluation of Systems, (QEST'04)*, IEEE Computer Society, pp. 195–204. 2.4.2

van Moorsel, A. and Wolter, K. [2004b]. Optimal restart times for moments of completion times, *in* I. Awan (ed.), *UK Performance Engineering Workshop — UKPEW*, pp. 219–223. 2.4.2

Vapnik, V. [1995]. *The Nature of Statistical Learning Theory*, Springer, New York. 2.3.2

Vilalta, R. and Drissi, Y. [2002]. A perspective view and survey of meta-learning, *Artificial Intelligence Review* **18**(2): 77–95. 2.3.2

Vilalta, R., Giraud-Carrier, C. and Brazdil, P. [2005]. Meta-learning: Concepts and techniques, *in* O. Maimon and L. Rokach (eds), *Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers*, Springer. 2.3.2

Vovk, V., Gammerman, A. and Shafer, G. [2005]. *Algorithmic Learning in a Random World*, 1 edn, Springer. 4

Wang, J. L. [2005]. Smoothing hazard rate, *in* P. Armitage et al. (eds), *Encyclopedia of Biostatistics, 2nd Edition*, Vol. 7, Wiley, pp. 4986–4997. 3.3.2

Whaley, R. C., Petitet, A. and Dongarra, J. J. [2001]. Automated empirical optimization of software and the ATLAS project, *Parallel Computing* **27**(1–2): 3–35. 2.3.1

Wichert, L. and Wilke, R. A. [2005]. Application of a simple nonparametric conditional quantile function estimator in unemployment duration analysis, *Technical Report ZEW Discussion Paper No. 05-67*, Centre for European Economic Research. 3.4, 9.1, 10.1

Wolpert, D. H. and Macready, W. G. [1995]. No free lunch theorems for search, *Technical Report SFI-TR-95-02-010*, Santa Fe Institute, Santa Fe, NM. 1.1

Wolpert, D. H. and Macready, W. G. [1997]. No free lunch theorems for optimization, *IEEE Transactions on Evolutionary Computation* **1**(1): 67–82. 1.1

Xu, L., Hoos, H. H. and Leyton-Brown, K. [2007]. Hierarchical hardness models for SAT, *in* C. Bessiere (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 4741 of *LNCS*, Springer, pp. 696–711. 2.3.3, 2.4, 6.4

Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K. [2007]. SATzilla-07: the design and analysis of an algorithm portfolio for SAT, *in* C. Bessiere (ed.), *Principles and Practice of Constraint Programming — CP*, Vol. 4741 of *LNCS*, Springer, pp. 712–727. 2.3.3, 6.4, 10.3

Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K. [2008]. SATzilla: Portfolio-based algorithm selection for SAT, *Journal of Artificial Intelligence Research* **32**: 565–606. 1.2, 2.3.3

Yu, H., Zhang, D. and Rauchwerger, L. [2004]. An adaptive algorithm selection framework, *13th International Conference on Parallel Architectures and Compilation Techniques — PACT*, IEEE Computer Society, pp. 278–289. 2.3.1