# Learning Parallel Portfolios of Algorithms

Marek Petrik

Diploma Thesis
*Advisor:* RNDr., Ing. Mikulas Popper Ph.D.

Bratislava 2005

*Department of Mathematics Physics and Informatics,*
*Comenius University*

# CONTENTS

## ABSTRACT

Various algorithms for a single problem usually have diverse performance; a single algorithm rarely performs better than all others on all instances. We present an approach that takes advantage of this fact. The Parallel Portfolio of Algorithms (PPA) is a collection of diverse algorithms for a single problem. For solving a problem instance, all algorithms are executed in parallel. We consider both decision and optimization problems.

The amount of computational resources assigned to each algorithm, referred to as a schedule, determines the performance of a PPA. We explore how to create schedules optimal on a specific problem domain. The domain is represented by a training set of randomly chosen instances.

We define a formal framework for learning algorithms that perform well on a training set with regard to three performance measures. The framework requires the algorithms to perform well not only on the training set, but also on the set of all instances. We use it to formalize the learning of PPA.

We also provide a formal framework for representing the schedules of PPAs. We consider two types of schedules, dynamic and static. For static ones, we define a fast but suboptimal iterative algorithm, and a slower but optimal one. Then, we show that dynamic schedules may be represented by a Markov Decision Process and provide algorithms for this case.

Moreover, we explore the generalization capabilities of PPA with both types of schedules. We provide distribution-free bounds for learning schedules with a number of training instances polynomial in precision and certainty.

We evaluated the presented methods on the SAT problem. For this purpose we considered 23 state-of-the-art algorithms. The PPA showed up to a 3-fold speedup compared to the best algorithm, and consistently executed 2-times faster than the best algorithm. As a result, automatically learned PPAs are attractive for applications and future research.

**ACKNOWLEDGMENTS**

I wish to thank my advisor Mikulas Popper for enlightening conversations, guidance, and clarification of my ideas. Moreover, I am indebted to Klaus Dorer, Rastislav Kralovic, Nicoleta Neagu, Dana Pardubska, Jan Sefranek, and Shlomo Zilberstein for their valuable comments for my ideas and work. I especially thank Nicoleta Neagu for reviewing my thesis.

A special thank goes to my family. I wish to thank my parents, Fedor and Mariana, and my siblings, Ivica and Radoslav, for their consistent emotional support. Last, but not least, I want to thank my girlfriend Jana Hapcova, for her encouragements and for being there when I needed her.

# *Chapter 1*

# INTRODUCTION

Time is the most precious resource in the modern world. However, many important problems seem to be successfully resisting to the geometrically increasing power of computers. The aim of our work is to provide a general method for solving the real world problems faster and with better results.

In the following, we use the standard computer science terminology for *problem* and *problem instance*. Problem refers to the general definition, such as Traveling Salesman Problem (TSP). Problem instance refers to the specific instance of the problem; it is a weighted graph for TSP.

## *1.1 Motivation*

Representation of a specific domain is the heart of any intelligent system (Giarratano & Riley, 1998). The most common modeling paradigms for intelligent systems are logics, Bayes networks, constraint and mathematical programs (Horvitz, Breese, & Henrion., 1988; Russell & Norvig, 2002). The main problem of these representation schemes is their *intractability*. As a common feature, intractable representations usually have an accurate and tractable measure of quality of the solution[1]. Depending on the actual domain, it may be the probability of the solution, or its expected utility. For example, discovering optimal solutions is typically NP hard for Bayes networks (Park & Darwiche, 2004), and for integer linear programming (Papadimitriou & Steiglitz, 1998). Moreover, in many application it is crucial to obtain good results in short time.

A common way to solve hard problems is *instance decomposition*. It is, for example, a Hyper-Tree Decomposition (Gottlob, Leone, & Scarcello, 2000) for constraint satisfaction problems, and Dantzig-Wolfe Decomposition (Papadimitriou & Steiglitz, 1998) for linear programming. The main idea it to decompose a problem instance into smaller instances, solve each independently, and recombine them to form a solution. However, because instance decomposition is domain dependent, there are many cases where its application is problematic.

Another approach is *problem decomposition*[2]. Its concept is to decompose the set of all instances into smaller and more homogeneous sets. Each subset may then have its own solver, which specializes only on those instances. Though each solver is specialized for only a sub-problem, it may be actually able to solve problems from other sets. As a result, the whole system[3] is composed of many individual algorithms. A problem instance can then be solved by executing all available algorithms, either in parallel or sequentially.

Generality and good performance on specialized domains is a useful and practical property of algorithms. Such algorithms could be constructed once and used many times in various domains. The increased applicability may lower the development cost of high performance algorithms. Unfortunately, since the information about the application domain is not available during the construction of an algorithm, it cannot be utilized. However, the information usually becomes

[1] This is a feature of all NP problems.

[2] We introduce this terminology because it very well represents our main ideas.

[3] The composition of several algorithms is called either a *portfolio* or an *ensemble algorithm*.

available once the algorithm is deployed; at least by means of solved problem instances. Then, an automatic adaptation of the algorithm to this information may enhance its overall performance.

Reliability is also an important issue for many systems. Problem decomposition can increase reliability of a system, if the individual the subproblems overlap. The highest reliability is achieved if every instance is solvable by all available algorithms. In this case, even if few algorithms crash during solving a problem instance, it can still be solved by the algorithms that work.

## 1.2 PPA : Parallel Portfolio of Algorithms

The aim of this thesis is to present methods of effective composition of several algorithms for a single problem. Unlike in other research on algorithm composition, we do not consider having the ability to determine the best algorithm for a specific instance. Though, some of the proposed models allow incorporation of this option to increase their performance.

The collection of algorithms is usually referred to as *algorithm portfolio*. Since all algorithms are launched in parallel, when solving a problem instance, we call this kind of portfolio as *parallel*. Because most computer systems are inherently sequential, the parallel execution can be achieved by interleaving the execution of the algorithms in small time steps. This approach is widely used and supported by operating systems.

We consider portfolios of both decision and optimization algorithms. For optimization algorithms, we require anytime algorithms (Zilberstein, 1993). Anytime algorithms may be interrupted before finishing the calculation, but should still return at least a partial solution.

In real-life computations, many resources act as bottlenecks. Usually, it is processor time, memory space, communication throughput. We consider only the case when processor time is the limiting factor of the computation. The efficiency of PPA is controlled by distributing the available processor time to algorithms. This distribution is determined by a *schedule*, which is optimally calculated *offline* from a training set of problem instances. We use several measures of optimality, such as the expected time complexity, maximal time complexity, expected cost of solutions, and other metrics of algorithm performance.

To allow feasible computation of optimal schedules, as well as their practical implementation, we limit the set of available schedule choices. First, we consider schedules that assign constant resource share to all algorithms. These are called *static schedules.* Second, we consider ones that run only a single algorithm at a time, and can change it in discrete, equally sized, intervals. These are called *dynamic schedules.*

For decision problems, the utility of a PPA is based on the time needed to solve the instances. For optimization problems, we assume a fixed amount of available time. Then, the utility is based on the quality of the solutions.

A drawback of the approach is the requirement to create multiple algorithms for a single problem. It is not an issue in problems where many standard algorithms already exist. Such problems are, for instance, constraint programming, and mathematical programming. In addition, broader usability of the portfolio algorithm compensates the initial effort. Since multiple algorithms require more memory than a single one, the approach is not suitable for memory-limited applications.

## 1.3 Sample PPA

In many situations, PPAs are helpful because usually no algorithm dominates all others (Tsang, Borrett, & Kwan, 1995). It makes sense not only to compose principally different algorithms, but also to compose two instances of the same algorithm with different parameters, or heuristics. A standard approach to learn a composition of heuristics is to express the compound heuristics as a linear composition of the partial ones (Russell & Norvig, 2002; Sutton & Barto, 1998b). The

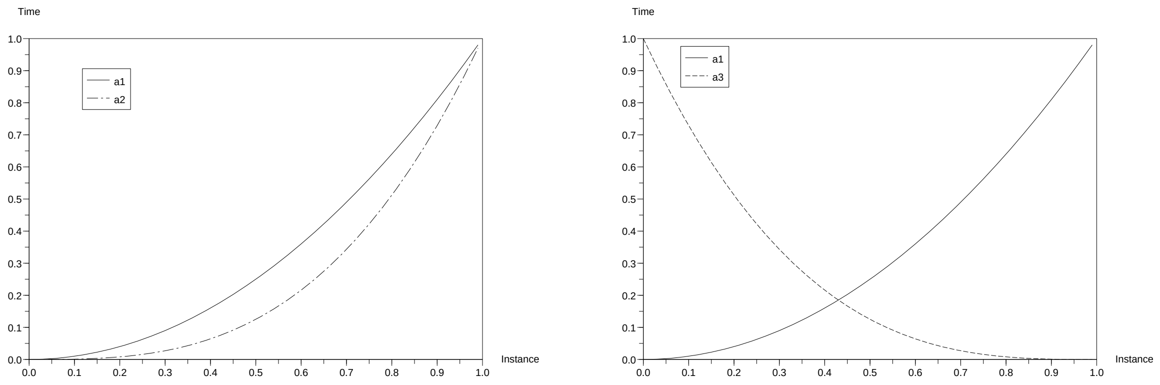

***Figure 1.1:*** *Comparison of complexity functions of algorithms $a_1$, $a_2$ and $a_3$.*

PPA is based on separating these heuristic functions into independent algorithms. This may be valuable when the heuristic functions are not correlated.

In some cases, even several instances of one algorithm may be composed into a parallel portfolio. Each algorithm can search only a part of the whole problem instance, obtained by adding additional constraints to it. Ideally, these constraints should define a different search space for each algorithm, and cover it whole.

For a simple illustration when a PPA may be useful, consider the following simplistic toy problem setup. It intends to show the benefit of diversity of algorithms. Let the problem consist of set $\mathcal{I}$ of equally probable instances, and three algorithms $a_1$, $a_2$, and $a_3$. We will show that in this case, even a PPA with a naive schedule, outperforms each algorithm with regard to expected solution time.

Then, assume that there is a bijection $b : \mathcal{I} \rightarrow \langle 0, 1 \rangle$ mapping a problem instance to a real number[4]. Each instance may be experienced with equal probability. Because $b$ is a bijection to a real domain, the instance set must be infinite. Let the complexity functions of the available algorithms, with regard to instances, be:

$$\begin{aligned} t_{a1}(x) &:= b(x)^2 \\ t_{a2}(x) &:= b(x)^3 \\ t_{a3}(x) &:= (1 - b(x))^3. \end{aligned}$$

The complexity graphs are shown on Fig. 1.1.

Average-case time complexities for algorithms are equal to their expected calculation times on all problem instances. The complexities are

$$\begin{aligned} \mathbf{E}\left[t_{a1}(x)\right] &= 0.33 \\ \mathbf{E}\left[t_{a2}(x)\right] &= 0.25 \\ \mathbf{E}\left[t_{a3}(x)\right] &= 0.25. \end{aligned}$$

Consider algorithm $c_1$ as a PPA of $a_1$ and $a_2$. Then consider an algorithm $c_2$ as a PPA of $a_1$ and $a_3$. The schedule assigns equal resources to each algorithm. Therefore, the appropriate complexity functions are

$$\begin{aligned} t_{c1}(x) &= \min\left\{2 * t_{a1}(x), 2 * t_{a2}(x)\right\} \\ t_{c2}(x) &= \min\left\{2 * t_{a1}(x), 2 * t_{a3}(x)\right\}. \end{aligned}$$

[4] This assumption is for most cases an unreasonable one; we present it only for sake of clarity. Moreover, the presented approach is based on the bijection not being known.

Average-case time complexities of algorithms $c_1$ and $c_2$ are

$$\begin{aligned}\mathbf{E}[t_{c1}] &= 0.5\\ \mathbf{E}[t_{c2}] &= 0.11,\end{aligned}$$

and are depicted in Figure 1.2.

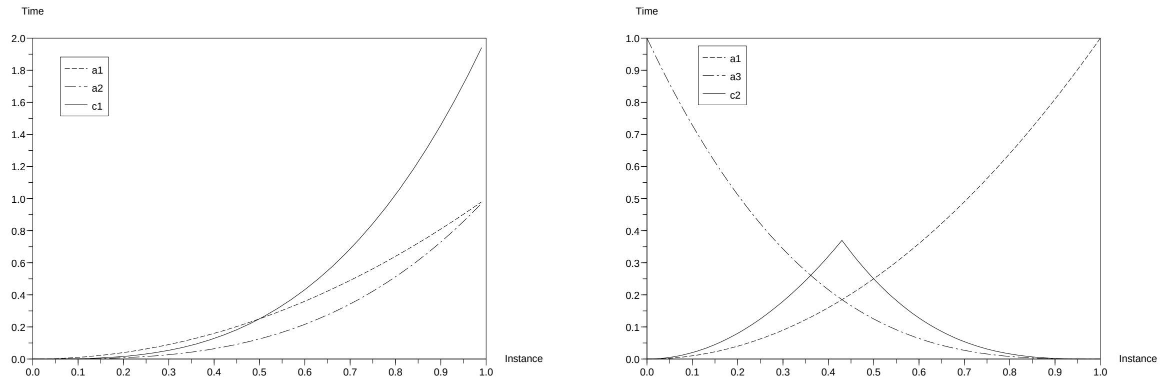

***Figure 1.2:*** *Comparison of complexity functions of $c_1$ and $c_2$ to the regarding atomic algorithms $a_1$, $a_2$, and $a_3$.*

It is notable that algorithm $c_1$ performs worse than both algorithms $a_1$ and $a_2$. On the other hand algorithm $c_2$ outperforms both $a_1$ and $a_3$. This illustrates why parallel portfolio of several algorithms whose complexity functions are anti-correlated leads to an algorithms outperforming both former ones.

In this case, we did not tune the algorithm shares to optimize the time, we only considered a portfolio with a naive schedule. In the following, we show how to calculate the assignments and demonstrate the usefulness on a practical example.

## *1.4 Summary*

This section outlines the main claims of the thesis and its organization.

### *1.4.1 Thesis Statement*

The thesis explores different possibilities of creating Parallel Portfolios of Algorithms. Most notably, it presents the theoretical foundations of finding optimal static and dynamic schedules. In addition, since the schedules are calculated from training instances, the thesis provides a light exploration of the issue of generalization. Finally, the PPAs are evaluated on a wide-spread artificial intelligence problem, the satisfiability problem (SAT). This work builds on and significantly extends the ideas and approaches from (Petrik, 2005).

### *1.4.2 Organization*

The remainder of the thesis is organized as follows.

*Chapter 2* Summarizes relevant work from related fields of machine learning, optimization, and algorithm portfolios.

*Chapter 3* Defines a model how the sample instances can be used in optimal schedules, and what optimality types of schedules are relevant.

*Chapter 4* Defines a formal model to capture the basic properties of parallel portfolios. It defines useful features of algorithms, used later during the analysis.

*Chapter 5* Analyzes how to find static schedules for PPA, that is, schedules in which the resource allocation for each algorithm is constant during the computation.

*Chapter 6* Extends the results from the preceding chapter to find optimal schedules, and shows a nice visualization scheme for PPA performance.

*Chapter 7* Analyzes dynamic schedules, that is, schedules in which the resource allocation may change during the calculation.

*Chapter 8* Shows theoretical, distribution-free generalization bounds for PPA. It is based on the formal learning framework from Chapter 3.

*Chapter 9* Provides experimental results of PPA applied to solving one of the most frequent problem in AI - Satisfiability problem.

*Chapter 10* Summarizes the thesis and provides ideas for future work.

*Appendix A* Summarizes the basic notation used in the thesis.

*Appendix B* Contains a short overview of the mathematical background for terms used in the thesis.

*Appendix C* Provides detailed numerical results of test runs of PPAs on the SAT problem.

*Appendix D* Contains implementation of calculation of optimal PPAs used for SAT application.

# *Chapter 2*

# RELEVANT WORK

In this section, we introduce related approaches of composing various kinds of algorithms. Though the idea is not new, there has not been much research done in this area. Our approach is also related to learning to solve problems and therefore to concept machine learning. Therefore, we present relevant research in machine learning too.

## *2.1 Learning to Solve Problems*

The motivation behind learning to solve problems is the same as behind concept machine learning. However, the problem is usually much harder, because the learned programs must contain advanced heuristics and solution templates to be competitive with algorithms developed by humans. Therefore, the main research concerns minor modifications of available algorithm skeletons. The main goal is to get an empirically fastest algorithm with regard to a given distribution. This problem is generally known as *speedup learning.*

Although speedup learning has not been as widely researched as concept learning, there have been many successful applications (Finkelstein & Markovitch, 2001, 1998; Gratch & Chien, 1996; Gratch & DeJong, 1996; Minton, Allen, Wolfe, & Philpot, 1995). Some of them are also related to *reinforcement learning* (Sutton & Barto, 1998a).

### *2.1.1 Explanation Based Learning*

Explanation based learning (EBL) is a type of concept machine learning in logic domain. Its main goal, unlike in most other machine learning approaches, is not to learn a new concept, but only to learn a concept that can be deducted the domain. This is useful in cases, when extracting the concepts from the domain is hard. For example, take the game Chess. Most people know the definition of the domain, but find hard to say whether for instance white can win in 6 moves. Therefore, it is expected that by adding the new rules, the domain can become more tractable for reasoning[1]. EBL takes the training examples, explains why they are true and add the corresponding rules. For more details see (Mitchell, 1997; Russell & Norvig, 2002).

The most widely used variant, *Explanation Based Generalization*, works on domains specified by first order logic. It finds the most general preconditions for each sample, and adds the rule classifying the sample based on these precodition. This approach can be extended to learning *search control* knowledge for general problem solvers, and planners.

PRODIGY, a successful rule based planner, can be extended by EBL to learn control knowledge (Gratch & DeJong, 1996). Also, a similar planner, SOAR, uses an EBL method called *chunking* to learn control rules (Laird, Rosenbloom, & Newell, 1986). EBL was also successfully used to learn control knowledge for partial order planners (Estlin, 1998).

An inherent problem of EBL learning is that the number of control rules may increase to the point where they actually slow the system down. Moreover, a rule that is speeds solving

[1] Similar approach is taken by Branch and Cut, and pre-solving of mathematical programming.

of a single instance is not necessarily beneficial for solving others and therefore may also slow the whole system down (Etzioni & Minton, 1992). These problems have were quite successfully addressed by COMPOSER (Gratch & Chien, 1996). COMPOSER adds new rules only when they show a statistically significant speedup. It was applied quite successfully on a NASA scheduling problem (Gratch & DeJong, 1996).

A slightly different approach was taken by MultiTAC (Minton et al., 1995; Minton & Underwood, 1994). It takes a problem specified in a general meta-language and a generator of typical instances. Then, it tries to apply various representation schemes and solution methods to obtain the best performing algorithm on the domain. It uses an algorithm inspired by EBL to specialize the general methods for the provided typical instances. In a case study, MultiTAC outperformed a Ph.D.-level computer scientist in designing CSP algorithms (Minton, 1993).

### 2.1.2 Formal Models

One of the most fundamental problems of machine learning is generalization, and it also applies to speedup learning (SL). It is even more vital for SL, because each sample instance must be solved what is in most cases more time-consuming than labeling instances in concept learning. Therefore, the training sets are generally smaller.

While much effort has been spent on inventing suitable models for concept learning (Mitchell, 1997), there have been only few attempts to formalize speedup learning. The most well-known one is (Tadepalli & Natarajan, 1996). It is inspired by PAC learning model, the most common model for evaluating concept learners.

Though the framework proposed by (Tadepalli & Natarajan, 1996) is interesting and applicable in many cases it is not suitable for our purpose, because it strictly limits the class of algorithms. Basically, it requires the algorithms to solve almost all problems in polynomial time. Another interesting framework, similar in spirit, was proposed in (Tadepalli, 1992). It considers a batch problem solving, that is a framework where the complexity is considered with regard to a set of instances.

## 2.2 Composition of Classifiers

Most of the work on composition of classifiers deals with binary classification in context of machine learning. The result of the *ensemble classifier* is determined by letting the individual classifiers vote on the result. In simpler schemes, the training samples are used to determine weights of the available classifiers. More advanced schemes also learn new classifiers based on the performance of the current ones.

### 2.2.1 Composition of Existing Classifiers

Prediction with expert advice deals with predicting an output given a set of experts. Each expert can vote on the output. Usually, the majority decides the result. The votes are weighted, and main concern is to calculate the voting weights to achieve the best overall prediction.

The simplest forms of prediction with expert advice are *halving* and *weighted majority* algorithms. Halving algorithm uses all available classifiers to vote on the label with equal weight. If the predicted label is incorrect, the classifiers with invalid input are discarded. The principal disadvantage of halving is high sensitivity to noise. Weighted majority algorithm works similarly, except it only decreases the voting weight of classifiers with incorrect prediction. Both schemes are used in the online learning framework (Kearns & Vazirani, 1994).

More involved prediction schemes were proposed in (Cesa-Bianchi, Freund, Helmbold, Haussler, Schapire, & Warmuth, 1993; Hutter & Poland, 2004; Jacobs, Jordan, Nowlan, & Hinton, 1991). Some related approaches are the expert mixture modeling (Meek, Thiesson, & Heckerman, 2002) and composition of information retrieval techniques (Arnt, Zilberstein, & Allen, 2004).

### 2.2.2 Composition of New Classifiers

A more flexible approach than taking a set of existing classifiers, is to learn new ones during the composition, according to the actual training instances. This is equivalent to composing infinite number of existing classifiers. The approach is generally referred to as *ensemble learning* (Meir & Ratsch, 2003). Depending on the method used for construction of the ensemble, the approach is either *bagging*, *boosting*, or *leveraging*.

The simplest approach, bagging, randomly draws subsets of the training set. These subsets are then used to train classifiers that comprise the ensemble classifier. In many cases bagging and its extensions increased the stability of classifiers, and increased their precision (Lee & Clyde, 2004; Oza & Russell, 2001; Chawla, Hall, Bowyer, & Kegelmeyer, 2004).

The most successful ensemble learning approach is boosting, represented mainly by AdaBoost, originated by (Freund & Schapire, 1996). AdaBoost cyclically adds new classifiers to the ensemble. The training set is always reweighed with regard to the performance of the past classifiers. It assigns the weight to each classifier according the its performance on the training set. The individual classifiers that are used are usually either simple neural networks or decision stumps[2]. The main advantage of boosting it its high generalization capability.

The main reasons for the success of boosting are not completely understood, but the common assumption is that it is because it maximizes the margin(Schapire, Freund, Bartlett, & Lee, 1997). Maximum margin classifiers are resistant against overfitting that plagues most learning algorithms. There is vast number of other publications analyzing the success of boosting (Friedman, Hastie, & Tibshirani, 2000; Kivinen & Warmuth, 1999; Shawe-Taylor & Cristianini, 1998). Leveraging is a generalized boosting (Meir & Ratsch, 2003).

A related area is the learning in multi-agent systems (MAS). However, there is usually no central control in MAS and the communication is limited (Stone & Veloso, 2000; Lander & Lesser, 1994; Lesser, 1999). Therefore, the emphasis of research is mainly on coordination among agents.

## 2.3 Composition of Algorithms

Unlike for composing binary classifiers, for algorithms it does not make sense to vote on the result. Even when many results are available, only one can be used. This fact makes the composition of algorithms fundamentally different from the composition of classifiers. As a result, the approaches developed for composing classifiers cannot be directly used for composing algorithms.

### 2.3.1 Algorithm Portfolios

As mentioned before, even a good algorithm on a problem usually does not outperform all other algorithms on all instances. Occasionally, instances that are virtually unsolvable by an overall superior algorithm are solved with seconds by a generally inferior one. Therefore, a prior knowledge of the algorithms' performance on an instance may enhance the performance, by running the most suitable algorithm for each instance. This is usually impossible to decide without actually solving the instance. However, the performance may determined from its similarity to instances with known performance. This way, easily-computable features of the instance may help to approximately determine the performance of each algorithm on it. The main obstacle is to decide whether computation of an instance feature will enhance classification precision. This problem is known as the *algorithm portfolios* (Gomes & Selman, 2001).

[2] Decision stump is a decision tree with only one decision node.

(Gomes & Selman, 2001) provides an empirical evaluation of composing randomized search algorithms on more processors into portfolios. Their results indicate that it is profitable to combine randomized algorithms with high variance for multiple processors. For a single processor, they found random restarts to be the optimal strategy.

A successful algorithm portfolio selection approach was also taken by (Leyton-Brown, Nudelman, Andrew, McFadden, & Shoham, 2003). It is based, as well as most of other approaches, on using a training set of problem instances to estimate a probabilistic model of the performance. The authors used statistical regression to determine the best algorithm for instances of combinatorial auction winner determination.

Concurrent use of various algorithms for solving one instance of a problem was proposed in *asynchronous teams* (A-teams) (Talukdar, Baerentzen, Gove, & de Souza, 1998; Cicirello, 2003). A-team is a set of memories and autonomous agents that asynchronously work on the results of others. Each agent can read and modify the content of memories. Even each memory may use a different representation of the problem. In fact, A-team is an extension of PPA, except the resources for algorithms are fixed, and the algorithms can communicate during the solving process. One of the prominent issues in the approach is to determine the optimal communication order. (Baerentzen & Talukdar, 1997) shows that the problem can be formulated as a Partially Observable Markov Decision Process. According to (Talukdar et al., 1998), A-teams appear to be a good framework for combining heterogeneous algorithms and formulations.

### *2.3.2 Asymptotically Optimal Methods*

A different approach to algorithm composition is to run all available algorithms in parallel. It can be shown that given an infinite number of algorithms, each problem instance is solved asymptotically as fast as would be solved by the fastest algorithm from the set. The method that is used for problems with quickly verifiable solutions is *Levin Search* (Levin, 1973; Schmidhuber, Zhao, & Wiering, 1997). It was later extended to all well-formulated problems in (Hutter, 2001) and thus is called Hutter Search. (Schmidhuber et al., 1997) proposed an extension of LS that uses instances encountered during previous solutions to determine how much resources should be allocated to each algorithm. Optimal Ordered Problem Solver (Schmidhuber, 2002) incrementally grows programs to solve arbitrary problems. It starts with simple problems instances, and as it proceeds to solve harder ones, it uses the previously learned knowledge. The approach was successfully applied to problems unsolvable by traditional reinforcement learners, such as Towers of Hanoi with 30 disks. A somewhat simpler extension, called Adaptive Levin Search, was proposed in (Schmidhuber & Wiering, 1996).

# *Chapter 3*

# FORMAL MODEL: LEARNING

In this chapter we propose a framework for *speedup learning* that we later use to find the optimal PPAs and to determine their generalization capability. We use the term time optimization for learning fast decision algorithms and quality optimization for learning optimization algorithms that produce results with high utility. Sometimes, we use the term speedup learning to refer to both these learning types.

## *3.1 Models*

The model we propose is similar to the standard PAC learning model used in concept learning (Mitchell, 1997). Since we assume that the training instances are randomly drawn from the training set, we allow a probability that the optimal algorithm is not learnable due to invalid sample. The following definitions formalize the notions we later use for characterization of the learning problems.

**Definition 3.1.1.** A *problem domain* is a set of possible problem instances together with a probabilistic distribution over them, we denote this set as $\mathcal{I}$. The distribution over the samples must be *stationary.*

**Definition 3.1.2.** An *algorithm* is a function that maps an input in form of a problem instance to a solution. Therefore, it must be deterministic.

**Definition 3.1.3.** An *algorithm domain* is a set of the all possible algorithms that can be used. We denote this set as $\Phi$. The set may alternatively denote the set of profit functions of these algorithms.

**Definition 3.1.4.** A *profit* is a value from $\mathbb{R}$ that indicates the utility of the solution. It may represent either the time required to reach the solution, or its quality. All profits in $\Phi$ must be either non-negative or non-positive.

The profit is abstract at this point to cover both time and quality optimization. As mentioned in Chapter 1, the profit of decision algorithms is based on time an algorithm needs to solve to problem. As well, for optimization algorithms it is based on the quality of result given a fixed amount of time. We provide its specific meaning for PPA in the next chapter.

**Definition 3.1.5.** *Profit function* $f : \mathcal{I} \to \mathbb{R}$ represents the profit of an algorithm on every problem instance.

**Definition 3.1.6.** *Training set* $I$ is a finite set of instances chosen randomly and independently from $\mathcal{I}$.

**Definition 3.1.7.** A *learner* takes a training set from problem domain and outputs an algorithm from $\Phi$ that fulfills some conditions.

Measure of usefulness of an algorithm is crucial for development of a learner. Thus, we consider the following three variants of learning.

*Mean Optimization* $\sup_{f \in \Phi} \mathbf{E}[f(X)]$

*Limit Optimization* $\sup_{f \in \Phi} \inf_{x \in \mathcal{I}} f(x)$

*Bound Optimization* $\sup_{f \in \Phi} \mathbf{P}[f(X) \geq B]$ where $B$ is a boundary value, which is provided.

We discuss reasons for each variant in the subsequent sections. Their actual application to PPA learning is the content of the subsequent chapters.

An interesting question is to determine the optimal algorithm on $\mathcal{I}$, given only $I$. In addition, it is important to analyze the number of samples required in order to be certain that the algorithm will perform well on all instances. Because, we assume that the instances are randomly chosen from the distribution, it is not always possible to learn a good algorithm. To address this issue, we allow the learner a probability of failing to learn an overall optimal algorithm. In the following, the instances from the training set $I$ will be denoted as $x_1, \dots, x_m$, where $m = |I|$. $X$ is a random variable that represents an instance from $\mathcal{I}$.

## 3.2 Mean Optimization

Finding an algorithm that is optimal with regard to expected execution time is useful when many problem instances must be solved. However, the model does not provide any assurance about profit on any particular instance. The performance measure of an algorithm on the training set is its *empirical mean profit*, defined as:

$$F_m(f) = \frac{1}{m} \sum_{i=1}^{m} f(x_i).$$

The following definition provides a bound for the overall mean profit of a learned algorithm, given its empirical mean profit.

**Definition 3.2.1.** We say that a learner *mean-generalizes* if for any $0 < \epsilon$ and $0 < \delta < 1$ it outputs an algorithm $f \in \Phi$, for which

$$\mathbf{P}[F_m(f) - \mathbf{E}[f(X)] > \epsilon] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

Even more important is whether the learner can output an algorithm that is close to the optimal one on $\mathcal{I}$. This is problematic, because it can use only the training set, which is typically very small compared to all instances. The following definition addresses this concern.

**Definition 3.2.2.** Let the globally optimal algorithm be

$$f^*(X) = \arg\sup_{g \in \Phi} \mathbf{E}[g(X)].$$

We call a learner *mean optimal* if it for all $0 < \epsilon$ and $0 < \delta < 1$ outputs an algorithm $f$

$$\mathbf{P}[\mathbf{E}[f^*(X)] - \mathbf{E}[f(X)] > \epsilon] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

## 3.3 Limit Optimization

In some cases, it is practical to optimize the worst case of the profit. This issue is also addressed by the worst-case theoretical analysis. However, those bounds are usually asymptotic and thus usually impractical. They also do not consider the specific features of the implementation. We calculate the worst case empirically from sample runs of the algorithm. To cope with the incompleteness of the training set, we consider mostly probabilistic bounds.

The measure of algorithm performance of a training set is *empirical minimal profit* obtained as

$$G_m(f) = \min_i f(x_i).$$

We proceed in the same manner as for mean optimization. Definition 3.3.1 addresses the issue of generalization of a learned algorithm. Definition 3.3.2 compares the performance of a learned algorithm from the training set to the algorithm optimal on $\mathcal{I}$.

**Definition 3.3.1.** We say that a learner *limit generalizes* if for any $0 < \epsilon < 1$ and $0 < \delta < 1$ it outputs an algorithm $f \in \Phi$, for which

$$\mathbf{P}\left[\mathbf{P}\left[f(X) < G_m(f)\right] \geq \epsilon\right] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

**Definition 3.3.2.** Let $f^*$ be the optimal algorithm, chosen according to

$$f^* = \arg\sup_{g\in\Phi}\inf_{x\in I} g(x).$$

We say that a learner is *limit optimal*, if for all $\epsilon$ and $\delta$, the learned algorithm's profit function $f$ fulfills

$$\mathbf{P}\left[\mathbf{P}\left[f(X) < \inf_Y f^*(Y)\right] \geq \epsilon\right] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

## 3.4 Bound Optimization

In many cases, problem instances need to be solved with at least a minimal profit. This is particularly useful if the profit indicates time required to solve a problem instance. Then, a problem instance is usually considered as unsolvable when it is not solved in a specified time frame. Therefore, we search an algorithm that minimizes the probability of the algorithm obtaining a profit smaller than a given bound.

The measure of performance of an algorithm on a training set is *empirical failure profit*, defined as:

$$H_n(f) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{I}\left\{f(x_i) \leq B\right\}.$$

We proceed in the same manner as for mean and limit optimization. Definition 3.4.1 addresses the issue of generalization of a learned algorithm. Definition 3.4.2 compares the performance of a learned algorithm from the training set to the algorithm optimal on $\mathcal{I}$.

**Definition 3.4.1.** We say that a learner *bound-generalizes* if for any $0 < \epsilon < 1$ and $0 < \delta < 1$ it outputs an algorithm $f \in \Phi$, for which for a given $B$

$$\mathbf{P}\left[\mathbf{P}\left[f(X) \leq B\right] - H_m(f) \geq \epsilon\right] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

**Definition 3.4.2.** Let the globally optimal algorithm be

$$f^* = \arg\sup_{g \in \Phi} \mathbf{P}\left[f(X) > B\right].$$

Then, let $f$ be an algorithm the learner. We say that a learner is *bound optimal* if it outputs an algorithm $f$ that fulfills

$$\mathbf{P}\left[\mathbf{P}\left[f(X) < B\right] - \mathbf{P}\left[f^*(X) < B\right] > \epsilon\right] \leq \delta.$$

The learner must use at most a polynomial number of training instances in $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$ to achieve the bound.

## *3.5 Summary*

In this chapter, we defined a framework for learning optimal decision and optimization algorithms with regard to three practical measures. The framework considers the capability of a learner to find an algorithm that performs on the whole set comparably to its performance on the training set. This capability is called, according to the measure of optimality, as mean-generalize, limit-generalize, and bound-generalize. In addition, it takes into account whether a learner outputs an algorithm close to the optimal on the whole set. Such learners are mean-optimal, limit-optimal, and bound-optimal. We later use this framework to learn empirically optimal, or close to optimal, PPAs. Then, we analyze their generalization ability.

# *Chapter 4*

# FORMAL MODEL: PARALLEL PORTFOLIO OF ALGORITHMS

In this chapter, we propose a formal model for creating PPAs. It builds upon the learning framework from the previous chapter.

## *4.1 Problems and Algorithms*

Practically encountered problems are usually of two types, decision or optimization. Decision problems ask whether there is a solution with given characteristics. An example is a Traveling Salesman Problem (TSP) when the question is whether there is a tour with a cost smaller than $C$. Optimization problems usually have many acceptable solutions, some of them are more preferable to others. The preference is most often represented by a cost function. Solutions with lower costs are preferable. Therefore, the optimal solution is the one with the minimal cost. For instance, finding a tour with the smallest cost in TSP is an optimization problem.

As in the previous chapter, we first define a general model and then specialize it to needs of either of optimization or decision problems. However, optimization and decision algorithms may not be combined within a single PPA. By Definition 3.1.2, we consider only deterministic algorithms. Possibilities of extension to stochastic algorithms are mentioned in Chapter 10. Additional assumptions placed on algorithms are defined in the following.

**Definition 4.1.1.** An algorithm is *preemptable* if it may be suspended and resumed at any time with a neglectable cost of processor time. Therefore, preemptable algorithms can be interleaved with no extra cost.

### *4.1.1 Decision Algorithms*

As mentioned before, the time to solve a problem is the crucial measure for decision algorithm. Notice that any optimization problem can be formulated as a decision one. The formulation may be either "Is there a solution that has quality above $q$?", or "Is the solution within $p$% of the optimal one?". The latter formulation is not usually possible, because the optimal quality is not known. However, some algorithms, such as Primal-Dual algorithm for linear programming, calculate also the range for optimal result (Papadimitriou & Steiglitz, 1998).

To simplify the creation of schedules we will require the decision algorithms to fulfill some conditions. These are summarized in the following definition of schedulability. This will be the universal criterium for creation of all types of

**Definition 4.1.2.** An decision algorithm is *schedulable* if it is preemptable and its use of processor can be adjusted continuously without any penalty.

### *4.1.2 Optimization Algorithms*

The term *anytime algorithm* was first introduced by (Dean & M.Boddy, 1988). They used the term to leverage from the standard definition of algorithm, that once started runs until it finds the solution. The available time for problem solving is often limited, therefore the used algorithms need to be interruptible at any instant it is needed. However, if interrupted before the expected time, at least a partial result should be returned.

**Definition 4.1.3.** An algorithm is *interruptible* if it can be interrupted any time, and it returns an answer[1]. An algorithm has a *well-behaved improvement* if the answers it returns are a well-defined function of time.

**Definition 4.1.4.** An algorithm is said to be *anytime* if it is preemptable, interruptible, and has a well-behaved improvement.

The meaning of anytime is in sense of (Zilberstein, 1996). The anytime assumption is reasonable, because such algorithms exist for variety of problems (Wallace & Freuder, 1995).

**Definition 4.1.5.** An optimization algorithm has *diminishing returns* if the function of its return qualities over time is concave, monotonically non-decreasing, and twice continuously differentiable[2].

**Definition 4.1.6.** An optimization algorithm is *schedulable* when it is anytime, has measurable quality, and diminishing returns.

## 4.2 PPA Model

First, let us define a formal model that represents the available algorithms and problem instances. We use this model to derive the methods to find PPA with greatest profit in sense of mean, limit, or bound optimization. All algorithms comprising a PPA must be schedulable. Decision algorithms in terms of Def. 4.1.2 and optimization in terms of Def. 4.1.6.

Given a PPA, its performance may be tuned by assigning different portion of the constrained resources. In our case, this constraint is the processor, e.g. the number of available operations. The amount of resources assigned to an algorithm at each point of time is determined by the resource allocation function. Resource allocation functions for all algorithms in the portfolio comprise the schedule. Hence the following definitions.

**Definition 4.2.1.** The *resource allocation function* (RAF) for an algorithm $e$, $r_e : \mathbb{R}_+ \to \langle 0, 1\rangle$ defines its share of processor power in each instance of time during a calculation.

We extend the notion of *profit* from the last chapter to be determined not only by an algorithm and an instance but also by its resource allocation function. We deliberately do not specify the profit function at this point, to keep the model general enough.

**Definition 4.2.2.** The *resource profit function* of an algorithm $e$ is a function[3]

$$p_e : r_e \times I \to \mathbb{R},$$

where $I$ is the set of instances.

**Definition 4.2.3.** A *Parallel Portfolio of Algorithms* $\mathcal{P}$ is a tuple $(I, E)$. $I$ is a finite set of training instances. $E$ is a finite set of algorithms with resource profit functions.

The symbols that are not specifically defined, represent the following quantities:

[1] Preemptable algorithm does not have to return anything when interrupted.
[2] The standard definition does not require continuity.
[3] With a RAF as a parameter.

$m$ Number of instances, generally indexed using $i$.

$n$ Number of algorithms, generally indexed using $j$.

$x_i$ Problem instance.

$p_j$ Profit function for the algorithm $e_j$.

Other symbols, frequently used in the thesis, are summarized in Appendix A.

**Definition 4.2.4.** A Schedule $S$ for a PPA $\mathcal{P}$ is a list of resource allocation functions for algorithms from $E$. Each *resource allocation function* corresponds to one profit function and is defined $\mathbb{R}_+ \to \mathbb{R}_+$. Moreover, all functions must fulfill the resource limitation constraint

$$\sum_{j=0}^{n} r_j(t) \leq 1 \text{ for } t \in \mathbb{R}_+.$$

As an intrinsic characteristics of PPA only one result for the problem may be used, even if more algorithms provide solutions. Therefore, the PPA profit is the maximal value of profits of individual algorithms. This is in contrast with ensemble learning, where classifiers vote and the ensemble result is sum of votes. The max operator makes the analysis harder, because it introduces discontinuity to the optimization problem defined for static schedules.

**Definition 4.2.5.** *Profit* $P(S,x)$ of a PPA is equal to the maximal value of profits that its algorithms achieve on a given instance. Mathematically, it is:

$$P(S,x) = \max_{j=1,\dots,n} p_j(r_j, x).$$

*Remark* 4.2.6. In the next, we use $P$ also for mean and limit optimization. In addition, we vary the parameters according to the situation. If the value of parameters is not clear from the context, then it is explicitly clarified.

**Definition 4.2.7.** A schedule is $\mathcal{B}$*-optimal* if it maximizes the empirical $\mathcal{B}$-profit[4] of PPA on all available instances with regard to the operator $\mathcal{B}$ that is applied on the set of profit function on instances.

Notice that in case of finding optimal schedules, the set of available algorithms $\Phi$, as defined in Def. 3.1.2, is composed of resource weighted portfolios of algorithms from $E$. This results into a theoretically infinite set of possible algorithms, created from a finite one. This is similar to the concept mixed strategies, versus pure strategies in game theory (Osborne & Rubinstein, 1994).

### *4.2.1 Profit Function*

We introduced the concept of profit function to allow a single analysis for both optimization and decision algorithms. For *decision algorithms*, we assume that the profit is achieved by finding a solution in the shortest possible time. The quality of the solutions are disregarded, because all solutions are equally worth. In contrast, for *optimization algorithms*, the profit is achieved by finding a solution with greatest quality. For sake of simplicity, in this case the total available time is fixed.

Notice that for optimization problems, where the provided time is fixed, it is only important that the algorithms receive the processor time share, and it is irrelevant when. Therefore,

[4] That is either mean, minimal, or failure, depending on the choice of operator.

though a RAF defines amount as a continuous share, it does not have to obeyed for the actual implementation. The performance should be identical.

Time optimization is used in case of decision algorithms. The actual profit function for static schedules is provided in the following section. Because the time is unlimited, value of resource allocation functions is relevant in the whole time interval from 0 to $\infty$. The instances that are not solved within a reasonable time are regarded as having a minimal profit.

Quality optimization is used in case of optimization algorithms. There is no extra profit for achieving the optimal solution. Therefore, the profit function depends on the quality of a solution at the final time, not on the time when the solution was obtained.

### *4.2.2 Optimality Operator*

The optimality operator $\mathcal{B}$ defines how the schedules are compared, given their profits on a set of training instances. There are various possible learning objectives. We do not yet specify the operator to allow joint analysis of more objectives. We consider the optimization types specified in Chapter 3, and they determine the operator $\mathcal{B}$ as:

*Mean optimization* $\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} P(x_i, S)$.

*Limit optimization* $\mathcal{B} = \min_{i=1,\ldots,m} P(x_i, S)$.

*Bound optimization* $\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} \mathrm{I}\{P(x_i, S) > B\}$.

## *4.3 Static Schedules*

Static schedules allow only constant resource allocation functions. While this certainly is a limitation, most experimental results do not show a significant decrease of performance compared to the more complex dynamic schedules. The main reason to use static functions is to decrease the complexity of choosing the optimal schedule.

**Definition 4.3.1.** A schedule is *static* when all resource allocation functions are constant in time.

Because RAF are constants in static schedules, we do not use the time parameter in their notation. Also, profit functions of algorithms may be defined in term of these constants. Hence the definition

**Definition 4.3.2.** The resource profit function for static schedules may be redefined as

$$p_e : \mathbb{R}_+ \times I \to \mathbb{R}_+.$$

In other words, it only considers real values for profit, not functions.

According to Def. 4.2.5, the profit of the PPA on an instance $x$, given a schedule $S$ is:

$$\max_{j=1,\ldots,n} p_j(r_j, x).$$

Therefore, performance on the whole set of training samples will be

$$\mathcal{B}\left\{ \max_{j=1,\ldots,n} p_j(r_j, x_i) \,\middle|\, i = 1, \ldots, m \right\}.$$

The optimal schedule is one that maximizes the ensemble profit. It can be formulated as a nonlinear mathematical program:

$$\begin{aligned} \text{maximize} \quad P(S) &= \mathcal{B}\left\{ \max_{j=1,\dots,n} p_j(r_j, x_i) \,\middle|\, i = 1,\dots,m \right\} \\ \text{subject to} \quad \sum_{j=1}^{n} r_j &= 1, \\ r_j &\geq 0 \quad j = 1,\dots,n \end{aligned} \tag{4.3.1}$$

$S$ is the schedule for PPA, and it represents a vector of resource allocation values.

For optimization of quality, the performance profile function maps directly to the resource profit function. This is possible because of the assumptions from Def. 4.1.6. Specifically, the algorithm can be interrupted at any instant of time and it returns a solution of the instance.

**Proposition 4.3.3.** *Let $T$ be the time allocated for run of the $\mathcal{P}$. Let the function $f$ represent the performance profile of a schedulable optimization algorithm. Then, the resource profit function that maximizes the obtained quality is*

$$p(r, x) = f(T * r, x).$$

*Moreover, this function is concave and twice continuously differentiable in $r$.*

*Proof.* The proposition clearly follows from the preemptability assumption. The concavity and continuity follows from the well-behaved improvement of $f$. □

For optimization of time, the profit is determined by the following proposition.

**Proposition 4.3.4.** *For schedulable decision algorithms, the time to find the first solution is optimized if the resource profit function $p$ for an algorithm is*

$$p(r, x) = -\frac{\eta_x}{r},$$

*where $\eta_x$ is the time to solve the instance $x$. This function is concave in $r$ and it is twice continuously differentiable.*

*Proof.* Let an algorithm that receives a fraction $r$ from the time share and needs $\eta_x$ to solve the instance. Then, from the preemptability criteria the time it needs to solve the instance will be $\frac{\eta_x}{r}$. The resource allocation profit is only this value negated. Therefore, its maximization leads to minimization of the time. Also, it is equivalent for limit and bound optimizations. The function is clearly concave, because its second derivative in $r$ is negative. □

For static schedules, we consider only *mean optimization* and *limit optimization*. The type of optimization determines the choice of the operator $\mathcal{B}$ and therefore the form of (4.3.1). For the mean optimization, it is

$$P(S) = \sum_{i=1}^{m} \max_{j=1,\dots,n} p_j(r_j, x_i).$$

For limit optimization it is

$$P(S) = \min_{i=1,\dots,m} \max_{j=1,\dots,n} p_j(r_j, x_i).$$

In the following sections, we explore how to solve these problems, which are in general non-convex and defined on non-convex sets.

## 4.4 Dynamic Schedules

Dynamic schedules allow the resource allocation function to change in discrete, equally spaced intervals. Only one algorithm may be scheduled for each interval. As we show later, the model is a more complex, but the proposed algorithms do not require as strict conditions as the algorithms for static schedules.

Non-constant allocation of resources to algorithms during the calculation may be helpful for decision problems, and thus time optimization. Notice it would not be helpful for optimization problems, as mentioned before. Therefore, we consider only decision algorithms for dynamic schedules.

The resource profit function is similar to that in Proposition 4.3.4, only extended to non-constants. This requires different formalization to express the time to solve the problems.

**Definition 4.4.1.** Time is divided into $b-1$ equally spaced intervals, and one infinitely long one. The interval do not overlap. $\tau$ corresponds to the start of the last interval. The length of each finite interval is $z$ and the total number of intervals is $b$.

By the preceding definition, the number of intervals fulfills

$$b = \frac{\tau}{z-1}.$$

The situation is depicted in Figure 4.1.

**Definition 4.4.2.** A schedule is *dynamic* when all resource allocation functions are constants from {0,1} on each interval, including the last, infinitely long one.

**Definition 4.4.3.** Resource allocation functions in dynamic schedules may be reformulated as a list of constants $r(1), \ldots, r(b)$. They correspond to the constant values of the RAF. The available resource constraint is

$$\sum_{j=1}^{n} r_j(k) \leq 1.$$

**Definition 4.4.4.** Let $\Theta(l)$ represent the time interval $\langle (l-1)z, lz)$ for $l = 1, \ldots, b$ if $l < b$. And $\Theta(l) = \langle (l-1)z, \infty)$ if $l = b$.

The profit of an algorithm on an instance is $-\eta$, with regard to the resource allocation function. Profit of an PPA on an instance is also $-\eta_S$, with regard to the schedule for the portfolio. We do not provide the formal definition because it uselessly clutters the text.

We consider dynamic schedules only for *mean optimization* and *bound optimization*. We show in Chapter 7 that both of these problems can be formulated as Markov Decision Processes, used in many areas of control theory and AI (Russell & Norvig, 2002; Sutton & Barto, 1998b). This allows use of many standardized algorithms.

## 4.5 Summary

The model we proposed in this chapter captures the basic ideas of learning PPAs. It somewhat simplifies the actual problem. The most prominent simplification is the lack of a classifier determining - at least approximately - the most suitable algorithm for each instance. Also, we ignored the existence of stochastic algorithms, and therefore also their high aptitude for combining (Gomes & Selman, 2001). These issues should be addressed in the future. We provide a discussion regarding them in Chapter 10.

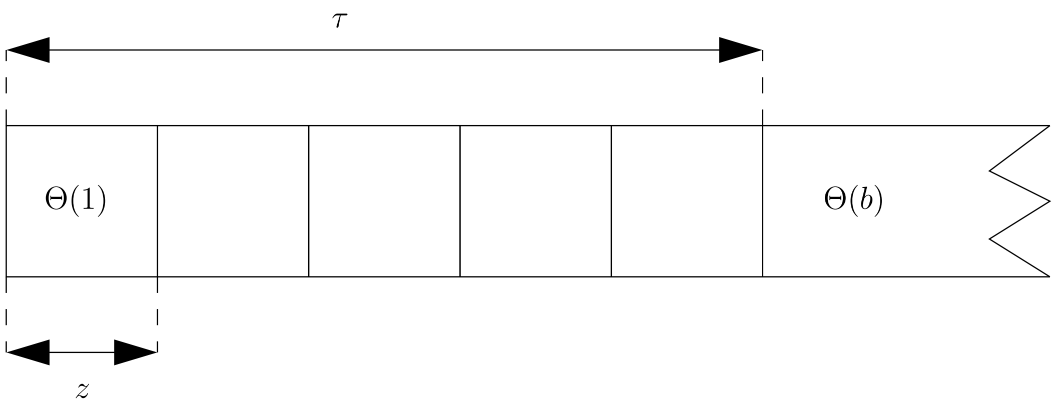

***Figure 4.1:*** *Sketch of how time is split for the dynamic schedules.*

# *Chapter 5*

# STATIC SCHEDULES

In this chapter we propose a simple but a quite powerful approach for optimizing performance of PPA limited to static schedules.

## *5.1 Basic Properties*

The combination of algorithms only makes sense in case it is at least as fast as the fastest available algorithm. We formally show this in the following proposition.

**Proposition 5.1.1.** *The profit from using a static schedule PPA is at least as great (on training instances) as when using the best algorithm, both for mean and limit Optimization.*

*Proof.* To prove the theorem for mean optimization, we need to show that

$$\max_{S:S\in C}\sum_{i=1}^{m}P(S,x_i)\geq\max_{j=1,\ldots,n}\sum_{i=1}^{m}p_j(x_i),$$

where $C$ is the simplex constrained by the resource constraints. Actually, the profit of choosing the best algorithm is equivalent to

$$\max_{j=1,\ldots,n}\sum_{i=1}^{m}p_j(x_i)=\max_{S:S\in C,S\in\mathbb{I}^*}\sum_{i=1}^{m}P(S,x_i),$$

where $\mathbb{I}^*$ is set of integer vectors. Then, by Theorem B.1.3, the theorem holds for mean optimization, because

$$\{S\,|\,S\in C,S\in\mathbb{I}^*\}\subseteq\{S\,|\,S\in C\}.$$

The proof for bound optimization follows the same idea. □

*Remark* 5.1.2. This property cannot be guaranteed generally for the whole problem distribution. In particular cases, the generalization bounds may show that the parallel portfolio is with a high probability more profitable than each atomic algorithm.

## *5.2 Formulation*

The optimization problem (4.3.1) is unsolvable by standard gradient methods, because the inner max operator makes the problem discontinuous. Therefore, in this section we propose a *block gradient method* to solve this problem. Because the problem is not convex, we cannot in general expect an optimal solution. We present an algorithm that can find optimal schedules in the next chapter.

First, we define a matrix that determines which algorithm from PPA solves each instance. Additionally, we provide additional functions that are determined by this classification.

**Definition 5.2.1.** *Classification* is the assignment of instances to algorithms in a portfolio. It determines which algorithm solves the particular instance; even if there is another one with a bigger profit on the instance. The assignment is realized by an $m \times n$ matrix $W$ with elements from $\{0, 1\}$. The matrix must fulfill

$$\sum_{j=1}^{n} W_{ij} = 1 \quad i = 1, \dots, m. \tag{5.2.1}$$

The semantics of $W_{ij} = 1$ is that the algorithm $j$ solves the instance $i$. The classification is used independently from the actual schedule.

**Definition 5.2.2.** The classification matrix separates the instance set $I$ into subsets $I_j$ $j = 1, \dots, n$ as

$$x_i \in I_j \Leftrightarrow W_{ij} = 1.$$

Also

$$m_j = |I_j| \,.$$

*Classification function* $k(\cdot, \cdot)$ induced by the classification matrix $W$ is defined as

$$k(l, j) = \{x_l \in I_j \,|\, x_l = x_i\} \; W_{ij} = 1 \quad j = 1, \dots, n \quad l = 1, \dots, m_j,$$

and for $W$ that fulfills (5.2.1) has cardinality 1.

Clearly, by (5.2.1), each instance is in exactly one set $I_j$. The optimization problem can now be expressed both in terms of $W$ and $S$. Profit of a PPA is not equal to the profit of the best algorithm on the instance, but instead to the algorithm to which it is assigned. The objective function may be reformulated as

$$\begin{aligned} &\max_S \mathcal{B}\left\{ \max_{1 \le j \le n} p_j(r_j, x_i) \,\middle|\, i = 1, \dots, m \right\} = \\ &= \max_S \mathcal{B}\left\{ \max_{w \ge 0} \left\{ \sum_{j=1}^{n} w_j p_j(r_j, x_i) \,\middle|\, \sum_{j=1}^{n} w_j = 1,\; w_j \in \{0, 1\} \right\} \,\middle|\, i = 1, \dots, m \right\}. \end{aligned}$$

Further, by Thm. B.1.2 the optimization problem reduces to

$$\begin{aligned} \text{maximize} \quad P(S, W) &= \mathcal{B}\left\{ \sum_{j=1}^{n} W_{ij} p_j(r_j, x_i) \,\middle|\, i = 1, \dots, m \right\} \\ \text{subject to} \quad \sum_{j=1}^{n} r_j &= 1, \\ \sum_{j=1}^{n} W_{ij} &= 1 \quad i = 1, \dots, m, \\ r_j &\ge 0 \quad j = 1, \dots, n, \\ W_{ij} &\in \{0, 1\} \quad i = 1, \dots, m \quad j = 1, \dots, n \end{aligned} \tag{5.2.2}$$

where $W$ is the classification matrix. The solutions of this reformulation do not necessarily correspond to real executions. Generally, they offer a lower bound of the real profit. The analysis is summarized by the following proposition.

**Proposition 5.2.3.** *The optimal solution of (5.2.2) is equivalent to the optimal solution of (4.3.1).*

```
Randomly initialize $W^0$
$i \leftarrow 0$
while $P(S^i, W^i) > P(S^{i-1}, W^{i-1})$ do
    $S^{i+1} \leftarrow \arg\max_S P(S^i, W^i)$
    $W^{i+1} \leftarrow \arg\max_W P(S^{i+1}, W^i)$
    $i \leftarrow i + 1$
end while
```

***Figure 5.1:*** *General CMA.*

**Definition 5.2.4.** Operator $\mathcal{B}$ is *monotonous* if it fulfills

$$(a_i \geq b_i \quad i = 1, \dots, k) \Rightarrow \mathcal{B}\{A\} \geq \mathcal{B}\{B\},$$

for any equally sized vectors $A = (a_1, \dots, a_k)$ and $B = (b_1, \dots, b_k)$.

The problem can further be simplified by dropping the integrality assumption, as the following proposition states.

**Proposition 5.2.5.** *Let the operator $\mathcal{B}$ be monotonous. Optimal solution of (5.2.2) without integrality constraints does not have a better solution as the original problem.*

*Proof.* The proof is simple. Let, without loss of generality, $W_{kl} \in (0, 1)$. From the assumption $\sum_{j=1}^{n} W_{ij} = 1$ there must be another indices $l_1, \dots, l_h$ where $W_{kl_i}$ is not integer. Let $l^*$ be the index with highest value of $p_{l^*}(r_{l^*}, x_k)$, breaking ties arbitrarily. By monotonicity of $\mathcal{B}$, setting $W_{kl^*}$ to 1 and other fractional values to 0 does not decrease the value of solution. □

As a result, the optimization problem may be solved without the integrality constraint. Then, it can be modified by the procedure from the proof above to satisfy the integrality constraint. Therefore, standard gradient methods could be applied to solve it. However, the main problem is that the problem is not convex and therefore the search for the global maximum would be quite problematic.

## 5.3 Classification-Maximization Algorithm

The reformulation above gives a way to a static schedule optimization algorithm that works in two phases. In the first phase it estimates the optimal *classification* of the instances. Given this classification, it *maximizes* the assignment of resources to algorithms. During the maximization phase, the algorithm that is assumed to solve each instance is fixed by the result of the classification phase. These phases are iterated while the PPA profit decreases. Thus the phases are *classification* and *optimization*.

This method is generally known as *block coordinate descent method* or *nonlinear Gauss-Seidel method* (Bertsekas, 2003). In general, it does not guarantee convergence to the global optimum, even for convex functions. We discuss globally optimal methods in Chapter 6. The general form of CMA is depicted in Fig. 5.1.

In the classification phase, the algorithm determines the optimal assignment matrix $W$ for a constant $S$. It means to solve

$$\begin{aligned} \text{maximize} \quad P(W) &= \mathcal{B}\left\{ \sum_{j=1}^{n} W_{ij} p_j(r_j, x_i) \,\middle|\, i = 1, \dots, m \right\} \\ \text{subject to} \quad \sum_{j=1}^{n} W_{ij} &= 1 \quad i = 1, \dots, m, \\ r_j &\geq 0 \quad j = 1, \dots, n. \end{aligned} \tag{5.3.1}$$

As the following proposition states, the optimal solution of this problem can be found by simply iterating the set of available instances and algorithms.

**Proposition 5.3.1.** *Let $\mathcal{B}$ be monotonous. Then, a solution of (5.3.1) with optimal profit is achieved if*

$$W_{ij} = 1 \Leftrightarrow j = \arg \max_{j=1,\ldots,m} p_j(r_j, x_i) \,, \tag{5.3.2}$$

*and ties may be broken arbitrarily. In other words, the optimal classification is the one that actually corresponds to the schedule.*

*Proof.* The reasoning is the same for both directions. Assume that the optimal solution does not satisfy (5.3.2). So let $W_{i\hat{j}} = 1$, be a classification with a sub-optimal profit. Let $j^* = \arg\max_{j=1,\ldots,m} p_j(r_j, x_i)$. Clearly, setting $W_{i\hat{j}} = 0$ and $W_{ij^*} = 1$ does not invalidate the classification constraints. Since this holds for all $i$, monotonicity of $\mathcal{B}$ implies that the profit is not decreased. Therefore, the solution that fulfills the condition must also have the optimal profit. □

During the maximization step, the algorithm determines the optimal $S$ given a constant $W$. It means to solve

$$\begin{aligned}
\text{maximize} \quad P(S) &= \mathcal{B}\left\{ \sum_{j=1}^{n} W_{ij} p_j(r_j, x_i) \;\middle|\; i = 1, \ldots, m \right\} \\
\text{subject to} \quad \sum_{j=1}^{n} r_j &= 1, \\
r_j &\geq 0 \quad j = 1, \ldots, n,
\end{aligned} \tag{5.3.3}$$

The maximization step depends on choice of $\mathcal{B}$. Therefore, we analyze the specific cases in the next section. In addition, because CMA is a randomized algorithm (see initialization of $W$), repeated execution may produce better results.

## 5.4 CMA-SPPA

In this section, we define *Classification Maximization Algorithm for Static Parallel Portfolios of Algorithms*. The classification stage is the same as in the general CMA, but the maximization phase is different for mean and limit optimization.

The maximization step maximizes the profit function given a firm assignment determined by $W$. This makes the problem much simpler, because we then deal with a continuous, convex problem. As we show, this problem can be solved analytically, using only constraint values. To perform the subsequent analysis, we need to place additional restriction on the class of profit profiles of the algorithms. Hence the following definition:

**Definition 5.4.1.** A set $E$ of resource profit functions $p$ is *homogeneous* if they all can be expressed as

$$p_j(r, x) = \nu_j(r) * \mu_j(x).$$

**Definition 5.4.2.** Operator $\mathcal{B}$ is *multiplicatively transitive* if

$$\mathcal{B}\left\{c * a_i \,|\, i = 1, \ldots, k\right\} = c * \mathcal{B}\left\{a_i \,|\, i = 1, \ldots, k\right\}.$$

**Definition 5.4.3.** Operator $\mathcal{B}$ is *recursive* if

$$\mathcal{B}(A \cup B) = \mathcal{B}\{\mathcal{B}(A), \mathcal{B}(B)\},$$

for any equally sized vectors $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$.

The next lemma trivially establishes the above properties for operators that are later used for mean optimization.

**Lemma 5.4.4.** *Operators* min *and* mean value$(\frac{1}{m}\sum_{i=1}^{m} x_i)$ *are recursive, multiplicatively transitive, and monotonous.*

*Proof.* The proof follows straight from the definitions. □

In the maximization phase of CMA, $W$ is given; and optimal $S$ must be determined. Thus, the objective function of (5.3.3) may be reformulated to

$$\begin{aligned}&\max_{S,W} \mathcal{B}\left\{ \sum_{j=1}^{n} w_j p_j(r_j, x_i) \,\middle|\, i = 1, \ldots, m \right\}\\ &= \max_{S,W} \mathcal{B}\left\{ p_j(r_j, x_i) \,|\, k(l,j) = x_i, \;\; i = 1, \ldots, m \right\}.\end{aligned}$$

Further by recursiveness of $\mathcal{B}$,

$$\max_{S} \mathcal{B}\left\{ \mathcal{B}\left\{ p_j(r_j, x_{k(l,j)}) \,|\, l = 1, \ldots, m_j \right\} \,|\, j = 1, \ldots, n \right\}.$$

For homogeneous profit functions we get $p_j(r_j, x_i) = \nu_j(r_j) * \mu_j(x_i)$. Assuming $\mathcal{B}$ to be multiplicatively transitive, the objective is

$$\begin{aligned}&\max_{S} \mathcal{B}\left\{ \mathcal{B}\left\{ \nu_j(r_j) * \mu_j(x_{k(l,j)}) \,|\, l = 1, \ldots, m_j \right\} \,|\, j = 1, \ldots, n \right\}\\ &= \max_{S} \mathcal{B}\left\{ \nu_j(r_j) * \mathcal{B}\left\{ \mu_j(x_{k(l,j)}) \,|\, l = 1, \ldots, m_j \right\} \,|\, j = 1, \ldots, n \right\}.\end{aligned}$$

The general form of the optimization problem is now

$$\begin{aligned} \text{maximize} \quad & P(S) &=& \; \mathcal{B}\left\{ \nu_j(r_j) \mathcal{B}\left\{ \mu_j(x_{k(l,j)}) \,|\, l = 1, \ldots, m_j \right\} \,|\, j = 1, \ldots, n \right\} \\ \text{subject to} \quad & \sum_{j=1}^{n} r_j &=& \; 1 \\ & r_j &\geq& \; 0 \quad j = 1, \ldots, n \end{aligned} \tag{5.4.1}$$

The above can be summarized in the following proposition.

**Proposition 5.4.5.** *Let the algorithms from $E$ be schedulable and homogeneous. Let $\mathcal{B}$ be recursive and multiplicatively transitive. Then the optimal solution of (5.4.1) results in the optimal schedule for a fixed classification.*

In the following, we address the choice of a specific $\mathcal{B}$ operator. For mean optimization it is mean value, and for limit optimization it is min. Both options lead to analytically expressible optimal solution conditions.

## 5.4.1 Mean Optimization

First, define the cumulative profit of an algorithm on the instances that are assigned to it as

$$d_j = \sum_{i=1}^{m_j} \mu_j(x_{k(i,j)}).$$

Notice, it is not a mean value but a sum. The specific problem for mean optimization then becomes

$$\begin{aligned} \text{maximize} \quad P(S) &= \frac{1}{m}\sum_{j=1}^{n} \nu_j(r_j)d_j \\ \text{subject to} \quad \sum_{j=1}^{n} r_j &= 1 \\ r_j &\geq 0 \quad j = 1,\dots,n \end{aligned} \tag{5.4.2}$$

**Theorem 5.4.6.** *Let the algorithms be schedulable. Then, the solution of (5.4.2) is globally optimal if it fulfills*

$$\frac{\nu_j'(r_j)}{\nu_k'(r_k)} = \frac{d_k}{d_j}.$$

*Proof.* The theorem follows from the second order optimality conditions. The Lagrangian function, Definition B.1.5, for an equivalent minimization problem is

$$L(S,\lambda) = -\frac{1}{m}\sum_{j=1}^{n} \nu_j(r_j)d_j + \lambda\sum_{j=1}^{n} r_j.$$

The equation in the theorem follows by algebraic manipulations from the necessary optimality criteria, Theorem B.1.4. Because the of schedulability assumption, it also fulfills the convexity criterium of Theorem B.1.6. Thus this is a local maximum. By Proposition B.1.7, this local maximum is also global. □

### *5.4.2 Limit Optimization*

First define the minimal profit for each algorithm as

$$e_j = \min_{i=1,\dots,m_j} \mu_j(x_i),$$

then the optimization problem for limit optimization may be formulated as

$$\begin{aligned} \text{maximize} \quad P(S) &= \min_{j=1,\dots,n} \nu_j(r_j) * e_j \\ \text{subject to} \quad \sum_{j=1}^{n} r_j &= 1 \\ r_j &\geq 0 \quad j = 1,\dots,n \end{aligned} \tag{5.4.3}$$

The problem is equivalent to

$$\begin{aligned} \text{maximize} \quad P(S,R) &= R \\ \text{subject to} \quad R &\leq \nu_j(r_j) * e_j \quad j = 1,\dots,n \\ \sum_{j=1}^{n} r_j &= 1 \\ r_j &\geq 0 \quad j = 1,\dots,n \end{aligned} \tag{5.4.4}$$

Assuming that each algorithm solves at least one problem instance.

**Lemma 5.4.7.** *Let the algorithms be schedulable. Let the resource profit functions be monotonically increasing*[1]. *Let each algorithm be assigned at least one instance. The local extrema of (5.4.4), where the inequality*

$$R \leq \nu_j(r_j) * e_j \quad j = 1, \dots, n$$

*is replaced by the equality*

$$R = \nu_j(r_j) * e_j \quad j = 1, \dots, n$$

*are equivalent to local extrema of the original problem.*

*Proof.* By contradiction. The solution vector is $M = (r_1, \dots, r_n, R)$. Let the optimal solution be $M^* = (S^*, R^*) = (r_1^*, \dots, r_n^*, R^*)$ and assume there exists such $l$ that

$$\nu_l(r_l^*) > R^*,$$

and for all $j \neq k$

$$\nu_j(r_j^*) \geq R^*.$$

Because $\nu_l$ is continuous, there is an $\epsilon$ such that $\nu_l(r_l^* - \epsilon) > R^*$. Now, let

$$\widehat{M} = (\widehat{r_1}, \dots, \widehat{r_n}, \widehat{R}) = (r_1^* + \frac{\epsilon}{n-1}, \dots, r_l^* - \epsilon, \dots, \min_{j=1,\dots,n} \nu_j(\widehat{r_j})).$$

Because $\nu_j$ are increasing, $\nu_j(\widehat{r_j}) > \nu_j(r_j^*)$ for $j \neq l$. Then also

$$\min_{j=1,\dots,n} \nu_j(\widehat{r_j}) > \min_{j=1,\dots,n} \nu_j(r_j^*),$$

thus $\widehat{R} > R^*$. Therefore

$$\nabla P(M^*)'(\widehat{M} - M^*) = \widehat{R} - R^* > 0.$$

This violates the necessary local optimality condition, Proposition B.1.1. Thus the to contradiction proves the theorem. Notice that Proposition B.1.1 is for minimization problems. Negating the objective function transforms it to a maximization problem; that is why the sign in this proof is opposite to one in the proposition. □

Therefore, the optimal schedule for a fixed classification can be determined analytically.

**Theorem 5.4.8.** *Let the algorithms be schedulable. Let the resource profit functions be monotonically increasing. Let each algorithm be assigned at least one instance. Then the globally optimal solution of (5.4.4) satisfies*

$$\frac{\nu_j(r_j)}{\nu_k(r_k)} = \frac{e_j}{e_k}.$$

*Moreover, there is only one such solution.*

*Proof.* The fact that the optimal solution must fulfill this equalities follows from Lemma 5.4.7. We only need to show that there is only one such solution. Assume by contradiction, that there are two different solutions $r_1, r_2, \dots, r_m$ and $r_1', r_2', \dots, r_m'$. Without loss of generality, assume that $r_1 > r_1'$. Because

$$\sum_{j=1}^{m} r_j = \sum_{j=1}^{m} r_j',$$

[1] The requirement for schedulable optimization algorithms is to be non-decreasing.

there must be $r_l < r'_l$. Let it be, without loss of generality $r_2$. Because both are the solutions of the same problem,

$$\begin{aligned} \nu_1(r_1) &= \nu_2(r_2) \\ \nu_1(r'_1) &= \nu_2(r'_2). \end{aligned}$$

From monotonicity of $r_j$ we get

$$\nu_1(r_1) > \nu_1(r'_1) = \nu_2(r'_2) > \nu_2(r_2),$$

what is a contradiction. Thus there is at most one solution, and therefore it is globally optimal. □

## 5.5 CMA: Decision Problems

The resource profit function of decision algorithms is the same for all algorithms, as Proposition 4.3.4 states. Therefore, in this case, we can derive more specific and useful condition. The execution time of a decision algorithm on instance $x$ is $\eta(x)$. Thus the resource profit functions are homogeneous and equal to

$$\begin{aligned} \nu(r) &= \frac{1}{r} \\ \mu(x) &= -\eta(x). \end{aligned}$$

First, we provide the specific theorems and then we note how these properties can be easily satisfied.

**Theorem 5.5.1.** *Let the PPA use decision algorithms. The mean optimal schedule for the maximization phase of CMA, given a fixed classification, must fulfill*

$$\frac{r_o}{r_p} = \sqrt{\frac{\sum_{i=1}^{m_o} \eta_o(x_{k(i,o)})}{\sum_{i=1}^{m_p} \eta_p(x_{k(i,p)})}}.$$

*Proof.* By Theorem 5.4.6. □

The following theorem predicts the mean run-time of a PPA with mean optimal static schedule for decision algorithms. Notice it is for a fixed classification of instances, and thus it is a lower bound for the execution.

**Theorem 5.5.2.** *Let $S$ be the optimal schedule of a PPA for decision problems, obtained in the maximization phase of CMA for mean optimization. Then the expected execution time of the schedule is*

$$\max_S P(S) = \left( \sum_{j=1}^{n} \sqrt{\frac{1}{m} \sum_{i=1}^{m_j} \eta_p(x_{k(i,j)})} \right)^2 .$$

*Proof.* By algebraic manipulation from the previous theorem. □

**Theorem 5.5.3.** *Let the PPA use decision algorithms. The limit optimal schedule for the maximization phase of CMA, given a fixed classification, must fulfill*

$$\frac{r_o}{r_p} = \frac{\min_{i=1,\dots,m_o} -\eta_o(x_{k(i,o)})}{\min_{i=1,\dots,m_p} -\eta_p(x_{k(i,p)})}$$

*Proof.* By algebraic manipulations from Theorem 5.4.8. □

The optimality equalities required by mean and limit optimizations for decision algorithms can be satisfied as follows. The resources are assigned to the algorithms ignoring the resource constraints to fulfill the condition from the theorem. Because normalization of the sum of algorithms' resource shares to 1 does not change the relative ratio of shares, the optimality constraint required by the theorem is satisfied after the normalization.

## 5.6 Summary

We presented in this chapter a simple method to calculate simple schedules. As we show later, despite its simplicity this approach is quite effective. The calculation procedure is done in iterative classification and maximization steps. This idea is similar to Expectation Maximization algorithm (Russell & Norvig, 2002; Bertsekas, 2003) and Block Coordinate Descent.

Because the optimization problems in this chapter are not convex, we cannot expect to arrive to an optimal solution. However, the randomization at the start of CMA increases the chance of getting to the global maximum in several executions. In the next chapter, we provide a modified algorithm that finds the optimal schedule, but has a higher computational complexity.

A simple extension, helpful in some cases, is to pre-process the portfolio by removing the algorithms that are worse for each instance than other single algorithm. It may even be possible to remove algorithm that is dominated by a different algorithm for each instance, but this would probably require a deeper analysis.

The applicability of the algorithms for static schedules, as presented in this chapter, is limited to PPA of homogeneous algorithms. While it is trivial condition for decision algorithms, it is not clear whether optimization algorithms tend to fulfill it.

Other research on algorithm portfolios is concerned with calculating a prediction function, which predicts the quality of the algorithms. If such a function was available, then the schedule could be calculated for each instance individually. The overhead of schedule calculation would be small, because CMA procedure is fast, even for large portfolios.

# *Chapter 6*

# STATIC SCHEDULES: GEOMETRIC FORMULATION

In this chapter, we propose an algorithm for calculating static schedules that is optimal but still tractable in terms of the sample size, and a method to visualize a static schedule of algorithms. These results are also used in generalization bounds in Chapter 8.

## *6.1 Optimal Algorithm*

A naive search for the globally optimal schedule may be obtained by a modifying CMA to search through all classifications. This algorithm is depicted in Fig. 6.1. The number of maximization

```
$i = 0$
$P^* = 0$
for all $W \in \{0,1\}^{m,n}$ do
  $P \leftarrow \max_S P(S, W^i)$
  if $P > P^*$ then
    $P^* \leftarrow P$
  end if
  $i \leftarrow i + 1$
end for
```

***Figure 6.1:*** *Naive Optimal CMA - NOCMA.*

runs of this algorithm is

$$\mathrm{O}\left(n^m\right),$$

where $m$ is the number of samples and $n$ is the number of algorithms. Therefore, the algorithm needs

$$\mathrm{O}\left(mn * n^m\right)$$

operations to find the optimum. This is exponential in the number of training samples, what is a generally high number. To get a more effective algorithm, we show that only a subset of all classifications must be explored. The following analysis leads to an algorithm with time complexity polynomial in $m$ and exponential in $n$.

**Definition 6.1.1.** A classification is *valid* when it is consistent with a schedule. On other words, each instance is assigned to the algorithm that has the best performance on it.

Only valid classification may happen in reality, and their profit is higher than a profit of a suboptimal classification. As a result, only valid classifications need to be explored to find the best schedule. This notion is stated formally in the next lemma.

**Lemma 6.1.2.** *Let $S^*$ be the globally optimal schedule of a PPA and let $\widehat{S}$ be the best schedule obtained from a valid classification and a single maximization phase. Then*

$$P(S^*) = P(\widehat{S}).$$

*Proof.* We show the lemma by contradiction. Let

$$P(S^*) > P(\widehat{S}).$$

Clearly, a classification $W^*$ from a schedule $S^*$ is valid. Then, by running a maximization phase of CMA on this classification, we get a schedule $S^{*\prime}$. Then, by

$$P(S^{*\prime}) \geq P(S^*) > P(\widehat{S}),$$

what is with contradiction of optimality condition of $\widehat{S}$. $\square$

The following lemma defines the properties of a valid classification.

**Lemma 6.1.3.** *Let the resource profit function be homogeneous. Let a, b be arbitrary algorithm from PPA. Then, for a valid classification, we have*

$$x_o \in I_a \wedge x_p \in I_b \Rightarrow \frac{\mu_a(x_o)}{\mu_b(x_o)} \geq \frac{\mu_a(x_p)}{\mu_b(x_p)},$$

where $I_a$ is the set of instances assigned to $a$ and $I_b$ to $b$.

*Proof.* Because the classification is valid,

$$\begin{aligned} p_a(x_o) &\geq p_b(x_o) \\ p_a(x_p) &\geq p_b(x_p). \end{aligned}$$

From the homogeneity assumption, we get that

$$\begin{aligned} \mu_a(x_o)\nu_a(r_a) &\geq \mu_b(x_o)\nu_b(r_b) \\ \mu_a(x_p)\nu_a(r_a) &\leq \mu_b(x_p)\nu_b(r_b). \end{aligned}$$

By arithmetic manipulations

$$\begin{aligned} \frac{\nu_b(r_b)}{\nu_a(r_a)} &\leq \frac{\mu_a(x_o)}{\mu_b(x_o)} \\ \frac{\nu_b(r_b)}{\nu_a(r_a)} &\geq \frac{\mu_a(x_p)}{\mu_b(x_p)}, \end{aligned}$$

the lemma follows. $\square$

As a consequence, we have the following lemma.

**Lemma 6.1.4.** *Let the resource profit functions be homogeneous. For any two instances $x_o, x_p$, and algorithms a and b, let without loss of the generality*

$$\frac{\mu_a(x_o)}{\mu_b(x_o)} > \frac{\mu_a(x_p)}{\mu_b(x_p)}.$$

*Then for a valid classification*

$$(x_o \in I_b \Rightarrow x_p \notin I_a) \wedge (x_p \in I_a \Rightarrow x_o \notin I_b),$$

*where $I_a$ is the set of instances assigned to a and $I_b$ to b.*

*Proof.* To show $x_o \in I_b \Rightarrow x_p \notin I_a$ by contradiction, assume

$$x_o \in I_b \wedge x_p \in I_a.$$

Then, application of Lemma 6.1.3 leads to

$$\frac{\mu_a(x_o)}{\mu_b(x_o)} \leq \frac{\mu_a(x_p)}{\mu_b(x_p)},$$

what is clearly a contradiction. The other statement, $x_p \in I_a \Rightarrow x_o \notin I_b$, can be proved analogously. $\square$

For each pair of algorithms, the instances can be decreasingly ordered according to

$$\frac{\mu_a(x)}{\mu_b(x)}.$$

Then, from Lemma 6.1.4, for each schedule there is a *split point* $D$, for which *no* instances for which $\frac{\mu_a(x)}{\mu_b(x)} > D$ are assigned to $b$ and *no* instances for which $\frac{\mu_a(x)}{\mu_b(x)} < D$ are assigned to $a$.

**Theorem 6.1.5.** *The number of valid classifications of instances to problems, when RAF are homogeneous, is at most*

$$(m+1)^{\binom{n}{2}}.$$

*Proof.* As we showed in Lemma 6.1.4, only schedules that can be represented by a set of $\binom{n}{2}$ split points are valid. Moreover, there is at most one classification, consistent with a set of split points. To see this, take two different classifications. Because they are different, they must assign at least one instance to different algorithm, let it be algorithms $a$ and $b$. This makes the split point between $a$ and $b$ different for each classification. Because, there is at most

$$(m+1)^{\binom{n}{2}}$$

possible split-point sets, the theorem follows. $\square$

As a result, of the above theorem, we propose a Valid Optimal Classification Maximization Algorithm, depicted in Figure 6.2. It iterates through all split-point sets, for each creates a classification, and finds the optimal schedule for that classification. A way to make classification from a split-point set is to assign instance to an algorithm if it is assigned to it for each split-point. Notice, that the conditions on valid schedules are necessary, not sufficient. Therefore, the algorithm also checks classifications that are not valid. This increases its complexity but does not impact the optimality.

**Theorem 6.1.6.** *VOCMA reaches a schedule with the optimal profit in*

$$O\left(mn * (m+1)^{\binom{n}{2}}\right),$$

*for mean and limit classifications.*

*Proof.* The profit is optimal because each valid schedule has a unique split-point set by Lemma 6.1.4, and it is sufficient to enumerate all valid schedule by Lemma 6.1.2. The complexity is evident from the total number of split-point sets and Theorem 5.4.6 and Theorem 5.4.8. $\square$

```
$i = 0$
$P^* = 0$
Initialize split points $(s_1, \ldots, s_{\binom{n}{2}}) = (0, \ldots, 0)$
**for all** $(s_1, \ldots, s_{\binom{n}{2}}) \in (0 \ldots m)^{\binom{n}{2}}$ **do**
  Create $W$ from $s_1, \ldots, s_{\binom{n}{2}})$
  **if** $W$ is not valid **then**
    CONTINUE
  **end if**
  $P \leftarrow \max_S P(S, W^i)$
  **if** $P > P^*$ **then**
    $P^* \leftarrow P$
  **end if**
  $i \leftarrow i + 1$
**end for**
```

***Figure 6.2:*** *Valid Optimal CMA - VOCMA.*

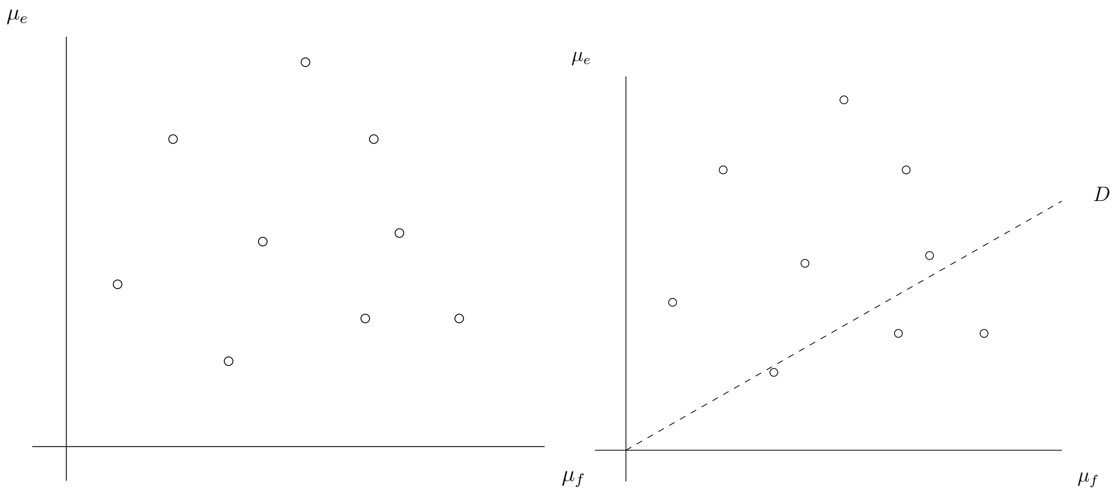

***Figure 6.3:*** *Point graph of performance of algorithms, and how the split point divides it.*

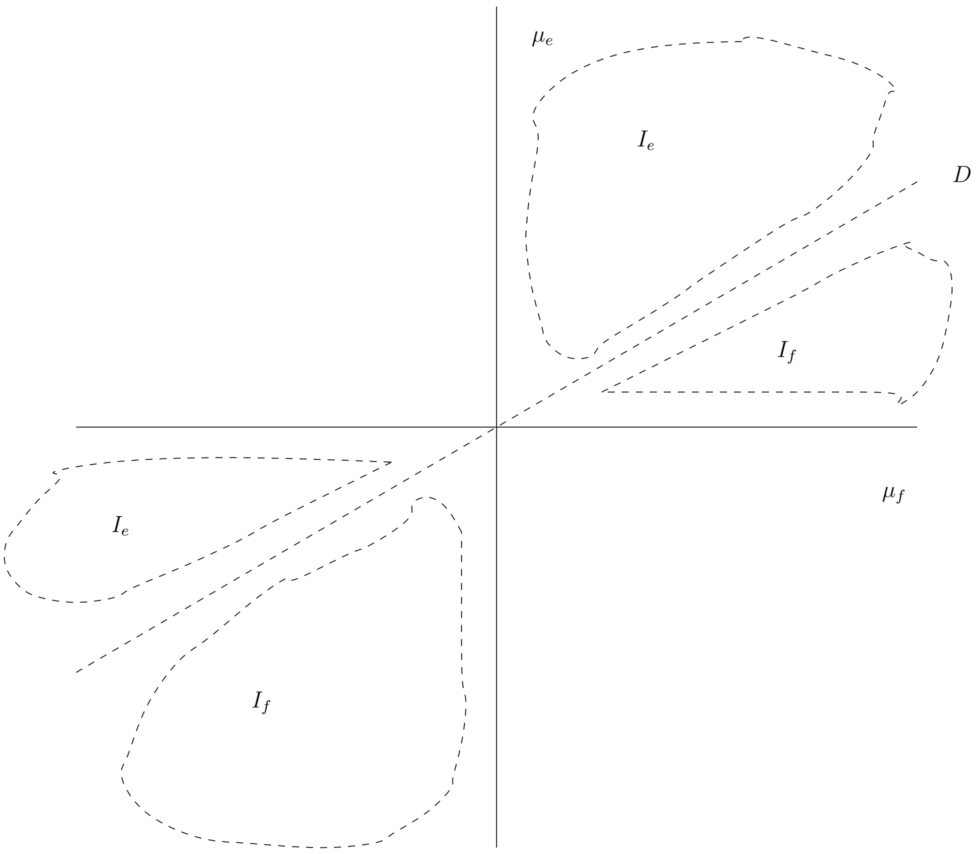

***Figure 6.4:*** *Picture of how split point divides the instance set.*

## 6.2 Visualization

Consider having only two algorithms $e$ and $f$. If they are homogeneous, their performance may depicted as a two-dimensional point graph. Each point represents an instance. Position of each point for instance $x$ is determined by $\mu_e(x)$ and $\mu_f(x)$ on axes. A simple example, with 5 instances is in Figure 6.3.

As shown in the section above, we can define a split point $D$. Because there are only two algorithms we have

$$\frac{\mu_e(x)}{\mu_f(x)} \geq D \Leftrightarrow x \in I_e \tag{6.2.1}$$

$$\frac{\mu_e(x)}{\mu_f(x)} \leq D \Leftrightarrow x \in I_f, \tag{6.2.2}$$

breaking ties arbitrarily to ensure $I_e \cap I_f = \emptyset$. Interestingly, the point graph defined above, a split point represents a line that separates $I_e$ and $I_f$. This is depicted in Figure 6.3 and Figure 6.4.

Obviously, for two algorithms, a split point defines a classification. However, it also defines a schedule. We can calculate what the allocation of resources must be in order for the classification to be valid. The conditions are:

$$\frac{\mu_e(x)\nu_e(r_e)}{\mu_f(x)\nu_f(r_f)} \geq 1 \quad \forall x \in I_e$$

$$\frac{\mu_e(x)\nu_e(r_e)}{\mu_f(x)\nu_f(r_f)} \leq 1 \quad \forall x \in I_f.$$

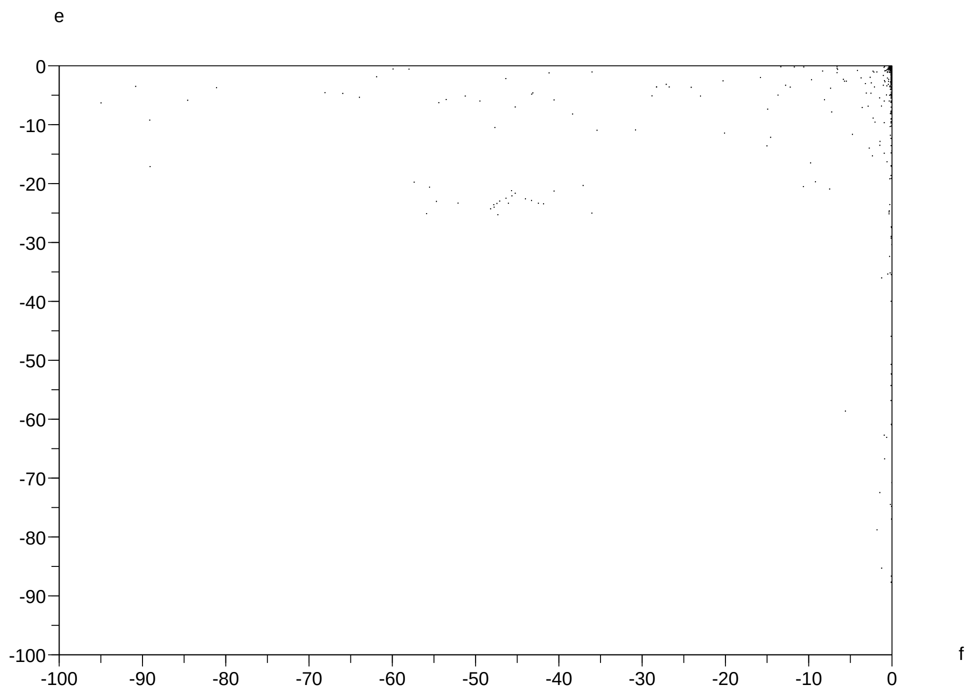

***Figure 6.5:*** *Visualization of algorithms* 21 *and* 20 *from the SAT problem application. $f$ denotes the $\mu$ of algorithm* 21*, and $e$ denotes $\mu$ of algorithm* 20*.*

From (6.2.1) and (6.2.2) we get the sufficient conditions for a valid schedule:

$$D\frac{\nu_e(r_e)}{\nu_f(r_f)} \geq 1 \quad D\frac{\nu_e(r_e)}{\nu_f(r_f)} \leq 1.$$

Thus the condition is

$$\frac{\nu_e(r_e)}{\nu_f(r_f)} = \frac{1}{D}.$$

The preceding can be formalized in the following theorem.

**Theorem 6.2.1.** *Let the PPA consist of two schedulable and homogeneous algorithms. Let the functions $\nu$ be monotonously increasing. A split-point $D$ the uniquely defines a schedule*[1] *as*

$$\frac{\nu_e(r_e)}{\nu_f(r_f)} = \frac{1}{D}. \tag{6.2.3}$$

*Proof.* The correspondence between a split point and a schedule is evident from the above analysis. Uniqueness of solution of (6.2.3) can be done similarly to the proof of Theorem 5.4.8. □

Specific visualization schemes from the application, from Chapter 9, are in Figure 6.5 and Figure 6.6. Notice the profits are negative, this is because it is for a decision algorithms. Figure 6.5 shows that the algorithms have quite different performance on many instances. On the other hand, Figure 6.6 shows algorithms with a similar performance.

[1] Also the resource allocation functions.

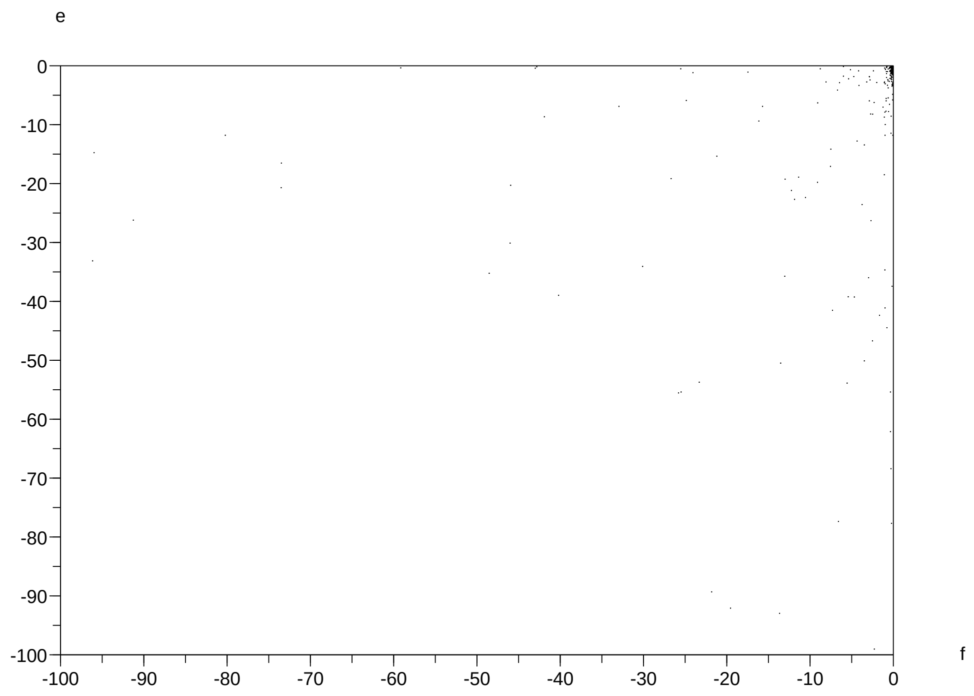

***Figure 6.6:*** *Visualization of algorithms 12 and 22 from the SAT problem application. $f$ denotes the $\mu$ of algorithm 12, and $e$ denotes $\mu$ of algorithm 22.*

## *6.3 Summary*

We presented in this chapter an algorithm to calculate optimal static schedules for PPA. It is based on CMA from the preceding chapter. Though the method guarantees optimality, it is impractical in other aspects because of its high computational complexity. Specifically, it has a polynomial time complexity in the number of instances, but exponential in the number of algorithms in PPA. Therefore, a calculation of schedules for more than 5 algorithms is typically infeasible. However, the naive implementation of the algorithm, as presented in this chapter, may be significantly improved by propagating the bounds and eliminating creation of schedules that are not valid. In the graph of split points between the algorithms, every edge is probably determined by all cycles that cross it. This could eliminate many of the created invalid schedules. This would require a further investigation, more details are in Chapter 10.

The VOCMA approach also leads to a nice visualization. The visualization allows to compare the performance of algorithms, and makes the notion of split points more intuitive.

# *Chapter 7*

# DYNAMIC SCHEDULES

In this chapter we analyze dynamic schedules, formally introduced in Chapter 4. As noted before, dynamic schedules make sense only for time optimization, that is portfolios of deterministic algorithms. However, for sake of consistency, we still use the notion of profit equal to negative value of execution time. Therefore, finding the schedules is a maximization problem.

## *7.1 Basic Properties*

Finding the optimal schedule for the dynamic model by heuristically enumerating all possible schedules would be a very time consuming task. A common method for solving hard problems dynamic programming. One formulation that permits the use of this technique is a Markov Decision Process (MDP). As mentioned above, it is possible to formulate the dynamic model as an MDP, in which each policy corresponds to a dynamic schedule. In this chapter, $X$ is a random variable representing an instance from $I$.

**Definition 7.1.1.** $Z_e(l, X)$ denotes the probabilistic event that the algorithm $e$ solves an instance from $I$ in the time interval $\Theta(l)$. Formally,

$$Z_e(l, X) = \mathrm{I}\left\{(l-1)z \le \eta_e(X) < lz\right\}.$$

Alternatively, $Z$ represents a set of instances, that are solved within that time. The actual meaning is determined by the context in which it is used.

**Definition 7.1.2.** $Z_S(l, X)$ denotes the event of an instance being solved by any algorithm from the portfolio in the time interval $\Theta(l)$. The algorithms are executed according to the schedule $S$.

We use $Z(l, X)$ when the choice of the schedule is clear.

**Definition 7.1.3.** $Z(k \dots l, X)$, $l \ge k$ represents the event of an instance being solved in any of the intervals $k$ through $l$. Formally,

$$Z(k \dots l, X) = \bigcup_{j=k,\dots,l} Z(j, X).$$

**Lemma 7.1.4.** *Let $e$ be a schedulable decision algorithm. Let*

$$\begin{aligned} \sum_{k=1}^{l} r_e(k) &= a, \\ r_e(l) &= 1. \end{aligned}$$

*Also, Let $e'$ be the same algorithm, but with resource allocation function*

$$r_{e'}(k) = 1 \quad k = 1, \dots, a.$$

*Then*

$$Z_e(l, X) \equiv Z_{e'}(a, X).$$

*Proof.* The proposition follows from preemptability of the algorithm $e$, because it can then be preempted and resumed with no performance penalty. □

In the further text, $Z_e$ refers to solving the instance in a hypothetical case if $e$ had $r_e(l) = 1$ for $l = 1, \ldots, b$. This is consistent with the original definition, because of the preceding lemma. In addition, $Z_k$ is a shorthand notation for $Z_{e_k}$.

The next lemma shows an important property of the resource allocation functions, which can be utilized to obtain effective algorithms for finding dynamic schedules.

**Lemma 7.1.5.** *Let the PPA be a set of schedulable decision algorithms. Let $r_e, r'_e$ be any resource allocation functions for which*

$$\sum_{l=1}^{k} r_e(l) = \sum_{l=1}^{k} r'_e(l) = h,$$

*and*

$$r_e(k+1) = r'_e(k+1).$$

*Then,*

$$p_e(r_e, x) = p_e(r'_e, x) \quad \forall x \in Z_e(h+1).$$

*Proof.* The lemma, as well as Lemma 7.1.4, follows from preemptability of the algorithms. □

For sake of simplicity we use,

$$p_e(a, x) = p_e(r_e, x) \text{ if } \sum_{i=1}^{k} r_e(l) = a$$

on time the interval $k+1$ whenever $x \in Z_e(a)$. We also use,

$$p_e(a \ldots, x) = p_e(r_e, x) \text{ if } \sum_{l=1}^{k} r_e(l) = a,\ r_e(l) = 1,\quad l = k+1, \ldots, b,$$

on the time intervals from $k+1$ to $b$ whenever $x \in Z_e(a \ldots)$. These simplifications are possible because of Lemma 7.1.5.

## 7.2 Formulation

We define two alternative MDP's for both mean and bound optimizations. They share states the same set of states and actions, but they differ in the definition of rewards.

*COMMON MDP DEFINITION* *States* are vectors of natural numbers. They are divided into two disjunct sets

$$\begin{aligned} S &= \left\{ (c_1, \ldots, c_n) \,\middle|\, c_j \in \{0, \ldots, b\},\ \sum_{j=1}^{n} c_j < b \right\} \\ F &= \left\{ (c_1, \ldots, c_n, k) \,\middle|\, c_j \in \{0, \ldots, b\},\ k \in \{1, \ldots, n\},\ \sum_{j=1}^{n} c_j \leq b \right\}. \end{aligned}$$

The initial state is $s_0 = (0, \ldots, 0) \in S$. *Actions* for each state from $S$ are from $\{A_1, \ldots, A_n\}$, where $A_j$ are arbitrary diverse elements. Given a state $s = (c_1, \ldots, c_n)$, and an action $A_k$, the possible subsequent state are:

$$\begin{aligned} s' &= (c_1, \ldots, c'_k = c_k + 1, \ldots, c_n) \in S \text{ if } \sum_{j=1}^{n} c_j < b - 1 \\ f' &= (c_1, \ldots, c'_k = c_k + 1, \ldots, c_n, k) \in F. \end{aligned}$$

The probability distribution over the subsequent states $s'$ and $f'$, given a random problem instance, an action $A$, and a state $s$, is:

$$\begin{aligned} \mathbf{P}\left[f' \,|\, s, A_k\right] &= \mathbf{P}\left[Z_k(c'_k, X) \,|\, \neg Z(s, X)\right] \\ \mathbf{P}\left[s' \,|\, s, A_k\right] &= \mathbf{P}\left[\neg Z_k(c'_k, X) \,|\, \neg Z(s, X)\right], \end{aligned}$$

where the notation $Z(s, X)$ denotes and event of any algorithm from the portfolio, solving the instance until this state. The *reward function* $R$ for

$$\begin{aligned} f &= (c_1, \ldots c_n, k) \in F \\ s &= (c_1, \ldots c_n) \in S \end{aligned}$$

is defined as

$$\begin{aligned} R(f) &= \mathbf{E}\left[\Pi_k(f, X) \,|\, Z_k(c_k, X) \wedge \neg Z(f \ominus k, X)\right] \text{ if } \sum_{j=1}^{n} c_j < b \\ R(f) &= \mathbf{E}\left[\Pi_k(f \ldots, X) \,|\, Z_k(c_k \ldots, X) \wedge \neg Z(f \ominus k, X)\right] \text{ if } \sum_{j=1}^{n} c_j = b \\ R(s) &= 0, \end{aligned}$$

where $(c_1, \ldots, c_k, \ldots, c_n) \ominus k = (c_1, \ldots, c_k - 1, \ldots, c_n)$. $\Pi$ denotes an abstract reward, later specified for mean and bound optimization. Next, we refer to $\Pi$ as to an abstract profit.

*REWARDS FOR MEAN OPTIMIZATION* The *reward function* $R$ is defined as

$$\begin{aligned} R(f) &= -\sum_{j=1, j \neq k}^{n} c_j + \mathbf{E}\left[p_k(c_k, X) \,|\, Z_k(c_k, X) \wedge \neg Z(f \ominus k, X)\right] \text{ if } \sum_{j=1}^{n} c_j < b \\ R(f) &= -\sum_{j=1, j \neq k}^{n} c_j + \mathbf{E}\left[p_k(c_k \ldots, X) \,|\, Z_k(c_k \ldots, X) \wedge \neg Z(f \ominus k, X)\right] \text{ if } \sum_{j=1}^{n} c_j = b \\ R(s) &= 0. \end{aligned}$$

In this case, $\Pi$ is

$$\Pi_k(f, X) = -\sum_{j=1, j \neq k}^{n} c_j + p_k(c_k, X).$$

*REWARDS FOR BOUND OPTIMIZATION* The *reward function* $R$ is defined as

$$\begin{aligned}
R(f) &= \mathbf{E}\left[ \mathrm{I}\left\{ -\sum_{j=1,\, j\neq k}^{n} c_j + p_k(c_k, X) \geq B \right\} \middle| Z_k(c_k, X) \wedge \neg Z(f \ominus k, X) \right] \\
&\quad \text{if } \sum_{j=1}^{n} c_j < b \\
R(f) &= \mathbf{E}\left[ \mathrm{I}\left\{ -\sum_{j=1,\, j\neq k}^{n} c_j + p_k(c_k \ldots, X) \geq B \right\} \middle| Z_k(c_k \ldots, X) \wedge \neg Z(f \ominus k, X) \right] \\
&\quad \text{if } \sum_{j=1}^{n} c_j = b \\
R(s) &= 0.
\end{aligned}$$

In this case, $\Pi$ is

$$\Pi_k(f, X) = \mathrm{I}\left\{ -\sum_{j=1,\, j\neq k}^{n} c_j + p_k(c_k, X) \geq B \right\}.$$

Probability is represented here as an expected value of an identity function. These two representations are equivalent, as Theorem B.2.2 states. For bound optimization, there is no point to have $\tau$ greater than $-B$[1]. The schedule after that does not affect the failure profit of the PPA. Let $\tau = -B$. Then the rewards become

$$\begin{aligned}
R(f) &= 1 \text{ if } \sum_{j=1}^{n} c_j < b \\
R(f) &= 0 \text{ if } \sum_{j=1}^{n} c_j = b \\
R(s) &= 0.
\end{aligned}$$

**Definition 7.2.1.** $l$-th *level* in the preceding MDP is the set of states where

$$\sum_{j=1}^{n} c_j = l.$$

**Lemma 7.2.2.** *Each dynamic schedule maps to a policy in the proposed MDP. Conversely, each policy in the proposed MDP maps to a dynamic schedule.*

*Proof.* A policy may be transformed to a schedule as follows. Let each state represent a hypothetical time instant during an hypothetical solution process. More specifically, the state represents number of intervals allocated to each algorithm until the state's time instant. The problem instance is unsolved in states of $S$, and solved in states of $F$. The action $A_j$ is equivalent to running the algorithm $e_j$ in the time instance of the state. Given a solution of the MDP is in form of an optimal actions for all states. Then, the schedule for the policy is the following. Let $(A_{j_1}, \ldots, A_{j_b})$ be the optimal sequence of actions for states $(s_1, \ldots, s_b)$, where $s_{l+1}$ is the non final successor of $s_l$. Then, the equivalent schedule for the model is the sequence $(e_{j_1}, \ldots, e_{j_b})$. Each element determines which algorithm has a non-zero resource allocation function in that time interval. The

[1] Notice, that in this case only negative bound makes since, as all profits are negative.

reverse transformation is similar. Though for a single schedule, there are more than one policies in the MDP, differing in actions in states unreachable from the initial one. Therefore, their profit is the same. □

In the following, we use $\Pi_S$ to denote the abstract profit[2] of a PPA with schedule $S$. In this case the PPA is considered to behave as a single algorithm. Its abstract profit on each instance is equal to the abstract profit of the algorithm that solves the instance. In fact, the optimal dynamic schedule is the schedule with the maximal average abstract profit in the initial state ($\mathbf{E}\left[\Pi_S(s_0 \ldots, X)\right]$). This is because the probability for bound optimization can be represented as an expected value, by Theorem B.2.2.

**Lemma 7.2.3.** *Utility $U$ of a state $s = (c_1, \ldots, c)n)$ in the MDP defined above with a fixed policy is*

$$U(s) = \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, \neg Z(s, X)\right],$$

*where $S$ is a schedule constructed from an MDP policy as defined in Lemma 7.2.2.*

*Proof.* We prove the lemma by a backward induction on the levels of states. First, clearly $U(f) = R(f)$. As the base of the induction, take $b-1$ level of state set $S$. For each state $s$ in this level, let the policy define independently an arbitrary action $A_k$. The possible successor state is $f$. Notice that $s = f \ominus k$. The utility of $s$ is ,by definition of reward,

$$\begin{aligned} U(s) &= U(f) = \mathbf{E}\left[\Pi_k(f \ldots, X) \,|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ &= \mathbf{E}\left[\Pi_S(f \ldots, X) \,|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right], \end{aligned}$$

because the profit of a PPA with schedule $S$ on instances solved by $e_k$ is equal to the profit of $e_k$. Each instance belongs to at least one interval, because the last interval spans to infinity. This shows the induction base. The inductive step can be proved as follows. Let $s$ be any state and let $s' \in S$ and $f' \in F$ be the subsequent states, given an action $A_k$. Clearly, $s$ is reached only when $\neg Z(s, X)$. Then

$$\begin{aligned} U(s) &= \mathbf{P}\left[f' \,\middle|\, s, A_k\right] U(f') + \mathbf{P}\left[s' \,\middle|\, s, A_k\right] U(s') \\ &= \mathbf{P}\left[Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_k(f', X) \,\middle|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ &\quad + \mathbf{P}\left[\neg Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_S(s' \ldots, X) \,\middle|\, \neg Z_k(c_k, X) \wedge \neg Z(s, X)\right], \end{aligned}$$

by the inductive assumption and the definition of MDP. Then, because $\Pi_S = \Pi_k$ if $e_k$ solves the instance, we have

$$\begin{aligned} U(s) &= \mathbf{P}\left[Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_S(f' \ldots, X) \,\middle|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ &\quad + \mathbf{P}\left[\neg Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_S(s' \ldots, X) \,\middle|\, \neg Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ &= \mathbf{P}\left[Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ &\quad + \mathbf{P}\left[\neg Z_k(c'_k, X) \,\middle|\, \neg Z(s, X)\right] \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, \neg Z_k(c_k, X) \wedge \neg Z(s, X)\right], \end{aligned}$$

because it can be shown for both mean and bound optimization that

$$\begin{aligned} \mathbf{E}\left[\Pi_S(f' \ldots, X) \,\middle|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] &= \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, Z_k(c_k, X) \wedge \neg Z(s, X)\right] \\ \mathbf{E}\left[\Pi_S(s' \ldots, X) \,\middle|\, \neg Z_k(c_k, X) \wedge \neg Z(s, X)\right] &= \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, \neg Z_k(c_k, X) \wedge \neg Z(s, X)\right]. \end{aligned}$$

Then by Theorem B.2.1 we get

$$U(s) = \mathbf{E}\left[\Pi_S(s \ldots, X) \,|\, \neg Z(s, X)\right],$$

what proves the theorem. □

[2] Empirical mean profit or empirical failure profit.

```
$t \leftarrow b - 1$
**while** $t \geq 0$ **do**
  **for all** States $s = (c_1, \dots, c_n)$ where $\sum_{j=1}^{n} c_j = t$ **do**
    **for all** Actions $e_k \in E$ **do**
      Calculate reward $R(f)$ according to $\Pi$
      Calculate utility $U(s)$
    **end for**
    Pick the best action $e$
  **end for**
  $t \leftarrow t - 1$
**end while**
Return schedule composed of best actions starting at state $(0, \dots, 0)$
```

***Figure 7.1:*** *DP-DPPA for both mean optimization and bound optimization.* $\Pi$ *determines the actual type of optimization.*

**Theorem 7.2.4.** *Let the algorithms be schedulable. A schedule based on the optimal policy of the formulated MDP has optimal mean or bound profit.*

*Proof.* From Lemma 7.2.3, the utility of the first state

$$U(s_0) = \mathbf{E}\left[\Pi_S(s_0 \dots, X) \,|\, \neg Z(s_0, X)\right] = \mathbf{E}\left[\Pi_S(s_0 \dots, X)\right].$$

This is equivalent to the empirical mean of bound profit of the schedule. Because, there is an MDP policy for each schedule, the optimal policy also defines the schedule with maximal profit. □

The approach presented in this chapter differs from (Petrik, 2005) in not assuming the instances to be partitioned into classes for each algorithm[3]. Therefore, it offers a more precise schedule, for the cost of increased complexity of the optimal calculation. This is because the solved instances must be determined individually for each state. Partitioning the instances into classes allowed pre-processing of the runtime data. Moreover, (Petrik, 2005) did not consider bound optimization.

## 7.3 Scheduling Algorithm

The formulation according to Theorem 7.2.4 allows direct solution by any general MDP solving algorithm. The standard method for solving MDP is the already mentioned dynamic programming, sometimes called Bellman update. An excellent introduction to MDPs and solution methods is (Kall & Wallace, 1994; Sutton & Barto, 1998b). DP-DPPA, in Figure 7.1, is very similar to the generic Bellman update, only it is somewhat simpler because the MDP does not contain any loops.

**Theorem 7.3.1.** *Let the algorithms be schedulable. DP-DPPA solves the scheduling problem in time*

$$O\left(m * n^2 * \binom{b}{n} * n! * b\right).$$

*Proof.* Correctness of DP-DPPA follows directly from Theorem 7.2.4. Regarding the complexity, notice there are at most $b$ levels to be calculated. The number of states per level increases up to the last level, which has $\binom{b}{n} * n!$ states. There are $\mathrm{O}(n)$ possible actions per a single state and evaluation of rewards takes $\mathrm{O}(m * n)$ operations. The theorem follows. □

[3] Equivalent of the classification from CMA.

```
$t \leftarrow b - 1$
while $t \geq 0$ do
  for all States $s = (c_1, \ldots, c_n)$ where $\sum_{j=1}^{n} c_j = t$ do
    for all Actions $e_k \in E$, except those with smaller index than the last non-zero component of $s$ do
      if $t = b - 1$ then
        $R(f) \leftarrow 0$
      else
        $R(f) \leftarrow 1$
      end if
      Calculate utility $U(s)$
    end for
    Pick the best action $e$
  end for
  $t \leftarrow t - 1$
end while
Return schedule composed of best actions starting at state $(0, \ldots, 0)$
```

***Figure 7.2:*** *DP-BDPPA, an algorithm for finding bound optimization dynamic schedules, if $\tau = -B$.*

Though the algorithm DP-DPPA works for both bound and mean optimization, for bound optimization, it can be improved to eliminate unnecessary steps. The following lemma indicates how.

**Lemma 7.3.2.** *Consider a PPA of deterministic algorithms. The order of execution of algorithms before $B$ does not affect the empirical failure profit of the PPA.*

*Proof.* Because exchanging scheduled algorithms for any two intervals before the bound does not change the profit, the theorem follows. □

The modified algorithm for bound optimization, DP-BDPPA, is in Figure 7.2. It does not iterate over all actions for each state. It explores only the actions that do not have a lower index than the preceding action.

**Theorem 7.3.3.** *Let the algorithms be schedulable and $\tau = -B$. Then DP-BDPPA solves the scheduling problem with the equal profit as DP-DPPA. Though the actual schedule may differ.*

*Proof.* Clearly, DP-BDPPA explores a subspace space of possible schedules in the same way as DP-DPPA. We need to show that the explored space contains a schedule with the same profit as the optimal one. Therefore, let $S^*$ be the optimal schedule determined by DP-BDPPA. The intervals of the algorithm before $\tau$ may be reorganized, such that the indexes of algorithms ordered by time form a nondecreasing sequence. By Lemma 7.3.2, the profit of the schedule is unchanged. Moreover, this is a schedule that may be a result of DP-DPPA algorithm. □

*Remark* 7.3.4. It may be also interesting to explore how the order of algorithms influence the generalization capability of a parallel portfolio.

## 7.4 Approximation

In this section, we show that for *mean optimization*, increasing the length of intervals may lead to an algorithm that is polynomial in the number of the intervals $b$.

Assume a fixed number of algorithms. Then, by Theorem 7.3.1, the worst-case complexity of DP-DPPA is polynomial in terms of $b$. Because $b$ may be encoded in binary, the complexity of the scheduling algorithm is exponential in terms of the input length. However, the complexity can be reduced by giving up some precision. Then, the optimal schedule can be found in the time independent of $b$, and only polynomially increasing in the degree of precision. In the following, *task* is calculation of an algorithm that is scheduled for an interval. As the approximation algorithm is for mean optimization only, $P(S)$ denotes the mean profit, thus

$$P(S) = \frac{1}{m} \sum_{i=1}^{m} P(S, x_i).$$

**Lemma 7.4.1.** *Let $S$ and $S'$ be the optimal dynamic schedules with $z$ and $z' = c * z$ respectively, where $c \in \mathbb{I}_+$. Then we have*

$$-P(S') \leq -P(S) + (c-1) * (n-1) * z.$$

*Proof.* We prove the theorem by transforming the schedule $S$ to a new schedule $S'$ that is valid for $z'$. Then, we show that $-P(S')$ is bounded from above. The basic idea to create $S'$ by swapping tasks from $S$ to ensure that switches between any two algorithms take place only at times divisible by $z'$. It is possible to create a schedule with delay in expected time of at most $(c-1)*(n-1)*z$ as follows.

Let $S'$ initially be an identical copy of $S$. Then, apply the following procedure. Start with $l = 0$. The tasks from in time $z' * l$ to time $z' * (l+1)$ may remain in place if they are scheduled for the same algorithm. Otherwise, assume without the loss of generality, that a task $t_1$ at the time $z' * l$ is scheduled for an algorithm $e$. In addition, assume that tasks $t_2, \dots, t_c$ are the first following tasks scheduled for $e$. Use empty tasks when there are no such finitely long tasks. Then schedule the tasks $t_1, \dots, t_c$ from time $z' * l$ to the time $z' * (l+1)$, preserving the original order. The tasks originally scheduled for these intervals are pushed for later, pushing also other task, to fill intervals freed by $t_2, \dots, t_c$. The ordering of the tasks should be preserved during the pushing. The procedure is depicted in Figure 7.3. The process then continues identically at the time $z' * (l+1)$ until the end of schedule is reached.

Most importantly, in the above construction, each algorithm can cause delay for each task at most once. Each delay is by at most $(c-1)z$. Therefore, the expected execution time is increased by at most $(c-1)*(n-1)*z$. □

**Theorem 7.4.2.** *Let $-P(S) \geq b$ for all $S$. There is a $\epsilon$-approximate algorithm that solves the scheduling problem for any $\epsilon$ such that*

$$\epsilon = \frac{(n-1)*(c-1)}{b} \text{ for all } c \in \mathbb{I}. \tag{7.4.1}$$

*The approximate algorithm is polynomial in both $\frac{1}{\epsilon}$ and the input length of the instance, given a constant number of algorithms.*

The algorithm is similar to a FPTAS, except it allows approximations only for discrete values of $\epsilon$.

*Proof.* The approximate algorithm is based on DP-DPPA, only using $z' := \epsilon * z$. The proof of the correct approximation and complexity is very similar to the proof of FPTAS for knapsack (Elmieh, 1999).

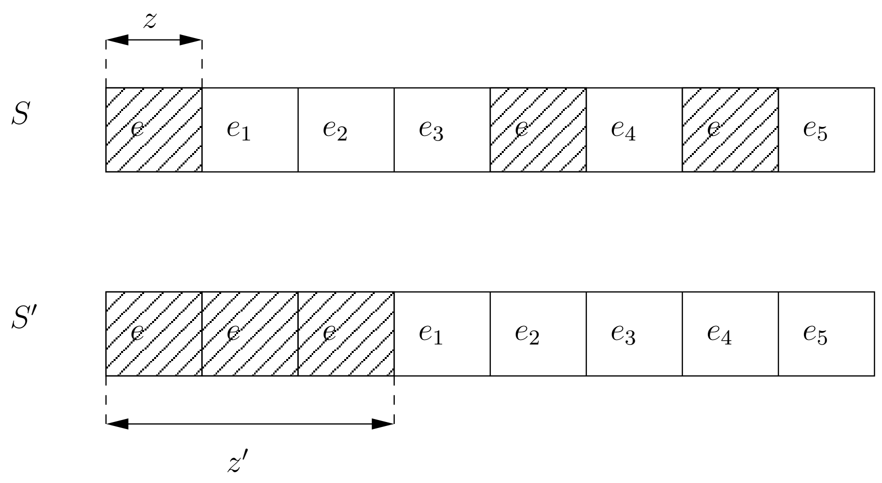

***Figure 7.3:*** *Illustration of the iterative step from proof of Lemma 7.4.1 when creating $S'$ from $S$. The picture shows intervals of the dynamic schedule and the algorithms scheduled for them.*

To prove the correct approximation, assume $S$ to be the optimal schedule calculated by DP-DPPA using $z$ and $S'$ to be the optimal schedule calculated using $z'$. Then by Lemma 7.4.1

$$-P(S') \leq -P(S) + (n-1) * (c-1) * z.$$

By definition, the algorithm is $\epsilon$-approximate if

$$-P(S) * (1+\epsilon) \geq -P(S').$$

Therefore, to show that the algorithm is approximate we have

$$\begin{aligned} -P(S) * (1+\epsilon) &\geq -P(S) + (n-1)(c-1)z \\ &\geq -P(S) + \epsilon * b \\ -P(S) &\geq b, \end{aligned}$$

what is assumption of the theorem.

The complexity is proved by observing how it depends on $\epsilon$. Trivially, the complexity of this modified algorithm is equal to the original algorithm only that $b' := \frac{b}{c}$. From (7.4.1) we have

$$\epsilon = \frac{(n-1)(c-1)}{b} \leq \frac{nc}{b}.$$

Thus

$$\frac{b}{c} \leq \frac{n}{\epsilon}.$$

Then, by Theorem 7.3.1 the complexity is

$$\begin{aligned} &\mathrm{O}\left(m * n^2 * \binom{\frac{b}{c}}{n} * n! * \frac{b}{c}\right) \\ \leq\ &\mathrm{O}\left(m * n^2 * \binom{\frac{n}{\epsilon}}{n} * n! * \frac{n}{\epsilon}\right), \end{aligned}$$

what is polynomial in $\frac{1}{\epsilon}$. This proves the theorem. □

## 7.5 Summary

In this chapter we introduced a method for finding optimal dynamic schedules. As the main result, dynamic schedules may be represented as Markov Decision Processes (MPD), and therefore calculated using the standard methods. Because calculation of a schedule depends exponentially on the number of algorithms and the length of the schedules, we also provided an approximate algorithm for mean optimization. For bound optimization, we presented a simplified algorithm as well.

Dynamic schedules, unlike their static counterparts can take advantage of diverse profit profiles of algorithms and do not require homogeneous resource profit functions. However, their calculation, even when formulated as MDP, is very time intensive. Moreover, since they can fit the distribution better, their generalization properties are more questionable. As a result, static schedules could be more beneficial for most domains. A possible combination of the static and dynamic schedules would be to run more calculate the steps in the method as static schedules. However, this approach would require an extensive analysis.

The method of finding optimal dynamic schedules, presented in this chapter, requires knowledge of the full algorithm statistics. It is suitable only for a problem domain with a static instance distribution and a large number of available training instances. For dynamically changing distribution, reinforcement learning (RL) may be used. RL methods solve the same problem as MDP, only probabilities and rewards do not need to be known in advance. They are discovered during the actual execution. Application of RL would lead to a truly adaptive scheduling algorithm. There is also a large number of Monte Carlo simulation algorithms capable of finding an optimal policy for large MDPs in short times. A good overview is (Sutton & Barto, 1998b). Static schedules for can be re-calculated quickly using CMA. Therefore, the problem with a dynamic instance distribution is not that interesting for further research.

*Chapter 8*

# GENERALIZATION BOUNDS

In this chapter, we theoretically analyze the proposed PPA learning schemes within the framework from Chapter 3. The primary importance is to show that a reasonable number of samples is sufficient to learn a PPA that is really useful for the whole problem distribution. This is the same assurance that is important for concept learning. Because the bounds are loose and are for the worst-case, they are not very practical for computing the specific bounds. In the whole chapter, we assume that the profit is positive. The theorems would hold, with minor modifications, for the fully negative case too.

## 8.1 General Results

In this section we provide a simple bounds for cases with finite number of potential algorithms $\Phi$. Except being generally useful, it can be used for dynamic schedules. However, because the space of static schedules is infinite, we provide a conceptually different analysis for those in the next section.

### 8.1.1 Mean Optimization

The simplest way to determine the quality of an algorithm is to use its profit on the training set. In fact, this profit is equal to $F_m(f)$. We call this learner *empirical profit maximizer.* In the next, we show that even this simple approach leads to finding an algorithm that closely approximates the optimal algorithm. However, these assurances assume that the set of all algorithms is finite and the profit is bounded.

**Theorem 8.1.1.** *Let the profit of each algorithm be bounded by $\langle a, b\rangle$. Then a learner that maximizes the mean profit needs*

$$m \geq \frac{(b-a)^2}{2 * \epsilon}(\ln|\Phi| + \ln\frac{1}{\delta})$$

*samples to fulfill the generalization bound from Def. 3.2.1. Therefore, if the set of available algorithms is finite, the learner mean generalizes.*

*Proof.* Let $f(X)$ be a profit function that fulfills the assumptions of the Hoeffding's Bound, Thm. B.2.4. Now, let

$$F_m(f) = \frac{1}{m}\sum_{i=1}^{m} f(x_i).$$

Therefore, the bound for $F_m(f)$, according to Thm. B.2.4, is

$$\mathbf{P}\left[F_m(f) - \mathbf{E}\left[F_m(f)\right] \geq \frac{\epsilon'}{m}\right] \leq \exp\left(\frac{-2\epsilon'^2}{\sum_{i=1}^{m}(b_i - a_i)^2}\right).$$

Given $\epsilon = \frac{\epsilon'}{m}$ and assumed bounds for $f(X)$ we get

$$\mathbf{P}\left[F_m(f) - \mathbf{E}\left[F_m(f)\right] \geq \epsilon\right] = \mathbf{P}\left[F_m(f) - \mathbf{E}\left[f(X)\right] \geq \epsilon\right] \leq \exp\left(\frac{-2m\epsilon}{(b-a)^2}\right).$$

Since there are $|\Phi|$ available algorithms, the probability of at least one having a worse deviation from the sample profit is bound by the following, using Boole's Bound, Theorem B.2.6.

$$\mathbf{P}\left[\max_{f\in\Phi} F_m(f) - \mathbf{E}\left[f(X)\right] \geq \epsilon\right] \leq |\Phi| \exp\left(\frac{-2m\epsilon}{(b-a)^2}\right).$$

To prove the theorem, we need to show that the probability may be made arbitrarily small by choosing enough samples

$$\begin{aligned}
\delta &\geq |\Phi| \exp\left(\frac{-2m\epsilon}{(b-a)^2}\right) \\
\ln\frac{1}{\delta} + \ln|\Phi| &\leq \frac{2m\epsilon}{(b-a)^2} \\
m &\geq \frac{(b-a)^2}{2\epsilon}\left(\ln\frac{1}{\delta} + \ln|\Phi|\right).
\end{aligned}$$

This proves the theorem, because the number of samples is polynomial in all required attributes. □

**Theorem 8.1.2.** *Let the profit of each algorithm be bounded by $\langle a, b\rangle$. Then, the learner that globally maximizes mean profit requires*

$$m \geq \frac{(b-a)^2}{2 * \epsilon}\left(\ln\left(|\Phi| + 1\right) + \ln\frac{1}{\delta}\right)$$

*samples, to satisfy the bound from Def. 3.2.2. Therefore, if the set of available algorithms is finite, the learner is mean optimal.*

*Proof.* Let $f^*$ be the globally optimal algorithm from $\Phi$. Its empirical profit is

$$F_m^* = F_m(f^*) = \frac{1}{m}\sum_{i=1}^{m} f^*(x_i).$$

Clearly, we are concerned only about the algorithms $f$ that offer a higher profit on training samples than the optimal one, so

$$F_m(f) \geq F.$$

From Theorem 8.1.1, we get:

$$\mathbf{P}\left[\max_{f\in\Phi} F_m(f) - \mathbf{E}\left[f(X)\right] \geq \epsilon_2\right] \leq |\Phi| \exp\left(\frac{-2m\epsilon_2}{(b-a)^2}\right).$$

Next, similarly to proof of Theorem 8.1.1 we get by Hoeffding's Bound

$$\mathbf{P}\left[\mathbf{E}\left[f^*(X)\right] - F_m(f^*) \geq \epsilon_1\right] \leq \exp\left(\frac{-2m\epsilon_1}{(b-a)^2}\right).$$

Then

$$\begin{aligned}
\mathbf{E}\left[f^*\right] - \mathbf{E}\left[f\right] &= \mathbf{E}\left[f^*\right] + F_m(f^*) - F_m(f^*) - \mathbf{E}\left[f\right] \\
&\leq (\mathbf{E}\left[f^*\right] - F_m(f^*)) + (F_m(f) - \mathbf{E}\left[f\right]).
\end{aligned}$$

Therefore, assuming $\epsilon = \epsilon_1 + \epsilon_2$, we get

$$\begin{aligned}
&\mathbf{P}\left[\max_{f\in\Phi}\mathbf{E}\left[f^*\right] + \mathbf{E}\left[f\right] \geq \epsilon\right] \\
\leq\ & \mathbf{P}\left[\max_{f\in\Phi}(\mathbf{E}\left[f^*\right] - F_m^*) + (F_m(f) - \mathbf{E}\left[f\right]) \geq \epsilon\right] \\
\leq\ & \mathbf{P}\left[\max_{f\in\Phi}\mathbf{E}\left[f^*\right] - F_m^* \geq \epsilon_1\right] + \mathbf{P}\left[\max_{f\in\Phi}F_m(f) - \mathbf{E}\left[f\right] \geq \epsilon_2\right] \\
\leq\ & \exp\left(\frac{-2m\epsilon_1}{(b-a)^2}\right) + |\Phi|\exp\left(\frac{-2m\epsilon_2}{(b-a)^2}\right) \\
\leq\ & (|\Phi|+1)\exp\left(\frac{-2m\epsilon}{(b-a)^2}\right).
\end{aligned}$$

Hence the theorem. □

*Remark* 8.1.3. To make the analysis practical for FPTAS learning algorithms, the additive $\epsilon$ must be changed to multiplicative. The multiplicative bound may then be derived from Chernoff's Bound, Theorem B.2.3.

### *8.1.2 Limit Optimization*

As with mean optimization, the simplest approach of maximizing the minimal empirical profit leads to a proper generalization, given the number of algorithms to chose from is finite.

**Theorem 8.1.4.** *A learner maximizing the minimal empirical profit requires*

$$m \geq \frac{1}{\epsilon}\left(\ln|\Phi| + \ln\frac{1}{\delta}\right)$$

*samples to fulfill the bound from Def. 3.3.1. Therefore, if the set of available algorithms is finite, the learner limit generalizes.*

*Proof.* To prove the generalization, we need to show that in the worst case the probability bounds hold. Therefore, we show that maximal probability for all algorithms that satisfy the condition of being learned is bounded by the theorem's inequality.

$$\begin{aligned}
&\mathbf{P}\left[\max_{f:f\in\Phi}\{\mathbf{P}\left[f(X) < G_m(f)\right]\} > \epsilon\right] \\
=\ & \mathbf{E}\left[\mathbf{P}\left[\max_{f:G_m(f)=Z}\{\mathbf{P}\left[f(X) < Z\right]\} > \epsilon \,\middle|\, Z\right]\right] \\
\leq\ & \sup_{Z\in\langle a,b\rangle}\mathbf{P}\left[\max_{f:G_m(f)=Z}\{\mathbf{P}\left[f(X) < Z\right]\} > \epsilon \,\middle|\, Z\right].
\end{aligned}$$

Now, let us derive the value of the probability, depending on $Z$.

$$\begin{aligned}
&\mathbf{P}\left[\max_{f:G_m(f)=Z}\{\mathbf{P}\left[f(X)<Z\right]\}>\epsilon\right] \\
&= \mathbf{E}\left[\mathrm{I}\left\{\max_{f:G_m(f)=Z}\mathbf{P}\left[f(X)<Z\right]>\epsilon\right\}\right] \\
&= \mathbf{E}\left[\max_{f\in\Phi}\mathrm{I}\left\{G_m(f)=Z\right\}\mathrm{I}\left\{\mathbf{P}\left[f(X)<Z\right]>\epsilon\right\}\right] \\
&\le \sum_{f\in\Phi,\mathbf{P}[f(X)<Z]>\epsilon}\mathbf{P}\left[G_m(f)=Z\right] \\
&\le \sum_{f\in\Phi,\mathbf{P}[f(X)<Z]>\epsilon}\mathbf{P}\left[G_m(f)\ge Z\right] \\
&= |\Phi|(1-\epsilon)^m.
\end{aligned}$$

Since the value of the probability does not depend on $Z$, it is also the value of the expectation. Now, the above upper bound can be simplified to (Mitchell, 1997):

$$|\Phi|\exp\left(-\epsilon n\right).$$

The theorem follows by algebraic operations. □

**Theorem 8.1.5.** *A learner that globally maximizes the minimal empirical profit needs the following number of samples to fulfill the bound from Def. 3.3.2.*

$$m \ge \frac{1}{\epsilon}\left(\ln|\Phi| + \ln\frac{1}{\delta}\right).$$

*Therefore, if the set of available algorithms is finite, the learner is limit optimal.*

*Proof.* To prove the theorem we need to bound the fraction of instances that are solved with smaller profit than the infimum of the optimal algorithm on $\mathcal{I}$. Let the minimal empirical profit of the optimal algorithm be $G_m^* = G_m(f^*) = \min_i f^*(x_i)$. Clearly,

$$\inf_{x\in\mathcal{I}} f^*(x) \le G_m^* \le G_m(f),$$

otherwise $f^*$ would be chosen as the optimum. Therefore, this theorem is a weaker version of Theorem 8.1.4 and number of samples is identical to result of it. □

### *8.1.3 Bound Optimization*

As before, it is easy to show that, given a finite set of available algorithms, a learner that minimized the empirical failure profit generalizes well.

**Lemma 8.1.6.** *The empirical failure profit $H_m(f)$ is an unbiased estimator of the true failure profit* $\mathbf{P}\left[f(X)<B\right]$.

*Proof.* Clearly

$$\begin{aligned}
\mathbf{E}\left[H_m\right] &= \mathbf{E}\left[\frac{1}{m}\sum_{i=1}^{m}\mathrm{I}\left\{f(x_i)\le B\right\}\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbf{E}\left[\mathrm{I}\left\{f(x_i)\le B\right\}\right] \\
&= \mathbf{E}\left[\mathrm{I}\left\{f(X)<B\right\}\right] \\
&= \mathbf{P}\left[f(X)<B\right].
\end{aligned}$$

Thus the theorem holds. □

For simplicity, we define

$$H(f) = \mathbf{P}\left[f(X) < B\right] = \mathbf{E}\left[\mathrm{I}\left\{f(X) < B\right\}\right].$$

Please notice that according to Lemma 8.1.6

$$H(f) = \mathbf{E}\left[H_m(f)\right].$$

Given this notation, the problem partially reduces to the agnostic learning problem (Mitchell, 1997). As a result, our proofs follow a similar path as the proofs in agnostic learning.

**Theorem 8.1.7.** *A learner that minimizes empirical failure profit requires*

$$m \geq \frac{1}{2\epsilon^2}\left(\ln|\Phi| + \ln\frac{1}{\delta}\right)$$

*samples to fulfill the bound from Def. 3.4.1. Therefore, if the set of available algorithms is finite, the learner bound generalizes.*

*Proof.* The proof follows from Theorem B.2.4 (Hoeffding's Bound). The basic line follows, assuming the worst case that all algorithms have the same empirical failure profit.

$$\begin{aligned}
&\mathbf{P}\left[\max_{f\in\Phi}\left\{H(f) - H_m(f)\right\} > \epsilon\right] \leq \\
&\leq \quad |\Phi| \max_{f\in\Phi}\mathbf{P}\left[H(f) - H_m(f) > \epsilon\right] \\
&\leq \quad |\Phi| \max_{f\in\Phi}\mathbf{P}\left[n * H(f) - n * H_m(f) > n * \epsilon\right] \\
&\leq \quad |\Phi| \exp\left(\frac{-2(n\epsilon)^2}{n(b-a)}\right) = \exp\left(\frac{-2m\epsilon^2}{(b-a)}\right) \\
&= \quad |\Phi| \exp\left(\frac{-2(n\epsilon)^2}{n}\right) = \exp\left(-2m\epsilon^2\right).
\end{aligned}$$

This leads straight forward to the bound from theorem. □

**Theorem 8.1.8.** *A learner that globally minimizes empirical failure profit requires*

$$m \geq 2\frac{1}{\epsilon^2}\left(\ln|\Phi| + \ln\frac{1}{\delta} + \ln 2\right)$$

*samples to fulfill the bound from Def. 3.4.2. Therefore, if the set of available algorithms is finite, the learner is bound optimal.*

*Proof.* Let $f^*$ be the globally optimal algorithm. Since the learner is an empirical failure profit minimizer, we get

$$H_m(f^*) \geq H_m(f).$$

Now, assuming the worst case of all algorithms having the same empirical failure profit, we get

$$
\begin{aligned}
&\mathbf{P}\left[\max_{f\in\Phi}\mathbf{P}\left[f(X)<B\right]-\mathbf{P}\left[f^*(X)<B\right]>\epsilon\right]\\
\leq\ & |\Phi|\max_{f\in\Phi}\mathbf{P}\left[\mathbf{P}\left[f(X)<B\right]-\mathbf{P}\left[f^*(X)<B\right]>\epsilon\right]\\
=\ & |\Phi|\max_{f\in\Phi}\mathbf{P}\left[H(f)-H(f^*)>\epsilon\right]\\
\leq\ & |\Phi|\max_{f\in\Phi}\mathbf{P}\left[H(f)-H_m(f)+H_m(f^*)-H(f^*)>\epsilon\right]\\
\leq\ & |\Phi|\max_{f\in\Phi}\left(\mathbf{P}\left[H(f)-H_m(f)>\frac{\epsilon}{2}\right]+\mathbf{P}\left[H_m(f^*)-H(f^*)>\frac{\epsilon}{2}\right]\right)\\
\leq\ & 2*|\Phi|\exp\left(\frac{-n\epsilon^2}{2(b-a)}\right)\\
=\ & 2*|\Phi|\exp\left(\frac{-n\epsilon^2}{2}\right).
\end{aligned}
$$

The last inequality is derived in a similar way to the proof of Theorem 8.1.7, using both Hoeffding's Bounds. □

## 8.2 Dynamic Schedules

Since the number of potential dynamic schedules is finite, we can apply the theorems from the preceding section to this case. Notice however that $\Phi$ is not equal to $E$, but it is the set of all schedules. Because we defined dynamic schedules only for mean and bound optimization, we provide the specific generalization bounds only for these two optimization types. The following lemmas establish how many different dynamic schedules there are. We always assume that the algorithms that comprise the PPAs are fixed, and there is finitely many of them.

**Lemma 8.2.1.** *The number of dynamic schedules for mean optimization with fixed $z$, $b$, and $\tau$ is*

$$n^b.$$

*Proof.* There are $n$ algorithm that can be scheduled for each of the $b$ intervals. Thus the lemma. □

**Theorem 8.2.2.** *Let the profit of dynamic schedules be from $\langle a, b\rangle$. A learner that finds dynamic schedules with fixed $z$, $b$, and $\tau$ that globally minimize the empirical mean profit is mean optimal. Moreover, the learner requires*

$$m \geq \frac{(b-a)^2}{2*\epsilon}\left(b\ln(n+1)+\ln\frac{1}{\delta}\right)$$

*samples to fulfill the bound.*

*Proof.* The theorem follows from Theorem 8.1.2 and Lemma 8.2.1 and because

$$\frac{(b-a)^2}{2*\epsilon}\left(\ln\left(n^b+1\right)+\ln\frac{1}{\delta}\right)\leq\frac{(b-a)^2}{2*\epsilon}\left(b\ln(n+1)+\ln\frac{1}{\delta}\right).$$

□

**Lemma 8.2.3.** *The number of dynamic schedules for bound optimization with fixed $z$, $b$, and $\tau$ is*

$$b^n.$$

*Proof.* There are $b$ intervals that can be assigned to $n$ algorithms. Thus the lemma. □

**Theorem 8.2.4.** *A learner that finds dynamic schedules with fixed $z$, $b$, and $\tau$ that globally maximize the empirical failure profit is bound optimal. Moreover, the learner requires*

$$m \geq 2\frac{1}{\epsilon^2}\left(n \ln b + \ln \frac{1}{\delta} + \ln 2\right)$$

*samples to fulfill the bound.*

*Proof.* The theorem follows from Theorem 8.1.8 and Lemma 8.2.3. □

Notice, that for we used fixed $z$, $b$, and $\tau$ for the bounds. If these parameters are tuned according to the specific sample, the proposed generalization bounds do not hold. Moreover, better bounds may be possibly obtained by using an approach based on Vapnik-Chervonenkis theory, but it would require a more extensive analysis. This approach is used in the next section for static schedules.

## 8.3 Static Schedules

The number of static schedules for a given set of algorithms to comprise PPA is infinite. Therefore, the bounds from the preceding sections are not applicable in this case. The analysis we provide below is based on research on concept learning, which faces a similar problem. Because the static schedules were solved only for mean and limit optimization, we analyze only these two cases. Notice, that mean-optimality and limit-optimality assume the learning algorithm finds an empirically globally optimal algorithm. This is not the case of CMA. Thus the optimality bounds hold only for the globally optimal VOCMA algorithm. However, mean-generalization and limit-generalization bounds hold also for CMA.

**Theorem 8.3.1.** *Let the resource profit functions be schedulable and homogeneous. Let $\widehat{p}$, $\widehat{\mu}$, $\widehat{\nu}$ be the maximal values of corresponding functions. Also, let $\frac{m*\epsilon^2}{\widehat{p}^2} > 2$. Then, the generalization probability is*

$$\mathbf{P}\left[\sup_{f\in\Phi} |F_m(f) - \mathbf{E}[f]| > \epsilon\right] \leq 8(m+1)^{\binom{n}{2}} 2^n \exp\left(\frac{-m\epsilon^2}{32\widehat{p}}\left(\frac{1}{\widehat{\nu}\widehat{\mu}n}\right)^2\right).$$

*Proof.* We prove the theorem in 4 steps. All steps, except the 3rd one are almost identical to the proof of Thm. B.2.7 from (Devroye, Gyorfi, & Lugosi, 1996).

*STEP1: FIRST SYMMETRIZATION BY A GHOST SAMPLE.* Define new random variables $X'_1, \ldots, X'_m \in \mathcal{I}$, such that all variables $X_1, \ldots, X_m, X'_1, \ldots, X'_m$ are independent. $F'_m$ now represents the empirical performance of the algorithm on the ghost sample. We show now that

$$\mathbf{P}\left[\sup_{f\in\Phi} |F_m(f) - \mathbf{E}[f]| > \epsilon\right] \leq 2\mathbf{P}\left[\sup_{f\in\Phi} |F_m(f) - F'_m(f)| > \frac{\epsilon}{2}\right]. \quad (8.3.1)$$

To show this, let $f^* \in \mathcal{I}$ be a set for which $|F_m(f^*) - \mathbf{E}[f^*]| > \epsilon$ if such a set exists, and a fixed algorithm from $\mathcal{I}$ otherwise. Then

$$\begin{aligned}
&\mathbf{P}\left[\sup_{f\in\Phi}|F_m(f) - F'_m(f)| > \frac{\epsilon}{2}\right] \\
&> \mathbf{P}\left[|F_m(f^*) - F'_m(f^*)| > \frac{\epsilon}{2}\right] \\
&> \mathbf{P}\left[|F_m(f^*) - \mathbf{E}[f^*]| > \epsilon, |F'_m(f^*) - \mathbf{E}[f^*]| < \frac{\epsilon}{2}\right] \\
&= \mathbf{E}\left[\mathrm{I}\{|F_m(f^*) - \mathbf{E}[f^*]| > \epsilon\}\,\mathbf{P}\left[|F'_m(f^*) - \mathbf{E}[f^*]| < \frac{\epsilon}{2}\,\middle|\, X_1,\dots,X_m\right]\right].
\end{aligned}$$

Now, the conditional probability inside may be bounded by Chebyshev's Bound B.2.5 as

$$\begin{aligned}
&\mathbf{P}\left[|F'_m(f^*) - \mathbf{E}[f^*]| < \frac{\epsilon}{2}\,\middle|\, X_1,\dots,X_m\right] \\
&\geq 1 - \frac{\mathbf{E}[f^*](1-\mathbf{E}[f^*])}{\frac{m\epsilon}{2}} \\
&\geq 1 - \frac{\hat{p}^2}{m\epsilon^2} \geq \frac{1}{2},
\end{aligned}$$

whenever $\frac{m*\epsilon^2}{\hat{p}^2} > 2$. This shows (8.3.1).

*STEP2: SECOND SYMMETRIZATION BY RANDOM SIGNS.* Let $\delta_1,\dots,\delta_m$ be independent identically distributed variables independent from $X_1,\dots,X_m,X'_1,\dots,X'_m$ with

$$\mathbf{P}[\delta_i = -1] = \mathbf{P}[\delta_i = -1] = \frac{1}{2}.$$

Because $X_1,\dots,X_m,X'_1,\dots,X'_m$ are independent, the distribution of

$$\sup_{f\in\Phi}\left|\sum_{i=1}^m (f(X_i) - f(X'_i))\right|$$

is the same as distribution of

$$\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i(f(X_i) - f(X'_i))\right|.$$

Thus by STEP 1

$$\mathbf{P}\left[\sup_{f\in\Phi}|F_m(f) - \mathbf{E}[f]| > \epsilon\right] \leq 2\mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i(f(X_i) - f(X'_i))\right| > \frac{\epsilon}{2}\right].$$

By applying the union bound, we can remove the ghost sample $X'_1,\dots,X'_m$

$$\begin{aligned}
&\mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i(f(X_i) - f(X'_i))\right| > \frac{\epsilon}{2}\right] \\
&\leq \mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i f(X_i)\right| > \frac{\epsilon}{4}\right] + \mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i f(X'_i)\right| > \frac{\epsilon}{4}\right] \\
&\leq 2\mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^m \delta_i f(X_i)\right| > \frac{\epsilon}{4}\right].
\end{aligned}$$

*STEP3: CONDITIONING.* To bound the probability from STEP 2, we condition on the sample $X_1, \ldots, X_m$

$$\mathbf{P}\left[\sup_{f\in\Phi}\left|\sum_{i=1}^{m}\delta_i f(X_i)\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right].$$

Since, there are at most $(m+1)^{\binom{n}{2}}$ possible instance to algorithm classifications (Thm. 6.1.5) represented by $W$, we obtain

$$(m+1)^{\binom{n}{2}}\sup_W \mathbf{P}\left[\sup_S\left|\sum_{j=1}^{n}\nu_j r_j\sum_{i=1}^{m_j}\delta_{k(i,j)}\mu_j(X_{k(i,j)})\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right],$$

where $S$ defines only the resource assignments, not the classification. Function $k$, and $m_j$ are defined the same way as in Chapter 5. For the choice of schedule here, we do not require equality of resources to one, just to be smaller. The, there is $2^j$ possible distribution of signs to components of the sum over $j$. Therefore, also by homogeneity assumption

$$\begin{aligned}
&(m+1)^{\binom{n}{2}}\sup_W \mathbf{P}\left[\sup_{f\in S}\left|\sum_{j=1}^{n}\nu_j(r_j)\sum_{i=1}^{m_j}\delta_{k(i,j)}\mu_j(X_{k(i,j)})\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right]\\
\leq\ & (m+1)^{\binom{n}{2}}\sup_W 2^n\sup_S \mathbf{P}\left[\left|\sum_{j=1}^{n}\nu_j(r_j)\sum_{i=1}^{m_j}\delta_{k(i,j)}\mu_j(X_{k(i,j)})\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right]\\
\leq\ & (m+1)^{\binom{n}{2}}2^n\mathbf{P}\left[\left|\sum_{j=1}^{n}\widehat{\mu}\widehat{\nu}\sum_{i=1}^{m}\delta_i\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right].
\end{aligned}$$

*STEP4: HOEFFDING'S BOUND.* With $X_1, \ldots, X_m$ fixed, the probability is a sum of $m$ random variables and therefore can be bound by the Hoeffding's Bound B.2.4. Then

$$\mathbf{P}\left[\left|\sum_{j=1}^{n}\widehat{\mu}\widehat{\nu}\sum_{i=1}^{m}\delta_i\right| > \frac{\epsilon}{4}\middle| X_1,\ldots,X_m\right] \leq 2\exp\left(\frac{-m\epsilon^2}{32\widehat{p}}\left(\frac{1}{\widehat{\nu}\widehat{\mu}n}\right)^2\right).$$

Taking the expected value from both sides yields the result. □

**Lemma 8.3.2.** *Let $\widehat{f}$ be the algorithm that maximizes the empirical mean profit. Then,*

$$\mathbf{E}\left[f^*\right] - \mathbf{E}\left[\widehat{f}\right] \leq 2\sup_{f\in\Phi}\left|\mathbf{E}\left[f\right] - F_m(f)\right|.$$

*Proof.*

$$\begin{aligned}
\mathbf{E}\left[f^*\right] - \mathbf{E}\left[\widehat{f}\right] &\leq \mathbf{E}\left[f^*\right] - F_m(f^*) + F_m(f^*) - \mathbf{E}\left[\widehat{f}\right]\\
&\leq \mathbf{E}\left[f^*\right] - F_m(f^*) + F_m(\widehat{f}) - \mathbf{E}\left[\widehat{f}\right]\\
&\leq 2\sup_{f\in\Phi}\left|F_m(\widehat{f}) - \mathbf{E}\left[\widehat{f}\right]\right|.
\end{aligned}$$

□

**Corollary 8.3.3.** *Let $f$ be the algorithm that maximizes the empirical mean profit. Let $f^*$ be the mean optimal static schedule as in Definition 3.2.2. If the assumptions of Theorem 8.3.1 hold, then we have*

$$\mathbf{P}\left[\mathbf{E}\left[f^*(X)\right] - \mathbf{E}\left[f(X)\right] > \epsilon\right] \leq 8(m+1)^{\binom{n}{2}} 2^n \exp\left(\frac{-m\epsilon^2}{128\widehat{p}}\left(\frac{1}{\widehat{\nu}\widehat{\mu}n}\right)^2\right).$$

*Proof.* The corollary follows directly from application of Lemma 8.3.2 on the theorem. $\square$

**Theorem 8.3.4.** *Let the assumptions of Theorem 8.3.1 hold. A learner that finds a static schedule PPA by maximizing the empirical mean profit is mean optimal. Moreover, it needs*

$$\max\left(\frac{512\binom{n}{2}(\widehat{\nu}\widehat{\mu}n)^2\widehat{p}}{\epsilon^2}\ln\frac{256\binom{n}{2}(\widehat{\nu}\widehat{\mu}n)^2\widehat{p}}{\epsilon^2}, \frac{256(\widehat{\nu}\widehat{\mu}n)^2\widehat{p}}{\epsilon^2}\ln\frac{2^{n+3}}{\delta}\right)$$

*samples.*

*Proof.* The proof is based on Problem 12.5 from (Devroye et al., 1996). The bound on samples follows from Corollary 8.3.3 as shown next. Let

$$d = \frac{\epsilon^2}{\widehat{p}}\left(\frac{1}{\widehat{\nu}\widehat{\mu}n}\right)^2.$$

The, the applying the bound

$$(m+1)^{\binom{n}{2}} \leq \exp\left(\frac{md}{256}\right),$$

whenever $m \geq \frac{512\binom{n}{2}}{d}\ln\frac{256\binom{n}{2}}{d}$. This holds because $2\ln x \leq x$ if $x \geq e^2$. $\square$

Next, we show the generalization capabilities of limit optimization.

**Theorem 8.3.5.** *Let the resource profit functions be homogeneous and schedulable. Let $\widehat{p}$, $\widehat{\mu}$, $\widehat{\nu}$ be the maximal values of corresponding functions. Also, let $\frac{m*\epsilon^2}{\widehat{p}^2} > 2$. Then, the generalization probability is*

$$\mathbf{P}\left[\sup_{f\in\Phi}\mathbf{P}\left[f(X) < G_m(f)\right] \geq \epsilon\right] \leq 8(m+1)^n \exp\left(\frac{-m\epsilon^2}{32}\right).$$

*Proof.* First, by the proof of Thm. 8.1.4, we need to bound the probability

$$\mathbf{P}\left[\sup_{f\in\Phi}\mathbf{P}\left[f(X) < Z\right] > \epsilon \,\middle|\, Z\right],$$

where $\Phi$ is the set of static schedules. This set is infinitely large. In the following let $A$ denote the event that the algorithm portfolio solves the problem with the specified profit $Z$. Therefore, let $\Phi$ denote a set of these probabilistic events for all portfolios. Then the probability is a probabilistic measure on this set. The rest of the proof is based on the Vapnik-Chervonenkis Theorem B.2.9. To use this theorem, see that

$$s(\Phi, m) \leq (m+1)^n.$$

To show this, split the event $A$ to

$$A = A_1 \wedge \ldots \wedge A_n,$$

where $A_i$ represents the event that the $i$-the algorithm in the portfolio, with the given processor time assignment, solves an instance. Also, let $\Phi_i$ be set of $A_i$. Clearly, because of the homogeneity assumption,

$$s(\Phi_1, m) \leq m + 1.$$

Let $A' = (A_1, \ldots, A_n)$, and similarly $\Phi'$ be the set of these events. Because $\Phi' \subseteq \Phi$,

$$s(\Phi, m) \leq s(\Phi', m) = (m+1)^n.$$

Then, the inequality is

$$\mathbf{P}\left[\sup_{f \in \Phi} \mathbf{P}\left[f(X) < Z\right] > \epsilon \,\middle|\, Z\right] \leq 8(m+1)^n \exp\left(\frac{-m\epsilon^2}{32}\right),$$

from Theorem B.2.9. Taking expected value from both sides makes the proof complete. □

**Corollary 8.3.6.** *Let $f$ be the algorithm that maximizes the empirical limit profit. Let $f^*$ be the limit optimal static schedule as in Definition 3.3.2. If the assumptions of Theorem 8.3.5 hold, then we also have*

$$\mathbf{P}\left[\mathbf{P}\left[f(X) < \inf_Y f^*(Y)\right] \geq \epsilon\right] \leq 8(m+1)^n \exp\left(\frac{-m\epsilon^2}{128}\right).$$

*Proof.* The corollary holds by a similar argument as Lemma 8.3.2. □

**Theorem 8.3.7.** *Let the assumptions of Theorem 8.3.5 hold. A learner that finds a static schedule PPA by maximizing the minimal empirical profit is limit optimal. Moreover, it needs*

$$\max\left(\frac{512n}{\epsilon^2} \ln \frac{256n}{\epsilon^2}, \frac{256}{\epsilon^2} \ln \frac{8}{\delta}\right)$$

*samples.*

*Proof.* The theorem is result of Corollary 8.3.6. The sample bound can be obtained similarly as for Theorem 8.3.4. □

*Remark* 8.3.8. A tighter bound could be obtained using VC results that assume that the training set can be learned with zero training error, as in Section 12.7 of (Devroye et al., 1996).

The preceding theorems show that static schedules can generalize to the whole instance distribution, though there are infinitely many of them. The generalization capability is reduced dramatically with the increasing number of algorithms in the portfolio.

## 8.4 Summary

We provided theoretical bounds for the specified types of learning dynamic and static schedules, as well as for general algorithm learning problem. The importance of the results is that, for both static and dynamic schedules, the number of samples is polynomial in both precision and confidence of learning. In addition, the bounds are also polynomial in the number of algorithms in the PPA. Therefore, it is reasonable to assume that the schedules are learnable.

The space of possible dynamic schedules is finite. Therefore, the worst-case of required training instances to be polynomial in both number of algorithms and the number of intervals. This bound can be probably improved by the Vapnik-Chervonenkis techniques used for generalization of static schedules.

The space of possible static schedules is infinite, what makes the approach presented before unusable. Inspired by the work on Vapnik-Chervonenkis dimension, we showed that it is also possible to learn the static schedules with a polynomial number of training instances in the number of samples.

The presented bounds are not tight and offer only a worst case analysis. They are not necessarily always practical. Despite the bound are polynomial, they still require a huge number of samples for a reasonable generalization probability.

*Chapter 9*

# APPLICATION: SAT PROBLEM SOLVING

In this chapter we the demonstrate practical applicability of Parallel Portfolios of Algorithms. As the application we use the Satisfiability problem (SAT). The purpose SAT problem is to determine whether a formula in propositional logic is satisfied for at least one interpretation. There are two main variants of the problem, characterized by their formulations. One is Conjunctive Normal Form (CNF) and the other is Disjunctive Normal Form (DNF). In CNF, the formula is provided as a conjunction of clauses and in DNF as a disjunction of terms.

SAT is an important problem in reasoning, planning and generally in artificial intelligence (Russell & Norvig, 2002). Moreover, SAT was the first problem to be shown NP complete (Cook, 1971).

## *9.1 Problem Setup*

In general, evaluation of algorithm performance takes a lot of processor time. Therefore, we used data from SAT-Ex, a library of performance of 23 cutting edge SAT algorithms (Simon, 2005). The instances were, as well as the algorithms, taken from the Sat-Ex project. The whole set consists of 1303 instances, and total processor time spent on running the instances on was more than 529 days on a P-II 400Mhz machine. The cutoff time for execution was 10000 seconds.

We evaluate the performance of algorithms by simply comparing their statistical results on the set of instances. We assign a execution time of 20000 seconds for problem instances that were not solved within the provided 10000 seconds. Obviously, this may skew the results. A more involved approach that addresses this problem was proposed by (Etzioni & Etzioni, 1994).

Implementation of algorithms to finds the optimal schedules for PPA is in Appendix D. Profit of PPA is calculated only theoretically on the available problem instanced. Thus, the portfolios were only simulated, not really implemented.

We used the following state-of-the-art algorithms for creation of PPAs. Later, we refer to each algorithm by the number in parentheses. Interesting algorithms are provided with a short description. If no citation is provided for the description, then the description is from (Simon, 2005). In addition, DP stands for Davis-Putnam algorithm and DLL for Davis-Longeman-Loveland algorithm.

| | | |
|---|---|---|
| zres | (1) | ZRes is a propositional solver based on the original procedure of DP, as opposed to its modified version of DLL. On some highly structures SAT instances, proved hard for resolution, ZRes performs very well and surpasses all classical SAT provers by an order of magnitude (Chatalic & Simon, 2000). |

| | | |
|---|---|---|
| relsat-200 | (2) | Relevance-bounded SAT solver (Bayardo & Schrag, 1997). |
| eqsatz | (3) | It is equivalence reasoning enhanced satz to solve satisfiability problems involving equivalence clauses (Li, 2005). |
| nsat | (4) | |
| ntab | (5) | |
| ntab-back2 | (6) | |
| csat | (7) | |
| sato-3.2.1 | (8) | |
| dr | (9) | Directional Resolution. It is one of the rare DP implementation as stated in 1960. It was also the first attempt to revive DP. |
| posit | (10) | |
| modoc | (11) | |
| sato | (12) | It is a propositional solver based on DP method. It uses additional heuristics to speed up the solving (Zhang, 1997). |
| heerhugo | (13) | |
| satz-213 | (14) | |
| sat-grasp | (15) | |
| calcres | (16) | It is an implementation of the original procedure of DP as stated in 1960 (DP as opposed to the DLL well-known version of this procedure). |
| satz | (17) | |
| relsat | (18) | It is a randomized DLL algorithm. Though, we treat it as deterministic. |
| asat | (19) | It is a simple DLL implementation. Still, it is often very efficient. |
| ntab-back | (20) | |
| zchaff | (21) | It is a very fast extension of DLL, one of the fastest SAT solvers (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001). |
| modoc-2.0 | (22) | |
| satz-215 | (23) | |

Because, only CMA can handle a large number of algorithms, we also use subsets of all available algorithms to evaluate performance and feasibility of VOCMA and dynamic schedules. The notation for the sets of algorithms is

| | |
|---|---|
| $E$ | All algorithms |
| $E_1$ | 3, 19, 21 |
| $E_2$ | 7, 20, 23 |
| $E_3$ | 5, 6, 8 |

The set of instances in Sat-Ex is divided into 11 families. Each family contains similar instances. We denote the set of all available instances as $I$. Further, any set decorated by $\widetilde{I}$ is a reduced set containing only instances that can be solved by at least one algorithm. Moreover, we define the following instance sets

| | | | |
|---|---|---|---|
| $I$ | The set of all instances | $I_4$ | Last 800 instances |
| $I_1$ | First 400 instances | $I_5$ | First 800 instances |
| $I_2$ | Middle 400 instances | $I_6$ | Instances 1100 - 1303 |
| $I_3$ | Last 400 instances | $I_7$ | Instances 700 - 900 |

The instances were randomly resorted, when taken from Sat-Ex site. The instance set may be downloaded as specified in Appendix D.

## *9.2 Algorithms*

This section summarizes the statistics of the algorithms on the instance set. The actual numerical results are in Appendix C. Table C.1 summarizes the performance of individual algorithms on the instance sets and Graph C.1 shows the performance of all algorithms on $I$.

Because many of the algorithms use a quite different approach to solve the problem, their results vary significantly as well. The best performer and once the fastest SAT solver is zChaff. It outperforms other algorithms on all instance subsets. However, there are still instances that can be solved faster by other algorithms, and PPAs exploit this fact. We provide the specific results in the following sections.

As mentioned before, the variance[1] of an algorithm is also important (Huberman, Lukose, & Hogg, 1997; Zilberstein, 1996). A low variance increases the reliability for using the algorithm in a complex system, and also can increase the confidence of the users. Thus, we provide a statistics of standard deviation of algorithms in Table C.2. Later, we also explore whether and how PPA can decrease the variance. Also, other ideas regarding the variance are in Chapter 10.

## *9.3 Static Schedules: CMA*

The CMA algorithm can be used to determine a locally optimal schedule for all 23 algorithms, because of its linear complexity in terms of the number of algorithms. Since the starting classification is chosen randomly, each it run may output a different PPA.

We tested mean optimization CMA for both $I$ and $\widetilde{I}$. The simulated execution times of several PPAs for instances from $I$ are in Table C.3. Table C.4 provides the same data for $\widetilde{I}$. In both cases the optimal algorithm significantly outperforms the best algorithm for the instance set (zChaff). In the first case the mean execution time was decreased by 37%, and the standard deviation was decreased by 24%. In the second case, the mean execution time was decreased by 68% and the standard deviation was decreased by 55%. The impressive results on $\widetilde{I}$ were mainly caused by each instance being solved by at least one algorithm, thus reducing the very long runtime for those instances.

The small number of different results on different runs of CMA indicates that there may be only a few local optima in the problem. We explore further whether it is typical for CMA to reach the global optimum in Section 9.4.

The results of several runs of limit optimization CMA for runs on $\widetilde{I}$ are in Table C.5. The execution time for instances not solved within 10000 seconds were set to be 25000. The purpose was to identify whether all the instances would be solved with in a time longer than 20000, or they would not be at all. Moreover, since the PPAs are evaluated only theoretically, if the instance is not solved by any algorithm within the allocated 10000 seconds, it is assumed to be never solved. PPA with a static schedule solved each instance with the worst-case time of 16165.8, while no individual algorithm solved all instances[2]. The standard deviation 1280.8 was much lower than the variance of the algorithm with the minimal variance on this instance (zChaff). Still, the standard deviation was higher than the deviation of mean optimal static schedule PPA.

As well as in the case of mean optimization, the results on various runs of CMA are different. We also explore the tendency of CMA to reach the global optimum for limit optimization in Section 9.4.

[1] Or equivalently, standard deviation.

[2] The fact that we dropped all instances that were not solved by either algorithm turns this experiment to only a theoretical study. However, in many cases it is feasible to assume that each instance is solved by at least one algorithm.

## 9.4 Static Schedules : VOCMA

The purpose of this section is to compare the results obtained by CMA to those obtained by the optimal algorithm VOCMA. Unfortunately, though the complexity of VOCMA is polynomial in number of samples, calculation of more than 20 samples for 3 algorithms turned out to be very time consuming. The complexity be attributed also to the naive implementation of the algorithm. Therefore, we tested VOCMA only on the first 20 samples of indicated instance sets.

The results on mean optimization are summarized in Table C.6. The results on limit optimization are summarized in Table C.7. It is clear that in both cases, CMA obtained the optimal solution. For most instances in both cases, most runs of CMA converged to the globally optimal solution. It is not possible to verify the hypothesis for large sets of algorithms. However, it may be conjectured from the small number of different results of CMA on the large sets that the optimum was reached.

## 9.5 Dynamic Schedules

The complexity of dynamic schedules depends on the both the number of instances and the number of algorithms. Thus, for the tests we limited the number of algorithms to 3 and the largest instance set we used contained 200 instances.

The results of mean optimization dynamic schedules on an instance set for several different parameter values are in Table C.8 and Table C.9. It is clear from the first table that the schedule with the best profit was obtained for $z = 1$. This can be explained by very short execution times of the considered algorithms on this instance set. In addition, increasing $b$ beyond 10 did not increase the performance. On the other hand, for the second table, the best performance was achieved when using $z = 50$. The reason is that in this case, there were many instances unsolvable by some algorithms, and these have the execution time 20000. These results indicate that dynamic schedules are very sensitive to the choice of $z$ and $b$.

The experimental results of bound optimization dynamic schedules are in Table C.10. They indicate a slight increase in performance of the PPA compared to the best algorithm. However, the gain could be more significant when using more diverse algorithms or larger instance sets.

We also compared the performance of PPA with dynamic schedules to PPA with static schedules. The results are summarized in Table C.11. There was no clear winner, but in most cases the results were similar. The exception was the first row, where dynamic schedules almost twice over-performed the static ones. In the next cases, static schedules seem more advantageous. The difference was that parameters of the dynamic schedules were not tuned for experiments except the first two rows. That is, we tested only one choice of $z$ and $b$.

## 9.6 Generalization

This section explores the generalization capabilities of PPA with static and dynamic schedules. These results are only empirical, in contrast with the theoretical treatment in Chapter 8. However, they practically indicate the actual generalization; in this case the theoretical bounds are wide, and do not guarantee anything.

The generalization is tested by having a training and test sets. The optimal PPA is calculated from the training set. The algorithm can be assumed to generalize well if its performance on the training set is acceptable. We further denote the training set by $I'$ and test set by $I$. If not specified otherwise, the test set is the set of all available instance, therefore the same symbol.

For static schedules we used the suboptimal CMA algorithm to calculate the schedule. Generally, the considered PPA is the best one from 15 runs of CMA. The PPA were created from the training set. Schedules were by no means tuned for test set.

The results of generalization tests of static schedules for mean optimization are in Table C.12. Though, the fraction of samples used to find the optimization was usually smaller than a half, in most cases the learned PPA performed better on the whole instance set than the best single algorithm. This indicates that static schedules may very well generalize even from fairly small sample sets.

The results from of generalization tests of static schedules for limit optimization are in Table C.13. Generally, only less than 2% of the samples from the whole set were not solved by the worst case time on the training set. As a result, even a small number of instances may lead to a good PPA in case of limit optimization.

Though dynamic schedules seem to be slightly more powerful than their static counterparts, it is very interesting to compare how much a good performance on training set generalizes to a bigger problem distribution. The comparison is of course possible only in case of mean optimization, which is shared by both approaches.

For dynamic schedules, we studied only the generalization of mean optimization. The results are in Table C.14. The table also contains the comparison to static schedules on the same instance sets. Clearly, dynamic schedules generalize well the full distribution. They over-performed static schedules on most training sets. However, static schedules had better performance for most test sets. As a result, static schedules seem to have a better generalization capability.

## *9.7 Summary*

PPAs proved to be useful for the application on the presented SAT solvers on considered instances. The best results were obtained by VOCMA algorithms and for dynamic schedules. However, VOCMA is computationally very difficult and dynamic schedules do not seem generalize as well as static ones.

The performance of CMA for static schedules was in most cases comparable to both VOCMA and dynamic schedules. With its low computational complexity, simple algorithm, and high good generalization, CMA was superior to the other two approaches.

*Chapter 10*

# CONCLUSION AND FUTURE WORK

In this chapter we summarize the main results of the thesis and provide pointers to interesting extensions and applications.

## *10.1 Conclusion*

We proposed a method of enhancing the performance of deterministic algorithms by running more algorithms for the same problem in parallel, thus creating a PPA. The way in which the algorithms are combined is determined by a schedule. It specifies the resources allocated to each algorithm in every time instance. Given an set of typical instances randomly drawn from the instance distribution, we explored how to tune the schedule for the best performance.

The two main types of schedules we explored were static and dynamic schedules. We presented two primary ways of calculating static schedules, CMA and VOCMA. CMA does not guarantee optimality of the results, but its calculation is very fast. VOCMA, on the other hand, is optimal but it is practically unusable for finding PPA of more than 5 algorithms for more than 100 instances. Dynamic schedules can be modeled as a Markov Decision Process (MDP). It is useful because this formulation allows the use of standard algorithms for calculation of optimal schedules. We proposed a dynamic programming algorithm for this case and a possibly useful approximate algorithm.

For the measure of the best PPA, we proposed three methods: mean optimization, limit optimization, and bound optimization. Each of them is applicable and practical in different context. For both static and dynamic schedules, we showed that they need an asymptotically reasonable number of samples to achieve good generalization. The number of required samples is polynomial in the number of algorithms in the PPA, precision, and certainty.

All presented approaches were able to significantly outperform the best algorithm in the SAT problem domain. In most cases, they over-performed the best algorithm even when trained only on a subset of all instances. Together with the polynomial generalization bounds, this indicates that PPA may be practically applicable in other domains. However, the applicability on optimization problems, thus finding a PPA with best results in a fixed time, is limited by the assumptions placed on profit functions of individual algorithms.

From all presented schedule types and calculation methods, the most advantageous seems to be the simplest one, that is static schedules with the CMA algorithm. The advantage of VOCMA vanishes with a larger number of algorithms or samples. Dynamic schedules can obtain better results on the training set, but their generalization seems to be empirically worse. They require more tuning and cannot handle many algorithms. On the other hand, dynamic schedules can be used for bound optimization. Moreover, dynamic schedules do not require homogeneous profit functions. Profit functions are naturally homogeneous for decision algorithms, but this may not be the case for optimization algorithms.

The presented result can also be applied to the popular field of multi-agent systems. In this scenario, we could have a community of agents, each trying to solve the same problem instance.

## 10.2 Future Work

The presented work on PPA is by no means complete. The next list summarizes the interesting extensions and applications. Each paragraph summarizes advancement a specific direction of PPA.

*SCHEDULES*

*Minimization of variance:* The idea is to not maximize minimal profit, but instead to minimize the variance of profits, given a desired mean profit. This is proposed in (Huberman et al., 1997).

*Combination of importance sampling algorithms:* The PPA approach may lower the error of importance sampling. Some research in this area has been done, for example (Owen & Zhou, 2000).

*Stochastic algorithms:* Extending the approach to cover also stochastic algorithms may be useful. One possibility is to use probabilistic results as a profit of an algorithm. This would reduce the algorithms to deterministic. Other approaches may be useful too.

*Randomized schedules:* Exploring probabilistic schedules is interesting too. They may be used to lower variance, as in research on optimal portfolio selection, Example 3.1.5 in (Bertsekas, 2003). Also, it may be possible to use the game theory analysis to create best schedules for the worst case distribution over the instances, as in mixed strategies (Osborne & Rubinstein, 1994).

*Branch and Bound:* PPA framework could be applied to branch and bound algorithms. In this case, the performance may be increased by communication of the current best bound between the algorithms. Moreover, a different schedule may be optimal for each bound.

*SOLUTION METHODS*

*Extension to non homogeneous algorithms:* For optimization algorithms, it may be useful to generalize CMA to handle also non-homogeneous algorithms.

*Hybrid CMA and VOCMA:* VOCMA algorithm that works by iteratively determining optimal split points for each pair of algorithms.

*VOCMA:* The VOCMA algorithm could be possibly enhanced by early checking whether a split point leads to a valid classification. Given algorithms $a$, $b$, and $c$, the split points between $a$-$b$ and $a$-$c$ determine the split point between $b$-$c$. Extending this idea to more algorithms may dramatically reduce the computation time of VOCMA.

*Domination:* CMA usually picks only a small number of algorithms to have non-zero resources. Determining the algorithms that must have zero resources in the optimal schedule may reduce the solution time, and improve the results.

*GENERALIZATION*

*Regression:* The schedule learning process is related to a non-parametric regression (Friedman et al., 2000). Therefore, regression methods may be used for finding schedules.

*General mean optimization bound:* The generalization result presented for mean optimization static schedules may be extended to arbitrary algorithms. One possibility is to form a forest induced by the domination relation of algorithms.

*Variance:* The variance on the training set may have a prediction capability for the generalization. Maybe it is possible to obtain tighter online bounds for learning using the training set variance.

*EVALUATION*

*Optimization problems:* The practical applicability of the PPA on optimization algorithms needs to be evaluated. There is the question whether the homogeneity assumptions is not too strict.

*Comparison to ALS:* It would be interesting to compare the PPA approach to the Adaptive Levin Search, as presented in (Schmidhuber & Wiering, 1996).

*Generalization from simple instances:* Learning on difficult instances takes a long time. This problem could be addressed by learning on easier instances of a similar type. However, it must be determined how to generate the simples instances, and determine the quality of the schedule.

*Comparison to heuristic combination:* The approach could be compared to combining two heuristics for a single problem. The standard approaches either represent the compound heuristic function as a linear combination of the individual ones, or take maximal values. Weights for the linear combination may be obtained using reinforcement learning, but in general it invalidates the admissibility of heuristics. On the other hand, taking the maximal value is often ineffective. Treating heuristics as different algorithms and combining them using PPA may lead to interesting results.

*OTHER EXTENSIONS*

*Parametrized complexity:* The PPA approach may yield interesting results for parametrized complexity algorithms (Downey & Fellows, 1998).

# *Appendix A*

## NOTATION

| **Symbol** | **Meaning** |
|---|---|
| $\mathcal{P}$ | Parallel Portfolio of Algorithms. |
| $\mathcal{B}$ | Operator that determines the optimality of a schedule. |
| $S$ | Schedule of a PPA. |
| $E$ | Set of individual algorithms for portfolio creation. |
| $I$ | Training set of instances. |
| $\mathcal{I}$ | All possible instances for the problem. |
| $\Phi$ | Set of all available algorithms for speedup learning. |
| $P$ | Profit function of a PPA, usually determined by a schedule and an instance. |
| $p$ | Profit function for individual algorithms. |
| $\mu$ | Profit function dependency on instance, in homogeneous algorithms. |
| $\nu$ | Profit function dependency on resource amount, in homogeneous algorithms. |
| $r$ | Resource allocation function, these comprise the schedule. |
| $e$ | Individual algorithm. |
| $x$ | Single instance, usually from $I$. |
| $X$ | Random variable that represents a single instance, generally from $\mathcal{I}$. |
| $\eta$ | Execution time of a decision algorithm. |
| $\tau$ | Maximal time for a dynamic schedule, only one algorithm is executed after this time. |
| $z$ | Length of each interval in dynamic schedules. |
| $b$ | Number of the intervals in dynamic schedules, including the last, infinitely long one. |
| $Z$ | Probabilistic event that the algorithm solves the instance during the specified time interval. |
| $\Theta$ | Time interval in dynamic schedules, its length is specified by $z$. |
| $B$ | Bound optimization bound value. |
| $m$ | Number of instances in the training set. |
| $i$ | Iterator used to iterate over the instances. |
| $n$ | Number of algorithms in the portfolio. |
| $j$ | Iterator used to iterate over the algorithms. |

# *Appendix B*

# MATHEMATICAL BACKGROUND

In this appendix we summarize mathematical concepts we use in the thesis. It contains mostly background on mathematical optimization, optimality conditions and convex analysis. In addition, it contains the probabilistic and static concepts used in pattern recognition.

## *B.1 Mathematical Optimization Basics*

This section present a small part of the non-linear and linear programming theory. The theorems and definitions are presented according to (Bertsekas, 2003).

**Proposition B.1.1.** *If $x^*$ is a local minimum of $f$ over $X$, then*

$$\nabla f(x^*)'(x - x^*) \geq 0, \quad x \in X. \tag{B.1.1}$$

*Moreover, if $f$ is convex over $X$, then the condition B.1.1 is also sufficient for $x^*$ to minimize $f$ over $X$.*

**Proposition B.1.2.** *Every function $f$ satisfies $\max_a \max_b f(a,b) = \max_{a,b} f(a,b)$.*

**Proposition B.1.3.** *Let $C_1 \subseteq C_2$. Then*

$$\max_{x \in C_1} f(x) \leq \max_{x \in C_2} f(x).$$

**Theorem B.1.4** (Lagrange Multiplier Theorem - Necessary Conditions)**.** *Let $x^*$ be a local minimum of $f$ subject to $h(x) = 0$, and assume that the constraint gradients $\nabla h_1(x^*), \ldots, \nabla h_m(x^*)$ are linearly independent. Then there exists a unique vector $\lambda^* = (\lambda_1^*, \ldots, \lambda_m^*)$, called a Lagrange multiplier vector, such that*

$$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla h_i(x^*) = 0.$$

*If in addition $f$ and $h$ are twice continuously differentiable, we have*

$$y' \left( \nabla^2 f(x^*) + \sum_{i=1}^{m} \lambda_i \nabla^2 h_i(x^*) \right) y \geq 0 \quad \textit{for all } y \in V(x^*),$$

*where $V(x^*)$ is the subspace of first order feasible variations*

$$V(x^*) = \left\{ y \,\middle|\, \nabla h_i(x^*) y' = 0, i = 1, \ldots, m \right\}.$$

**Definition B.1.5.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function, and $h_i(x) = 0 i = 1, \ldots, m$ the constraints. *Lagrangian function* $L : \mathbb{R}^{n+m} \to \mathbb{R}$ is defined as

$$L(x, \lambda) = f(x) + \sum_{i=1}^{m} \lambda_i h_i(x).$$

The necessary optimality conditions for $x^*$ in terms of the Lagrangian function are:

$$\nabla_x L(x^*, \lambda^*) = 0, \quad \nabla_\lambda L(x^*, \lambda^*) = 0.$$

**Theorem B.1.6** (Second Order Sufficiency Conditions)**.** *Assume that $f$ and $h$ are twice continuously differentiable, and let $x^* \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}^m$ satisfy*

$$\nabla_x L(x^*, \lambda^*) = 0, \quad \nabla_\lambda L(x^*, \lambda^*) = 0,$$

$$y' \nabla^2_{xx} L(x^*, \lambda^*) y > 0, \quad \textit{for all } y \neq 0 \textit{ with } \nabla h(x^*)' y = 0.$$

*Then $x^*$ is a strict local minimum of $f$ subject to $h(x) = 0$. In fact, there exist scalars $\gamma > 0$ and $\epsilon > 0$ such that*

$$f(x) \geq f(x^*) + \frac{\gamma}{2} \|x - x^*\|^2, \quad \forall x \textit{ with } h(x) = 0 \textit{ and } \|x - x^*\| < \epsilon.$$

**Proposition B.1.7.** *A convex continuous function on a compact convex set has only one local minimum, which is also the global minimum.*

## B.2 Statistics

This section introduces fundamental theorems from probability theory and statistics, used in the work. The theorems and definitions are presented according to (Devroye et al., 1996).

**Theorem B.2.1.** *Let $X$ be a random variable. Then*

$$\mathbf{E}[X] = \mathbf{E}[X \mid y] \mathbf{P}[y] + \mathbf{E}[X \mid \neg y] \mathbf{P}[\neg y].$$

**Theorem B.2.2.** *Let $A$ be a random event, then*

$$\mathbf{P}[A] = \mathbf{E}[\mathrm{I}\{A\}].$$

**Theorem B.2.3** (Chernoff's Inequality)**.** *Let $X_i$ be random independent variables. Let their sum be*

$$S_n = \sum_{i=1}^{n} X_i \quad X_i \in \langle a_i, b_i \rangle.$$

*Then, for any $\epsilon > 0$ we have*

$$\begin{aligned} \mathbf{P}[S_n - \mathbf{E}[S_n] \geq \epsilon\sigma] &\leq \exp\left(-2\epsilon^2\right) \\ \mathbf{P}[S_n - \mathbf{E}[S_n] \leq -\epsilon\sigma] &\leq \exp\left(-2\epsilon^2\right). \end{aligned}$$

**Theorem B.2.4** (Hoeffding's Inequality)**.** *Let $X_1, \ldots, X_n$ be independent bounded random variables such that $X_i$ is from interval $\langle a, b \rangle$. Let their sum be $S_n = \sum_{i=1}^{n} X_i$. Then for any $\epsilon > 0$ we have*

$$\begin{aligned} \mathbf{P}[S_n - \mathbf{E}[S_n] \geq \epsilon] &\leq \exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^{n}(b_i - a_i)^2)}\right) \\ \mathbf{P}[S_n - \mathbf{E}[S_n] \leq -\epsilon] &\leq \exp\left(\frac{-2\epsilon^2}{\sum_{i=1}^{n}(b_i - a_i)^2)}\right). \end{aligned}$$

**Theorem B.2.5** (Chebyshev's Inequality)**.** *Let $X$ be a random variable. Then for each $t$,*

$$\mathbf{P}[|X - \mathbf{E}[X]| \geq t] \leq \frac{\mathbf{Var}[X]}{t^2}.$$

**Theorem B.2.6** (Boole's Inequality)**.** *For any finite or countable set of events $A_1, A_2, \ldots$ we have*

$$\mathbf{P}\left[\bigcup_i A_i\right] \leq \sum_i \mathbf{P}\left[A_i\right].$$

**Theorem B.2.7** (GlivenkoCantelli)**.** *Let $Z_1, \ldots, Z_n$ be i.i.d.*[1] *real valued variables with distribution function $F(z) = \mathbf{P}\left[Z_1 \leq z\right]$. Denote the standard empirical function by*

$$F_n(z) = \frac{1}{n}\sum_{i=1}^{n} \mathrm{I}\left\{Z_i \leq z\right\}.$$

*Then*

$$\mathbf{P}\left[\sup_{z\in\mathbb{R}} |F(z) - F_n(z)| \geq \epsilon\right] \leq 8(n+1)\exp\left(\frac{-n\epsilon}{32}\right).$$

**Definition B.2.8.** Let $\mathcal{A}$ be a collection of measurable sets. For $(z_1, \ldots, z_n) \in \{\mathbb{R}^d\}^n$, let $\mathcal{N}_{\mathcal{A}}$ be the number of different sets in

$$\{\{z_1, \ldots, z_n\} \cap A \mid A \in \mathcal{A}\}.$$

The $n$-th shatter coefficient is

$$s(\mathcal{A}, n) = \max_{(z_1,\ldots,z_n)\in\{\mathbb{R}^d\}^n} \mathcal{N}_{\mathcal{A}}(z_1, \ldots, z_n).$$

That is, the $n$-th shatter coefficient is the maximal number of different subsets of $n$ points that can be picked out by the class of sets $\mathcal{A}$.

We use notation $\nu(A) = \mathbf{P}\left[Z_1 \in A\right]$ and $\nu_n(A) = \frac{1}{n}\sum_{j=1}^{n} \mathrm{I}\left\{Z_j \in A\right\}$.

**Theorem B.2.9** (Vapnik and Chervonenkis)**.** *For any probability measure $\nu$ and class of sets $\mathcal{A}$, and for any $n$ and $\epsilon > 0$,*

$$\mathbf{P}\left[\sup_{A\in\mathcal{A}} |\nu_n(A) - \nu(A)| > \epsilon\right] \leq 8s(\mathcal{A}, n)\exp\left(\frac{-n\epsilon^2}{32}\right).$$

[1] Independent and identically distributed.

*Appendix C*

# APPLICATION RESULTS

In this chapter we present numerical results from experiments on SAT problem solving. The actual experiments are described and commented in Chapter 9.

| $E$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $I$ | 13176.0 | 715.0 | 1103.6 | 3731.7 | 4940.3 | 1704.9 | 3270.1 | 1319.7 |
| $I_1$ | 13040.6 | 820.6 | 1078.8 | 3899.3 | 4895.4 | 1985.4 | 3379.4 | 1299.4 |
| $I_2$ | 13050.3 | 559.5 | 1033.1 | 3484.7 | 5192.8 | 1545.6 | 2999.9 | 1261.1 |
| $I_3$ | 13398.0 | 967.1 | 1509.0 | 3818.7 | 4868.4 | 1847.3 | 3573.1 | 1650.1 |
| $I_4$ | 13183.7 | 653.1 | 1119.5 | 3647.3 | 4755.6 | 1550.3 | 3080.9 | 1269.1 |
| $I_5$ | 13036.8 | 665.7 | 1032.2 | 3671.4 | 5025.6 | 1742.5 | 3168.4 | 1256.8 |
| $I_6$ | 13911.1 | 374.9 | 568.5 | 3661.0 | 4979.9 | 1515.3 | 3485.4 | 901.0 |
| $I_7$ | 12866.8 | 808.8 | 1208.3 | 3606.6 | 4901.6 | 1546.7 | 2968.8 | 1281.1 |
| $\widetilde{I}$ | 13133.9 | 595.9 | 986.8 | 3631.2 | 4847.3 | 1591.9 | 3166.8 | 1204.3 |
| $E$ | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| $I$ | 18424.5 | 3885.8 | 1396.2 | 994.4 | 4503.0 | 1184.0 | 2674.3 | 13938.4 |
| $I_1$ | 18006.5 | 3932.3 | 1430.9 | 905.5 | 4671.2 | 1025.7 | 2763.2 | 13936.0 |
| $I_2$ | 18409.3 | 3456.8 | 1073.4 | 935.3 | 3973.8 | 1088.8 | 2272.7 | 13887.1 |
| $I_3$ | 18708.4 | 4369.7 | 1823.2 | 1412.0 | 5060.1 | 1639.2 | 3052.4 | 14085.1 |
| $I_4$ | 18681.5 | 3812.1 | 1426.3 | 1002.8 | 4426.6 | 1253.6 | 2651.5 | 13875.6 |
| $I_5$ | 18205.9 | 3673.9 | 1228.5 | 896.6 | 4302.5 | 1033.6 | 2495.8 | 13903.9 |
| $I_6$ | 18917.9 | 3912.2 | 1335.1 | 600.7 | 3839.9 | 1117.2 | 2430.1 | 14665.8 |
| $I_7$ | 19105.7 | 3509.5 | 1263.6 | 1022.2 | 4446.6 | 1082.2 | 2182.2 | 13303.1 |
| $\widetilde{I}$ | 18414.7 | 3786.2 | 1281.3 | 877.0 | 4407.2 | 1067.7 | 2567.3 | 13900.9 |
| $E$ | 17 | 18 | 19 | 20 | 21 | 22 | 23 | |
| $I$ | 1336.0 | 951.0 | 5722.8 | 1841.4 | 372.3 | 1408.7 | 1100.0 | |
| $I_1$ | 1050.9 | 1097.9 | 5682.7 | 2024.5 | 371.9 | 1531.5 | 960.8 | |
| $I_2$ | 1222.3 | 665.0 | 5462.1 | 1740.3 | 413.0 | 1098.0 | 934.3 | |
| $I_3$ | 1941.6 | 1228.9 | 6138.2 | 2061.4 | 470.0 | 1723.6 | 1583.2 | |
| $I_4$ | 1492.2 | 898.5 | 5636.8 | 1706.3 | 343.4 | 1396.6 | 1151.6 | |
| $I_5$ | 1113.1 | 857.3 | 5554.3 | 1859.5 | 368.0 | 1291.1 | 923.7 | |
| $I_6$ | 1276.4 | 800.9 | 6268.2 | 1527.4 | 118.5 | 1253.4 | 1019.4 | |
| $I_7$ | 1524.5 | 811.3 | 5352.8 | 1857.7 | 513.7 | 1080.8 | 1070.4 | |
| $\widetilde{I}$ | 1220.7 | 833.4 | 5634.6 | 1729.2 | 251.1 | 1293.9 | 983.3 | |

***Table C.1:*** *Mean execution time of individual algorithms on various instance sets.*

| $E$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\sigma(I)$ | 9192.0 | 3599.2 | 4404.6 | 7629.7 | 8565.2 | 5512.9 | 7272.3 | 4864.5 |
| $\sigma(\widetilde{I})$ | 9204.6 | 3274.3 | 4159.2 | 7545.0 | 8509.2 | 5338.3 | 7174.4 | 4651.8 |
| $E$ | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| $\sigma(I)$ | 5380.1 | 7810.9 | 4972.8 | 4103.7 | 8194.2 | 4563.1 | 6655.0 | 9118.3 |
| $\sigma(\widetilde{I})$ | 5395.3 | 7731.2 | 4767.5 | 3833.7 | 8128.1 | 4329.9 | 6534.2 | 9134.0 |
| $E$ | 17 | 18 | 19 | 20 | 21 | 22 | 23 | |
| $\sigma(I)$ | 4890.6 | 4198.5 | 8976.6 | 5716.9 | 2633.3 | 4999.5 | 4410.4 | |
| $\sigma(\widetilde{I})$ | 4679.6 | 3934.4 | 8933.6 | 5552.8 | 2140.2 | 4795.8 | 4165.3 | |

***Table C.2:*** *Standard deviation in execution time of individual algorithms on the training sets.*

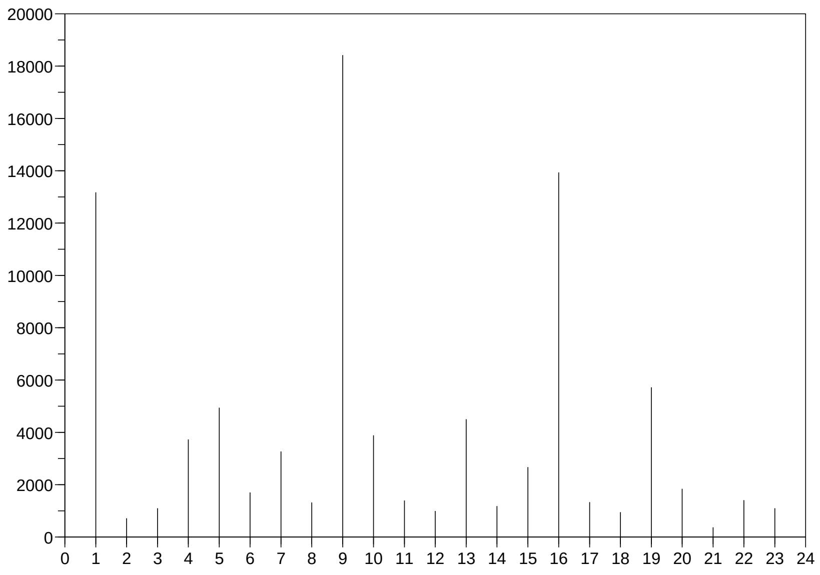

***Figure C.1:*** *Mean execution time of individual algorithms on $I$.*

| Run | $-\mathbf{E}[P]$ | $\sigma(P)$ |
|---:|---:|---:|
| 1 | 331.5 | 2656.3 |
| 2 | 294.8 | 2462.7 |
| 3 | 255.2 | 2226.2 |
| 4 | 318.7 | 2583.6 |
| 5 | 314.6 | 2580.5 |
| 6 | 321.2 | 2591.3 |
| 7 | 343.7 | 2704.2 |
| 8 | 255.2 | 2226.2 |
| 9 | 307.6 | 2520.6 |
| 10 | 308.4 | 2504.6 |
| 1-200 | **233.2** | 2005.6 |
| Simple | 3365.2 | 36787.2 |

***Table C.3:*** *Mean execution time and standard deviation of 10 calculations of CMA on $I$ and the best result from 200 runs.* Simple *denotes a naive PPA, in which all algorithms get an equal share.*

| Run | $-\mathbf{E}[P]$ | $\sigma(P)$ |
|---|---|---|
| 1 | 126.6 | 1849.2 |
| 2 | 78.8 | 972.2 |
| 3 | 194.5 | 1866.4 |
| 4 | 78.8 | 972.2 |
| 5 | 135.7 | 1766.7 |
| 6 | 126.6 | 1849.2 |
| 7 | 78.8 | 972.2 |
| 8 | 78.8 | 972.2 |
| 9 | 78.8 | 972.2 |
| 10 | 78.8 | 972.2 |
| 1-200 | **77.1** | 1000.5 |
| Simple | 544.3 | 8036.5 |

***Table C.4:*** *Mean execution time and standard deviation of 10 calculations of CMA on $\widetilde{I}$ and the best result from 200 runs.* Simple *denotes a naive PPA, in which all algorithms get an equal share.*

| Run | $-\min P$ | $\sigma(P)$ |
|---|---|---|
| 1 | **16165.8** | 1380.5 |
| 2 | **16165.8** | 1380.5 |
| 3 | 26096.3 | 2187.9 |
| 4 | 24929.4 | 2104.2 |
| 5 | **16165.8** | 1380.5 |
| 6 | **16165.8** | 1380.5 |
| 7 | **16165.8** | 1380.5 |
| 8 | **16165.8** | 1380.5 |
| 9 | **16165.8** | 1380.5 |
| 10 | 257515.9 | 37054.4 |
| 1-200 | **16165.8** | 1380.5 |
| Simple | 224907.8 | 8036.5 |

***Table C.5:*** *The worst case execution time and standard deviation of PPAs calculated by CMA $\widetilde{I}$. Simple denotes a naive execution where all algorithms in the portfolio have the same resource share. Instances not solved by an algorithm in 10000 seconds are assumed to be never solved. Moreover, the penalty for not solving an instance was changed to 25000 seconds.*

| Instances | Algorithms | $-P^*$ | $-P^{CMA}$ | C | $-P^B$ |
|---|---|---|---|---|---|
| $I_6$ | $E_1$ | 2.13 | 2.13 | 10 | 15.59 |
| $I_6$ | $E_2$ | 2.14 | 2.14 | 3 | 199.05 |
| $I_6$ | $E_3$ | 1.92 | 1.92 | 7 | 8.48 |
| $I_7$ | $E_1$ | 0.52 | 0.52 | 6 | 1.29 |
| $I_7$ | $E_2$ | 85.90 | 85.90 | 8 | 85.94 |
| $I_7$ | $E_3$ | 1.18 | 1.18 | 2 | 1.68 |

***Table C.6:*** *Optimality of using CMA for mean optimization. $P^*$ is the profit of the mean optimal PPA, calculated by VOCMA. $P^{CMA}$ is the profit of the best PPA out of 10 runs of CMA. C is the number of runs of CMA that resulted in its optimal profit. $P^B$ is the best profit of an individual algorithm from the specified algorithm set.*

| Instances | Algorithms | $-P^*$ | $-P^{CMA}$ | C | $-P^B$ |
|---|---|---|---|---|---|
| $I_6$ | $E_1$ | 12.09 | 12.09 | 10 | 162.97 |
| $I_6$ | $E_2$ | 10.93 | 10.93 | 7 | 3939.34 |
| $I_6$ | $E_3$ | 19.63 | 19.63 | 8 | 102.40 |
| $I_7$ | $E_1$ | 4.03 | 4.03 | 7 | 9.78 |
| $I_7$ | $E_2$ | 1710.24 | 1710.24 | 9 | 1710.24 |
| $I_7$ | $E_3$ | 15.74 | 15.74 | 4 | 22.54 |

***Table C.7:*** *Optimality of using CMA for limit optimization. $P^*$ is the profit of the limit optimal PPA, calculated by VOCMA. $P^{CMA}$ is the profit of the best PPA out of 10 starts of CMA. $C$ is the number of starts of CMA that resulted in its optimal profit. $P^B$ is the best profit of a single algorithm from the specified algorithm set. All profits are with regard to specified instance set.*

| $z$ | $b$ | $\mathbf{E}[P(S)]$ | $\sigma(P(S))$ |
|---|---|---|---|
| 1 | 10 | **1.39** | 3.56 |
| 1 | 15 | **1.39** | 3.56 |
| 2 | 10 | 1.64 | 4.19 |
| 2 | 15 | 1.64 | 4.19 |
| 5 | 10 | 2.19 | 5.23 |
| 5 | 15 | 2.19 | 5.23 |
| 10 | 10 | 2.86 | 6.79 |
| 20 | 10 | 3.36 | 8.58 |
| $e$ | | 15.59 | 46.15 |

***Table C.8:*** *Results of calculating mean optimal dynamic schedules for PPA for the first 20 instances of $I_6$ and the algorithm set $E_1$. $e$ is an algorithm from $E$ with the best empirical performance.*

| $z$ | $b$ | $\mathbf{E}[P(S)]$ | $\sigma(P(S))$ |
|---|---|---|---|
| 1 | 10 | 211.3 | 2000.2 |
| 10 | 10 | 204.7 | 2001.7 |
| 20 | 10 | 205.4 | 2003.6 |
| 50 | 15 | **19.5** | 122.3 |
| 100 | 10 | 23.0 | 124.6 |
| 100 | 15 | 23.0 | 124.6 |
| 200 | 10 | 27.1 | 138.8 |
| $e$ | | 217.8 | 1999.8 |

***Table C.9:*** *Results of calculating mean optimal dynamic schedules for PPA for the first 100 instances of $I_6$ and the algorithm set $E_1$. $e$ is an algorithm from $E$ with the best empirical performance.*

| $I$ | $z$ | $b$ | $B = 4$ | $B = 10$ | $B = 16$ |
|---|---|---|---|---|---|
| $I_6(1:20)$ | 1 | 10 | 0.9 | 0.95 | 1 |
| $I_6(1:20)$ | 2 | 10 | 0.85 | 0.9 | 1 |
| $I_6(1:20)$ | $e_1$ | | 0.85 | 0.85 | 0.9 |
| $I_6(1:100)$ | 1 | 10 | 0.87 | 0.9 | 0.93 |
| $I_6(1:100)$ | $e_2$ | | 0.85 | 0.88 | 0.92 |

***Table C.10:*** *The results of bound optimal dynamic schedules using algorithms from $E_1$. Column $B$ is the fraction of instances solved before the bound. $e_1$ and $e_2$ are algorithms from $E$ with the best empirical performances.*

| $E$ | $I$ | $\mathbf{E}[P(S)]$ | $\mathbf{E}[P(S')]$ | $\sigma(P(S))$ | $\sigma(P(S'))$ | $\mathbf{E}[P(e)]$ | $\sigma(P(e))$ |
|---|---|---|---|---|---|---|---|
| $E_1$ | $I_6(1:20)$ | 1.39 | 2.13 | 3.56 | 4.26 | 15.59 | 46.15 |
| $E_1$ | $I_6(1:100)$ | 19.54 | 24 | 122.3 | 2265.8 | 217.8 | 1999.8 |
| $E_1$ | $I_6$ | 44.85 | 27.13 | 332.39 | 166.6 | 118.5 | 1404.7 |
| $E_2$ | $I_6$ | 717.4 | 715.79 | 3462.3 | 3459.7 | 1019.4 | 4164.87 |
| $E_2$ | $I_7$ | 1070.44 | 1070.17 | 4381.5 | 4384.4 | 1070.44 | 4381.51 |
| $E_3$ | $I_7$ | 1241.87 | 1235.18 | 4771.1 | 4852.8 | 1281.09 | 4760.9 |

***Table C.11:*** *Comparison of PPA performance with dynamic schedules versus static schedules. $S$ denotes the mean optimal dynamic schedule for training instance. $S'$ denotes the best mean static schedule from a 10 runs of CMA. Dynamic schedules were created with parameters $z = 50$ and $b = 15$, except the first row where $z = 1$. $e$ represents an algorithm with best empirical performance.*

| $I'$ | $-\mathbf{E}[P(S) \mid I']$ | $-\mathbf{E}[P \mid I]$ |
|---|---|---|
| $I_1$ | 399.8 | 387.4 |
| $I_2$ | 225.1 | 418.8 |
| $I_3$ | 380.1 | 311.7 |
| $I_4$ | 228.5 | 297.3 |
| $I_5$ | 225.2 | 259.7 |
| $I_6$ | 26.4 | 514.1 |
| $I_7$ | 393.1 | 311.5 |

***Table C.12:*** *Generalization of mean optimization static schedules. The schedules were obtained by CMA for $I'$ and simulated on $I$, which is the set of all instances.*

| $I'$ | $-\mathbf{E}[P \mid I']$ | $\mathbf{P}\left[P(S) < B \,\middle|\, \widetilde{I}\right]$ |
|---|---|---|
| $I_1$ | 7726.4 | 0.005 |
| $I_2$ | 5174.6 | 0.013 |
| $I_3$ | 10569.1 | 0.004 |
| $I_4$ | 10573.2 | 0.002 |
| $I_5$ | 11230.0 | 0.003 |
| $I_6$ | 1485.8 | 0.012 |
| $I_7$ | 9829.7 | 0.015 |

***Table C.13:*** *Generalization of limit optimization static schedules. The schedules were obtained by CMA algorithm for $I'$ and simulated on $\widehat{I}$.*

| $E$ | $I'$ | $I$ | $\mathbf{E}[P(S) \mid I']$ | $\mathbf{E}[P(S') \mid I']$ | $\mathbf{E}[P(S) \mid I]$ | $\mathbf{E}[P(S') \mid I]$ |
|---|---|---|---|---|---|---|
| $E_1$ | $I_6(1:20)$ | $I_6$ | 1.39 | 2.13 | 114.77 | 51.28 |
| $E_1$ | $I_6(1:100)$ | $I_6$ | 19.54 | 24.02 | 44.85 | 30.92 |
| $E_2$ | $I_7(1:20)$ | $I_7$ | 85.94 | 85.94 | 1070.44 | 1070.44 |
| $E_3$ | $I_7(1:100)$ | $I_6$ | 720.91 | 680.59 | 1281.1 | 1291.62 |

***Table C.14:*** *The generalization results comparison for mean optimization dynamic and static schedules. $S$ denotes the mean optimal dynamic schedule. $S'$ denotes the best schedule obtained from 10 runs of CMA. The schedules were calculated from $I'$ and simulated on $I$.*

*Appendix D*

# IMPLEMENTATION

This chapter contains the code of algorithms that calculate both static and dynamic schedules. All algorithms were implemented in Scilab 3.0. Scilab is a free algebraic and mathematical programming tool that offers similar capabilities as Matlab. It can be downloaded from http://www.scilab.org. The programs are optimized for implementation simplicity, not for the performance of the calculations. Thus, the calculation of the optimal static schedules and dynamic schedules could be significantly enhanced. A Scilab ready matrix containing the algorithms can be downloaded from http://marekpetrik.webpark.sk/performancematrix.

## *D.1 Static Schedules*

```
//X contains profits, which are equivalent to negative solution times
//The following code loads the test instances
//Rows in the matrix represent instances and columns algorithms
//X = fscanfMat("<path>");
//X = modifyEndValue(X, -20000);
//X1 = X(1:400,:);
//X2 = X(400:800, :);
//X3 = X(800:1200, :);
//X4 = X(503:1303, :);
//X5 = X(1:800, :);
//X6 = X(1100:1302, :);
//X7 = X(700:900, :);
//Xx = eliminateUnsolvable(X);

//X6A1 = X6(:,[3,19,21]);
//X6A2 = X6(:,[7,20,23]);
//X6A3 = X6(:,[5,6,8]);
//X7A1 = X7(:,[3,19,21]);
//X7A2 = X7(:,[7,20,23]);
//X7A3 = X7(:,[5,6,8]);

//Generally, the semantics of the symbols is following:
//X - performance of algorithms; instances x algorithms
//W- assignment of solutions to algorithms, has the same dimensions as X
//c - schedule - vector of r values

//Simulates the run of the parallel portfolio with a static schedule
function z = simulate(X,c)
        ieee(2);
       W = calculateW(X,c);
        r = ones(size(c,1),1)./c;
        r = naneliminate(r);
        //This will not work for positive profits and min, because then
```

```
    //zero will be minimal value.
    z = X .* W * r;
endfunction

//Calculates the empirical mean profit
//x - list of performances
function r = mySum(x)
    y = thrownan(x);
    r = sum(y) ./ size(y,1);
endfunction

//calculates the optimal W, given instances and schedule
function W = calculateW(X,c)
    R = zeros(X);
    W = R;
    ieee(2);
    for i=1:size(X,1), R(i,:) = X(i,:) ./ c'; end;
    for i=1:size(X,1), [m,n] = max(R(i,:)); W(i,n) = 1; end
endfunction

//Calculates the optimal schedule for a fixed classification
//t - type of the optimization, the same meaning as in calculateCMA
function c = calculateC(X,W,t)
    A = X.*W

    if t == 1 then
        a = sum(A, 'r');
        b = sum(W, 'r');
        ieee(2);        //eliminate errors from multiplication by zero
        a = naneliminate(a ./ b);
        b = normalize(b);
        //use abs, because: sqrt(-0) = -0,
        //and if a = -0 then 1/a==-inf, and that is wrong
        //it is caused by a scilab bug
        c = abs(sqrt(-b .* a))';
    elseif t == 2 then
        a = min(A, 'r');
        c = a';
    end

    c = normalize(c);
endfunction

//Calculate the optimal c, W using the
//t - type of optimization,
//t = 1 means "mean optimization", t = 2 means "limit optimization".
function [c, W] = calculateCMA(X,t)
    c = normalize(rand(size(X,2),1));

    for i=1:20 do
        W = calculateW(X,c);
        cOld = c;
        c = calculateC(X,W,t);
        n = norm(c - cOld);
        //iterate only until the last element is reached
        if n == 0 then break; end;
    end
endfunction

//Column performance plot
function plotPerformance(p)
    p1 = tabul(p);
    p1(:,2) = cumsum(p1(:,2));
    p1(:,2) = p1(:,2) ./ size(p,1);
    plot2d(-p1(:,1), p1(:,2));
endfunction

//**** VOCMA algorithm


//VOCMA algorithm
//pc - the schedule
//p - the profit of the schedule
//t - the same as in calculateCMA
function [pc, p] = solveOptimal(X,t)
    m = size(X,1);
    n = size(X,2);
    k = size(R,2);
    R = buildStructure(X);
    p = -10000;
    pc = 0;

    o = zeros(k, 1);
    d = 0; vm = 0;
    while sum(abs(o - ones(k,1)*m)) > 0 do
        d = d+1;
        if modulo(d, 1000) == 0 then
            disp([d, vm, p, pc']);
        end;
        W = classifyVector(o, R, m, n);
        [W,v] = createW(W);
        if v then
            vm = vm + 1;
            c = calculateC(X,W,t);
            if t == 1 then
                e = mySum(simulate(X,c));
            else
                e = min(simulate(X,c));
            end;
            if e > p then
                p = e;
                pc = c;
            end
            p = max(p, e);
        end
        o = createNextVector(o, m);
    end
endfunction

//Builds structure required to translate split points into
//a classification
function R = buildStructure(X)
```

```
    n = size(X, 2);
    R = zeros(size(X,1), n * (n-1) /2);
    z = 0;
    for j=1:n do
        for k = (j+1):n do
            z = z + 1;
            a = X(:,j) ./ X(:,k);
            [b,c] = gsort(a, 'g', 'i');
            R(:,z) = c;
        end;
    end;
endfunction

//Creates a schedules for a set of split points
//the split point is defined in terms of the first algorithm
//h - the vector to classify
//R - the result of build structure - mapping to instances
//m - number of instances
//n - number of algorithms
function W = classifyVector(h, R, m, n)
    s = size(h,1);
    W = zeros(m, n);
    z = 0;
    for j=1:n do
        for k = (j+1):n do
            z = z + 1;
            W(R(1:h(z),z),j) = W(R(1:h(z),z),j) + 1/(n-1);
            W(R(h(z)+1:$,z),k) = W(R(h(z)+1:$,z),k) + 1/(n-1);
        end
    end
endfunction

//For a partially specified W, returns 0/1 W, if possible
//v specifies if each problem instance is solved
function [W, v] = createW(W)
    W = floor(W);
    v = (sum(W) == size(W,1));
endfunction

//Creates a next split point vector
//k is the limit of the highest number
function o = createNextVector(o, k)
    for i=size(o,1):-1:1 do
        if(o(i) >= k)
            o(i)=0;
        else
            o(i)=o(i)+1;
            break;
        end
    end
endfunction

//*** helper functions

//Removes the rows that are not solved by either algorithm
function X = eliminateUnsolvable(X)
```

```
    r = find(max(X, 'c') <= -10000);
    X(r, :) = [];
endfunction

//Modifies the end value of the problem,
//if the standard non finish profit is < 10000
function X = modifyEndValue(X, nval)
    X(find(X <= -10000)) = nval;
endfunction

//Normalizes the vector
function x = normalize(x)
    x = x ./ sum(x);
endfunction

//Replaces all occurrences of %nan and %inf by zero
function x = naneliminate(x)
    b = isnan(x);
    c = isinf(x);
    x(b) = 0;
    x(c) = 0;
endfunction

//Reverses the input matrix in the specified direction
function X = reverse(X, d)
    if d == 'r' then
        X = X($:-1:1,:)
    elseif d == 'c' then
        X = X(:,$:-1:1)
    end
endfunction
```

## D.2 Dynamic Schedules

```
//The following must be executed in the console,
//for correct function of the algorithms
//global N O M;

//Finds an optimal dynamic schedule, given the set
//of training instances for bound optimization,
//the bound is equivalent to the end time
//icount - number of intervals, including the last one
//The end time = icount - 1
//t - type of optimization,
//t = 1 mean optimization, t = 2 limit optimization
//Example: dynOptimize(X,9,2)
function R = dynOptimize(X, icount, t)
    //N - new state values
    //O - old state values, they are replaced by new to save memory
    //M - successions, define which algorithm is optimal in which state
    global N O M;

    m = size(X, 2);
    s = ones(m,1)*icount;
    [p, q] = stateToPosition(s,icount);
    N = spzeros(p, q);
    M = spzeros(p, q);

    generateStates(zeros(m,1), icount-1, 1, icount, 0, t);
    O = N;
    for i=(icount-2):-1:0
        N = spzeros(p, q);
        generateStates(zeros(m,1), i, 1, icount, 1, t);
        O = N;
    end
    R = O;
endfunction


//Reconstructs the dynamic schedule from results calculated by
//dynOptimize
//algs - number of algorithms
//icount - number of intervals
function V = reconstructSequence(algs, icount)
    global M;
    s = zeros(algs, 1);
    V = zeros(icount);
    for i=1:icount do
        [p, q] = stateToPosition(s, icount);
        k = full(M(p,q));
        V(i) = k;
        s(k) = s(k)+1;
    end
endfunction

//Simulates a dynamic schedule on the provided instances
//X - instances
//V - dynamic schedule
//t - type, t = 1 mean, t = 2 bound simulation
function R = simulateD(X,V)
    R = zeros(size(X,1),1);
    S = zeros(size(X,1),1);
    s = zeros(size(X,2),1);
    for i=1:(size(V,1)-1) do
        k = V(i);
        H = X(:,k) >= s(k) & X(:,k) < s(k) + 1
        H = H & ~S;
        R(H) = X(H,k) + sum(s) - s(k);
        S = S | H;
        s(k) = s(k) + 1;
    end
    //similar, but for the last interval
    k = V($);
    H = X(:,k) >= s(k);
    H = H & ~S;
    R(H) = X(H,k) + sum(s) - s(k);
endfunction

//Calculates the reward of the last final states
//t - type of optimization,
//t = 1 mean optimization, t = 2 bound optimization
//s - the state
```

```
//k - the algorithm that was run
//X - the algorithm instance performance matrix
function r = calculateEndStateReward(s, k, X, t)
    C = zeros(size(X,1), 1);
    for j=1:size(X,1) do
        C(j) = sum((X(j,:) - s') < 0) == 0;
    end
    C = bool2s(C);
    if sum(C) > 0 then
        r = sum(X(:,k).*C) / sum(C) + sum(s)     - s(k);
    else
        r = 0;
    end
endfunction

//Calculates the reward and probability for solving an instance
//in the states that precede the last state
//k - the algorithm to be run
function [p,r] = calculateInnerStateReward(s, k, X, t)
    C = zeros(size(X,1), 1);
    D = zeros(size(X,1), 1);
    h = s(k)+1;

    for j=1:size(X,1) do
        C(j) = (sum(X(j,:) < s') == 0);
        D(j) = (X(j,k) < h);
    end

    C = bool2s(C);
    D = bool2s(D);
    E = C .* D;

    if(sum(E) > 0)
        if t == 1 then
            r = sum(X(:,k).*E)/sum(E) + sum(s)  - s(k);
        else
            r = -1;
        end;
        p = sum(E) / sum(C);
    else
        r = 0;
        p = 0;
    end
endfunction

//Calculate the utility of inner states from S,
//that is those that do not lead to last states is F
//s - the state
function calculateInnerStateUtility(s, icount, t)
    global N O M;

    [p,q] = stateToPosition(s,icount);
    if t == 1 then
        start = 1;
    else
        start = lastNonZero(s);
    end

    for k=start:size(s,1) do
        h = s; h(k) = h(k) + 1;
        [p2,q2] = stateToPosition(h,icount);
        [pp, pr] = calculateInnerStateReward(s, k, X, t);
        r = pp*pr + full(O(p2,q2))*(1-pp);

        o = full(N(p,q));
        if o == 0 then
            N(p,q) = r;
            M(p,q) = k;
        else
            if r < o then
                N(p,q) = r;
                M(p,q) = k;
            end
        end
    end
endfunction

//Calculate the utility of end states from S,
//that is those that lead to last states is F
//s - the state
function calculateEndStateUtility(s, icount, t)
    global N M;
    [p,q] = stateToPosition(s,icount);
    for k=1:size(s,1) do
        r = calculateEndStateReward(s, k, X, t)
        o = full(N(p,q));
        if o == 0 then
            N(p,q) = r;
            M(p,q) = k;
        else
            if r < o then
                N(p,q) = r;
                M(p,q) = k;
            end
        end
    end
endfunction

//Generates successively states and calculates their utility
//s - the current state
//l - the starting point
//g - size total sum
//O - old values, sparse matrix
//X - the statistics
//icount - the maximal value allowed for each state
//N - new values, sparse matrix
//ie == 0 -> end state, otherwise inner state
//Example: generateStates(zeros(3,1), 9, 1, 9, 0)
function generateStates(s, g, l, icount, ie, t)
    global N O;
    if l >= size(s,1) then
        s(l) = g;
        if ie == 0 then
            if t == 1 then
                calculateEndStateUtility(s, icount, t);
            end
        else
            calculateInnerStateUtility(s, icount, t);
        end
    else
        for i=0:g do
            s(l) = i;
            generateStates(s, g-i, l+1, icount, ie, t);
        end
    end
endfunction



//************ helper functions

//Transforms a state into coordinates in a 2-dimensional matrix
//s - state vector - column
//m - the maximal value for each position
//p - the first dimension
//q - the second dimension
function [p,q] = stateToPosition(s, m)
    p = 0;
    q = 0;

    k = 1;
    l = size(s, 1);

    for i=floor(l/2):-1:1 do
        p = p + k * s(i);
        k = k*m;
    end
    k=1;
    for i=l:-1:(floor(l/2)+1) do
        q = q + k * s(i);
        k = k*m;
    end
    p=p+1;  q=q+1;
endfunction

//returns the last non-zero element of the vector,
//returns 1 for a zero vector
function i=lastNonZero(s)
    for i=size(s,1):-1:1 do
        if s(i) ~= 0 then
            break;
        end
    end
endfunction
```

# BIBLIOGRAPHY

Anthony, M., Bartlett, P., Ishai, Y., & Shawe-Taylor, J. (1996). Valid generalisation from approximate interpolation. *Combinatorics, Probability and Computing*, *5*(1), 191–214.

Arnt, A., Zilberstein, S., & Allen, J. (2004). Dynamic composition of information retrieval techniques. *Journal of Intelligent Information Systems*, *23*(1), 67–97.

Baerentzen, L., & Talukdar, S. (1997). Improving cooperation among autonomous agents in asynchronous teams..

Bayardo, R. J. J., & Schrag, R. C. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pp. 203–208, Providence, Rhode Island.

Bertsekas, D. P. (2003). *Nonlinear Programming*. Athena Scientific.

Cesa-Bianchi, N., Freund, Y., Helmbold, D. P., Haussler, D., Schapire, R. E., & Warmuth, M. K. (1993). How to use expert advice. In *Annual ACM Symposium on Theory of Computing*, pp. 382–391.

Chatalic, P., & Simon, L. (2000). ZRes: The old Davis-Putnam procedure meets ZBDDs. In McAllester, D. (Ed.), *17th International Conference on Automated Deduction (CADE'17)*, pp. 449–454.

Chawla, N. V., Hall, L. O., Bowyer, K. W., & Kegelmeyer, W. P. (2004). Learning ensembles from bites: A scalable and accurate approach. *J. Mach. Learn. Res.*, *5*(1), 421–451.

Cicirello, V. A. (2003). *Boosting Stochastic Problem Solvers Through Online Self-Analysis of Performance*. Ph.D. thesis, Carnegie Mellon University.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158. ACM Press.

Dean, T., & M.Boddy (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*.

Devroye, L., Gyorfi, L., & Lugosi, G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer.

Downey, R. G., & Fellows, M. R. (1998). Parameterized complexity after (almost) 10 years: Review and open questions..

Elmieh, B. (1999). Fully polynomial time approximation schemes 1. Lecture notes.

Estlin, T. A. (1998). *Using Multi-Strategy Learning to Improve Planning Efficiency and Quality*. Ph.D. thesis, The University of Texas at Austin.

Etzioni, O., & Etzioni, R. (1994). Statistical methods for analyzing speedup learning experiments. *Machine Learning*, *14*(1), 333–347.

Etzioni, O., & Minton, S. (1992). Why ebl produces overly-specific knowledge: a critique of the prodigy approaches. In *ML92: Proceedings of the ninth international workshop on Machine learning*, pp. 137–143. Morgan Kaufmann Publishers Inc.

Finkelstein, L., & Markovitch, S. (1998). A selective macro-learning and its application to nxn sliding-tile puzzle. *Journal of Artificial Intelligence Reasearch*, *8*(1), 223–263.

Finkelstein, L., & Markovitch, S. (2001). Optimal schedules for monitoring anytime algorithms. *Artificial Intelligence*, *126*(1-2), 63–108.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, pp. 148–156.

Friedman, J., Hastie, T., & Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, *28*(1), ?

Giarratano, J., & Riley, G. (1998). *Expert Systems - Priciples and Programming.* PWS Publishing Company.

Gomes, C., & Selman, B. (2001). Algorithm portolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Gottlob, G., Leone, N., & Scarcello, F. (2000). A comparison of structural CSP decomposition methods. *Artificial Intelligence*, *124*(1), 243–282.

Gratch, J., & DeJong, G. (1996). A decision–theoretic approach to adaptive problem solving. *Artificial Intelligence*, *88*(1-2), 101–142.

Gratch, J., & Chien, S. (1996). Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, *4*(1), 365–396.

Horvitz, E. J., Breese, J. S., & Henrion., M. (1988). Decision theory in expert systems and artificial intelligence. *International Journal of Approximate Reasoning*, *2*(1), 247–302.

Huberman, B. A., Lukose, R. M., & Hogg, T. (1997). An economics approach to hard computational problems. *Science*, *275*(1), 51–56.

Hutter, M., & Poland, J. (2004). Prediction with expert advice by following the perturbed leader for general weights. In Ben-David, S., Case, J., & Maruoka, A. (Eds.), *Proc. 15th International Conf. on Algorithmic Learning Theory (ALT-2004)*, Vol. 3244 of *LNAI*, pp. 279–293, Padova. Springer, Berlin.

Hutter, M. (2001). The fastest and shortest algorithm for all well-defined problems. *Submitted to the International Journal of Foundations of Computer Science*, *1-2*(IDSIA-16-00), 12 pages.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive mixtures of local experts. In *Neural Computation.*

Kall, P., & Wallace, S. W. (1994). *Stochastic Programming.* Wiley, Chichester.

Kearns, & Schapire (1994). Efficient distribution-free learning of probabilistic concepts. In *Computational Learning Theory and Natural Learning Systems, Volume I: Constraints and Prospect, edited by Stephen Jose Hanson, George A. Drastal, and Ronald L. Rivest, Bradford*, Vol. 1. MIT Press.

Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory.* MIT Press, Cambridge, MA, USA.

Kivinen, J., & Warmuth, M. K. (1999). Boosting as entropy projection. In *Computational Learing Theory*, pp. 134–144.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers.

Lander, S. E., & Lesser, V. R. (1994). Sharing meta-information to guide cooperative search among heterogeneous resuable agents. Tech. rep. UM-CS-1994-048, University of Massachussets Amherst.

Lee, H. K. H., & Clyde, M. A. (2004). Lossless online bayesian bagging. *J. Mach. Learn. Res.*, *5*(1), 143–151.

Lesser, V. R. (1999). Cooperative multiagent systems: A personal view of the state of the art. *Knowledge and Data Engineering*, *11*(1), 133–142.

Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, *8*(3), 265–266.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003). Boosting as a metaphor for algorithm design.. In *CP*, pp. 899–903.

Li, C. M. (2005). Eqsatz. Website: http://www.laria.u-picardie.fr/ cli/EnglishPage.html.

Meek, C., Thiesson, B., & Heckerman, D. (2002). Staged mixture modelling and boosting. Tech. rep., Microsoft Reasearch.

Meir, R., & Ratsch, G. (2003). *Introduction to Boosting and Leveraging*, pp. 119–184. Springer.

Miagkikh, V. V., & III, W. F. P. (1999). An approach to solving combinatorial optimization problems using a population of reinforcement learning agents. In *Genetic and Evolutionary Computation Conference*.

Minton, S., Allen, J. A., Wolfe, S., & Philpot, A. (1995). An overview of learning in the Multi-TAC system..

Minton, S. (1993). An analytic learning system for specializing heuristics. In *IJCAI*, pp. 922–929.

Minton, S., & Underwood, I. (1994). Small is beautiful: A brute-force approach to learning first-order formulas. In *National Conference on Artificial Intelligence*, pp. 168–174.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill Book Co.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*.

Osborne, M. J., & Rubinstein, A. (1994). *A Course in Game Theory*. The MIT Press.

Owen, A., & Zhou, Y. (2000). Safe and effective importance sampling. *Journal of the American Statistical Association*, *95*(449), 135–155.

Oza, N. C., & Russell, S. (2001). Experimental comparisons of online and batch versions of bagging and boosting. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 359–364, New York, NY, USA. ACM Press.

Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial Optimization, Algorithms and Complexity*. Dover Publications, Inc.

Park, J. D., & Darwiche, A. (2004). Complexity results and approximation strategies for map explanations. *Journal of Artificial Intelligence Research*, *1*(1), 101–133.

Petrik, M. (2005). Statistically optimal combination of algorithms. In *Local Proceedings of SOFSEM 2005*.

Russell, S., & Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1997). Boosting the margin: a new explanation for the effectiveness of voting methods. In *Proc. 14th International Conference on Machine Learning*, pp. 322–330. Morgan Kaufmann.

Schmidhuber, J. (2002). Optimal ordered problem solver. Tech. rep., IDSIA.

Schmidhuber, J., & Wiering, M. (1996). Solving POMDPs with Levin search and EIRA. In *13th Int. Conf. on Machine Learning*.

Schmidhuber, J., Zhao, J., & Wiering, M. (1997). Shifting inductive bias with success-story algorithm, Adaptive Levin Search, and incremental self-improvement. *Machine Learning*, *28*(1), 105–130.

Shawe-Taylor, J., & Cristianini, N. (1998). Robust bounds on generalization from the margin distribution. Tech. rep., Royal Holloway, University of London.

Simon, L. (2005). Satex. Website: http://www.lri.fr/ simon/satex/satex.php3.

Stone, P., & Veloso, M. M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, *8*(3), 345–383.

Sutton, R., & Barto, A. (1998a). *Reinforcement Learning*, chap. 11, pp. 283–290. MIT Press.

Sutton, R. S., & Barto, A. (1998b). *Reinforcement Learning.* MIT Press.

Tadepalli, P. (1992). A theory of unsupervised speedup learning. In *National Conference on Artificial Intelligence*, pp. 229–234.

Tadepalli, P., & Natarajan, B. K. (1996). A formal framework for speedup leaning from problems and solutions. *Journal of Aritficial Intelligence Research*, *4*(1), 445–475.

Talukdar, S., Baerentzen, L., Gove, A., & de Souza, P. (1998). Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, *4*(1), 295321.

Tsang, E., Borrett, J., & Kwan, A. (1995). An attempt to map the performance of a range of algorithm and heuristic combinations. In *Hybrid Problems, Hybrid Solutions.* IOS Press.

Wallace, R. J., & Freuder, E. C. (1995). Anytime algorithms for constraint satisfaction and SAT problems. In *Working Notes of IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling, Montreal, Canada.*

Zhang, H. (1997). SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pp. 272–275.

Zilberstein, S. (1993). *Operational Rationality through Compilation of Anytime Algorithms.* Ph.D. thesis, University of California at Berkley.

Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, *17*(3), 73–83.