Learning Parallel Portfolios of Algorithms

Marek Petrik and Shlomo Zilberstein
Department of Computer Science
University of Massachusetts Amherst, MA 01003
{petrik,shlomo}@cs.umass.edu

April 16, 2007

Abstract

A wide range of combinatorial optimization algorithms have been developed for complex reasoning tasks. Frequently, no single algorithm outperforms all the others. This has raised interest in leveraging the performance of a collection of algorithms to improve performance. We show how to accomplish this using a Parallel Portfolio of Algorithms (PPA). A PPA is a collection of diverse algorithms for solving a single problem, all running concurrently on a single processor until a solution is produced. The performance of the portfolio may be controlled by assigning different shares of processor time to each algorithm. We present an effective method for finding a PPA in which the share of processor time allocated to each algorithm is fixed. Finding the optimal static schedule is shown to be an NP-complete problem for a general class of utility functions. We present bounds on the performance of the PPA over random instances and evaluate the performance empirically on a collection of 23 state-of-the-art SAT algorithms. The results show significant performance gains over the fastest individual algorithm in the collection.

Keywords: algorithm portfolios, resource bounded reasoning, combinatorial optimization

1 Introduction

Research advances in search and automated reasoning techniques have produced a wide range of different algorithms for hard decision problems. In most cases, there is no single algorithm that is superior to all others on all instances. A good example is satisfiability (SAT), for which many algorithms have been developed, with no single algorithm that dominates the performance of all the others. SAT is particularly interesting because it is not representing a single application; it is a prominent problem in both theoretical and

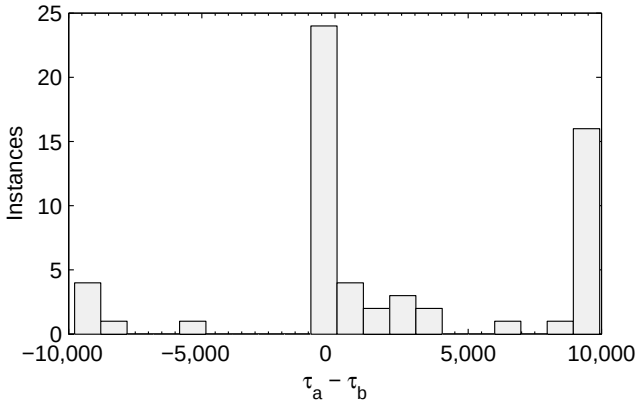

Figure 1: Number of SAT instances for various difference in runtimes of 2 algorithms. Runtimes in seconds are denoted as τ_a for eqsatz and τ_b for zChaff.

applied computer science with many important applications in artificial intelligence, operation research, electronic design and verification.

The objective of this work is to leverage a collection of algorithms to produce a method that outperforms any single algorithm in the collection when applied to some class of problem instances. To illustrate the motivation for this work, Figure 1 shows the relative performance of two SAT algorithms from the Sat-Ex collection [13]. It is obvious that each one of the algorithms outperforms the other on a significant number of instances. The performance difference varies over 10000 seconds–the time bound used in the original data-set for solving each instance. We propose a combination of algorithms, which takes advantage of this fact. We do not assume to have any prior information about the performance of each algorithm (such as a performance profile); instead, we gather the necessary performance information from a sample test set.

A Parallel Portfolio of Algorithms (PPA) is thus a collection of algorithms for solving a certain type of problem, such as SAT. When a particular problem instance is given, all the algorithms are launched and execute in an interleaved manner on a single processor. In practice, many computational resources may act as bottlenecks, such as processor time, memory space, or communication throughput. We focus in this paper only on processor time as the limiting factor of the computation. Thus, CPU time is distributed among the algorithms so as to optimize the expected performance (of the entire portfolio) on a set of training instances.

In the interest of clarity, we focus on the application of this framework to *decision* problems, but it can be easily extended to *optimization* problems. In the case of decision problems, the PPA is designed to minimize the expected time needed to solve problem instances. In the case of optimization problems, the available time is constant and the PPA is designed to maximize the expected quality of the solution. To fit into this framework,

the individual algorithms must be preemptable without a significant overhead (i.e., it is assumed that they can be stopped and resumed with negligible overhead). Furthermore, optimization algorithms must have the anytime property [3, 14]. Anytime algorithms may be interrupted before their termination and return an approximate solution. The quality of the solution typically improves with computation time.

Prior knowledge of the algorithms' performance on instances may enhance the performance of a portfolio by always using the most suitable algorithm for each instance. But it is usually impossible to have perfect prior knowledge without actually solving the instance. However, there have been some research efforts to estimate the performance by the similarity of the input instance to instances with known performance. This way, easily-computable features of the problem instance may help to approximately determine the performance of each algorithm on that instance. Finding informative features that always enhance the prediction of performance is a hard problem. It is generally known as *algorithm portfolio selection* [5, 8].

In related work, Gomes and Selman [5] provide an empirical evaluation of composing randomized search algorithms on multiple processors into portfolios. Their results indicate that it may be beneficial to combine randomized algorithms with high variance and run them in parallel. For a single processor, they found random restarts to produce very good results. A successful algorithm portfolio selection approach was also taken by [8]. It is based, like many other approaches, on using a training set of problem instances to estimate a probabilistic model of the performance. Statistical regression is used to determine the best algorithm for instances of combinatorial auction winner determination. Unlike previous approaches that use experimental methods to determine the composition of the portfolio, our approach defines a formal model for the problem, and analyzes the theoretical properties of the model and solution techniques.

The rest of the paper is organized as follows. Section 2 defines the framework used to model algorithm portfolios and their performance. In Section 3, we present an algorithm for finding a locally optimal PPA for a specified set of algorithms. A globally optimal algorithm and computational complexity bounds are described in Section 4 and Section 5. The following section, Section 6, presents basic theoretical generalization bounds of the approach. Finally, in Section 7 we demonstrate the effectiveness of this approach by applying it to a collection of 23 state-of-the-art Satisfiability (SAT) algorithms.

2 Framework

In this section we develop a general framework for creating parallel portfolios of algorithms. As indicated above, we develop the framework specifically for decision algorithms, but the extension to optimization algorithms is straightforward. We mention briefly the necessary modifications throughout the paper.

A *problem domain* is a set of all possible problem instances together with a probability

distribution over them, denoted as $\mathcal{X}$. The probability distribution of encountering the instances is *stationary*. An *algorithm* is a function that maps an input problem instance to a solution. Notice that the algorithms are assumed to be deterministic.

To preserve flexibility of the framework, we use the notion of *utility* that corresponds to the performance of an algorithm on an instance. For standard decision algorithms, it depends on the computation time; for optimization algorithms, it depends on the quality of the solution within a certain amount of time. Thus the *utility function* is defined as $u : \mathcal{X} \to \mathbb{R}$. The PPA performance will be optimized with respect to a *training set* I of instances, randomly drawn from the domain $\mathcal{X}$. The following definitions assume instances from the set $\mathcal{X}$, which may be extremely large or infinite. However, the proposed schedule optimization algorithms optimize the portfolio with regard to utility on the training set I.

The performance of a PPA depends on the processor share that is assigned to each algorithm. The share assigned to an algorithm at each point of time is determined by its resource allocation function, as defined below. Resource allocation functions for all algorithms in the portfolio comprise the schedule. In this paper we consider only constant allocation functions, but they can be extended to a wider class of functions, as shown in [12].

Definition 2.1. The *resource allocation function* (RAF) for an algorithm a is a constant from the interval $[0, 1]$ that defines its share of the processor time.

We now extend the notion of *utility* to be determined not only by an algorithm and an instance, but also by its resource allocation function. The precise function depends on the actual algorithm.

Definition 2.2. The *resource utility function* of algorithm a is: $u_a : [0, 1] \times \mathcal{X} \to \mathbb{R}$.

Definition 2.3. A *Static Parallel Portfolio of Algorithms* (SPPA) $\mathcal{P}$ is a tuple (A, s), where A is a finite list of algorithms with resource utility functions and s is a *schedule*, that is a vector of resource allocations for algorithms from A.

The resource allocation to the jth algorithm of the portfolio is denoted as $s[j]$. The resource allocations must fulfill the resource limitation constraint, so the set of allowed schedules is $S = \{s \mid \sum_{j=1}^{n} s[j] \leq 1\}$.

In the following, $m = |I|$ and $n = |A|$. An intrinsic characteristic of PPAs is that only one result for a problem instance may be used, even if more algorithms provide solutions. Therefore, the PPA utility of an instance is the maximal utility of all individual algorithms. This is in contrast with ensemble learning, where classifiers vote and the ensemble result is the average of the votes.

Definition 2.4. The *utility* $U(s, x)$ of a PPA is the maximum utility that any algorithm of the portfolio achieves on instance $x \in \mathcal{X}$ using schedule $s \in S$, that is:

$$U(s, x) = \max_{j=1,\ldots,n} u_j(s[j], x).$$

We denote the average performance of the schedule on the training set instances as

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

Then the problem of selecting the best schedule is defined as

$$\bar{s} = \arg\max_{s \in S} U(s).$$

Some additional measures of performance are proposed in [12].

The definition of the utility of an algorithm with respect to a problem instance depends on the specific problem setting. For decision problems, the available time is unlimited, but it is preferable to have a PPA that solves the problems as fast as possible. Assuming the algorithms may be executed on a single processor with no overhead, the utility function is defined as follows.

Definition 2.5. For decision algorithms, the time to find the first solution is optimized if the resource utility function u for an algorithm is

$$u(r, x) = -\frac{\tau_x}{r},$$

where τ_x is the time to solve the instance x and r is the resource allocation function. The function u is concave and twice continuously differentiable in r.

To apply the framework to optimization problems, we may assume that the computation time is fixed and the quality of the obtained solutions is important. Anytime algorithms can be characterized by their performance profile function f that maps the computation time to the current solution quality. Let T be the time allocated for the execution of the PPA. Then, the resource utility function that maximizes the obtained quality is

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

3 The Classification-Maximization Algorithm

In this section we present an algorithm to find locally optimal schedules for SPPAs. The task of finding the optimal static schedule on I may be formulated as the following non-linear mathematical optimization problem:

$$\begin{array}{lrcl} \text{maximize} & U(s) & = & \displaystyle\sum_{i=1}^{m} \max_{j=1,\ldots,n} u_j(s[j], x_i) \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} s[j] & = & 1, \\ & s[j] & \geq & 0 \quad j = 1, \ldots, n \end{array} \tag{1}$$

This problem is not solvable by standard non-linear programming techniques because the inner max operator makes the objective function discontinuous. To solve the program, we propose to decompose the solution process into two phases: classification and maximization. Hence the Classification-Maximization Algorithm.

The algorithm is based on a reformulation of (1) that removes the inner maximization. In order to do this, we introduce a classification matrix $W = m \times n$. Notice that we do not assume W to be given, but it represents additional optimization variables. Intuitively, the entries of the matrix indicate the best algorithm for each problem instance. In other words, the entry $W_{ij} = 1$ if and only if algorithm j has the highest utility for instance i, given the resource allocation. Clearly, for each instance exactly one algorithm is optimal, breaking ties arbitrarily. Therefore, the matrix must fulfill:

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

Using the matrix W, we can divide the set of instances I into subsets $I_j, j = 1, \ldots, n$ according to which algorithm the instance is assigned. Formally, we can write

$$I_j = \{x_i \,|\, W_{ij} = 1\},$$

with the cardinality of these sets denoted as $m_j = |I_j|$.

To simplify the notation, we introduce a *classification function* $k(l, j)$, which denotes the index of the l-th instance from I_j. Formally, it is defined as the minimal i' that satisfies $\sum_{i=1}^{i'} W_{ij} = l$.

Using a classification matrix W, the problem may be reformulated as

$$\begin{aligned}
\text{maximize} \quad U(s, W) &= \sum_{i=1}^{m}\sum_{j=1}^{n} W_{ij} u_j(s[j], x_i) \\
\text{subject to} \quad \sum_{j=1}^{n} s[j] &= 1, \\
\sum_{j=1}^{n} W_{ij} &= 1 \quad i = 1, \ldots, m, \\
s[j] &\geq 0 \quad j = 1, \ldots, n, \\
W_{ij} &\in \{0, 1\} \quad i = 1, \ldots, m \\
& \qquad\qquad\qquad\quad j = 1, \ldots, n
\end{aligned} \tag{2}$$

To illustrate the possible values of W, consider a simple example with two algorithms a_1 and a_2, and three instances x_1, x_2, and x_3. Assume that the result of the optimization

Randomly initialize W^0
$i \leftarrow 0$
while $U(s^i, W^i) > U(s^{i-1}, W^{i-1})$ **do**
  $s^{i+1} \leftarrow \arg\max_s U(s, W^i)$
  $W^{i+1} \leftarrow \arg\max_W U(s^{i+1}, W)$
  $i \leftarrow i + 1$
end while

Figure 2: General CMA.

is a schedule such that a_2 has the best performance for x_1 and x_2, and a_1 has the best performance for x_3. Then, the matrix W will be:

$$W = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The two formulations (1) and (2) can be shown to be equivalent as follows. Given an optimal solution s_1^* of (1), we can construct a matrix W, such that $W_{ij} = 1$ if algorithm j has the highest utility for instance i, given the schedule s_1^*. Since $U(s_1^*) = U(s_1^*, W)$, the optimal objective function of (2) is greater than or equal to the optimal objective function of (1). Given an optimal solution s_2^*, W of (2), the utility of the portfolio on each instance will be at most the utility of the highest-utility algorithm, since $W_{ij} \leq 1$. Therefore, (1) and (2) are equivalent.

The overall CMA works as follows. In the classification phase, it finds the optimal W for a fixed s. In the maximization phase, it finds the optimal s for a fixed W. Generally, this approach is known as Block Coordinate Descent, or Gauss-Seidel optimization [2]. Because this approach leads only to a local maximum, its performance may be enhanced by randomly choosing the initial classification over several executions. The general structure of the CMA is shown in Figure 2. We further elaborate on the individual phases.

3.1 Classification

The classification phase can be performed analytically, as the following proposition states.

Proposition 3.1. *The solution of (2) with an optimal utility for a fixed s is achieved if and only if*

$$W_{ij} = 1 \Leftrightarrow u_j(s[j], x_i) \geq \max_{k=1,\ldots,m} u_k(s[k], x_i)\ . \tag{3}$$

In other words, the optimal classification is the one that actually corresponds to the schedule.

Proof. The proof of necessary optimality condition is by contradiction. Assume that the optimal solution does not satisfy (3). So let $W_{i\hat{j}} = 1$, be a classification with a suboptimal utility. Let $j^* = \arg\max_{j=1,\ldots,m} u_j(s[j], x_i)$, breaking ties arbitrarily. Clearly, setting $W_{i\hat{j}} = 0$ and $W_{ij^*} = 1$ does not invalidate the classification constraints. Since this holds for all i, the monotonicity of $\sum$ implies that the utility is not decreased. Therefore, the solution that fulfills the condition must also have the optimal utility. The proof of sufficient optimality condition is similar. □

3.2 Maximization

For the maximization phase, we need the following assumption on the resource utility functions.

Definition 3.2. A set A of resource utility functions u_j is *homogeneous* if each utility function can be expressed as

$$u_j(r, x) = \rho_j(r) * \eta_j(x).$$

Notice that the resource utility functions are homogeneous for decision algorithms. If the resource utility functions are homogeneous then the maximization phase is equivalent to solving

$$\begin{array}{lrcl} \text{maximize} & U(s) & = & \displaystyle\sum_{j=1}^{n} \rho_j(s[j]) \sum_{i=1}^{m_j} \eta_j(x_{k(l,j)}) \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} s[j] & = & 1 \\ & s[j] & \geq & 0 \quad j = 1, \ldots, n \end{array} \tag{4}$$

Let $d_j = \sum_{i=1}^{m_j} \eta_j(x_{k(i,j)})$. The problem then becomes the following.

$$\begin{array}{lrcl} \text{maximize} & U(s) & = & \displaystyle\frac{1}{m} \sum_{j=1}^{n} \rho_j(s[j]) d_j \\ \text{subject to} & \displaystyle\sum_{j=1}^{n} s[j] & = & 1 \\ & s[j] & \geq & 0 \quad j = 1, \ldots, n \end{array} \tag{5}$$

This is easily solved by first order necessary conditions, and from the fact that the problem involves a maximization of a concave function on a convex set. Hence the following theorem.

Theorem 3.3. *The solution of (5) is globally optimal if it fulfills*

$$\frac{\rho_j'(s[j])}{\rho_k'(s[k])} = \frac{d_k}{d_j}.$$

Proof. The Lagrangian function of (5) is

$$L(s,\lambda) = -\frac{1}{m}\sum_{j=1}^{n}\rho_j(s[j])d_j + \lambda\sum_{j=1}^{n}s[j].$$

The theorem then follows from the second order optimality conditions [2]. Because the of the schedulability assumption, it also fulfills the convexity criterion. Thus this local maximum is also global. □

The theorems above complete the definition of CMA. In general, CMA is a suboptimal algorithm.

3.3 Decision Problems

For decision problems with no switching overhead, the utility function is formulated according to Definition 2.5. Thus we have that:

$$\rho_j(r) = \frac{1}{r}\eta_j(x) = \tau_j(x).$$

Now the maximization phase of the CMA can be solved as the following theorem states.

Theorem 3.4. *Let the SPPA use decision algorithms. The mean optimal schedule for the maximization phase of CMA, given a fixed classification, must fulfill*

$$\frac{s[o]}{s[p]} = \sqrt{\frac{\sum_{i=1}^{m_o}\tau_o(x_{k(i,o)})}{\sum_{i=1}^{m_p}\tau_p(x_{k(i,p)})}}.$$

Proof. By Theorem 3.3. □

Theorem 3.5. *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*

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

Proof. The theorem follows directly from Theorem 3.4. □

4 Optimal Algorithm

It is possible to find an optimal static schedule by searching for the optimal schedule for every possible classification. This approach leads to the optimal solution because the locally optimal schedule for the optimal classification is also globally optimal. The computational complexity of this approach is very high because the number of these classifications is exponential in the number of instances. However, not all of the classifications are possible when actually running the portfolio. For example, it is possible that if algorithm a_1 outperforms algorithm a_2 on instance x_2 then it also outperforms it on x_1. Thus, a classification that assigns x_2 to a_1 and x_1 to a_2 cannot happen during a real execution of a PPA. When we assume that the algorithms in the portfolio have homogeneous utility functions, we can limit the number of all realistic classifications to be polynomial in the number of instances and exponential in the number of algorithms as we show this section. The results from this section also help to establish the generalization bounds in Section 6.

Definition 4.1. A classification is *valid* when there exists a schedule that exactly defines it.

It is clear from Lemma 3.1 that for any classification that is not valid there is a valid classification with better or equal performance. Thus, limiting the search to valid classifications is sufficient to guarantee that an optimal schedule will be found.

Lemma 4.2. *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*

$$U(s^*) = U(\widehat{s}).$$

Proof. We show that the lemma holds by contradiction. Let

$$U(s^*) > U(\widehat{s}).$$

Clearly, a classification W^* defined by schedule s^* is valid. Then, by running the maximization phase of CMA on this classification, we get a schedule s'. Then, we get

$$U(s') \geq U(s^*) > U(\widehat{s}),$$

which contradicts the optimality condition of $\widehat{s}$. □

The set of valid schedules depends on the properties of the resource utility functions. In particular, we focus on the homogeneity of the functions. The main reason is that the resource utility functions of decision algorithms are homogeneous.

Lemma 4.3. *Let the resource utility function be homogeneous. Let a, b be arbitrary algorithms from a PPA. Then, for any valid classification I_a, I_b, we have*

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

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

Notice that this lemma is applicable also for the case of more than two algorithms. Then, $I_a \cup I_b \subset I$, i.e., $I_a \cup I_b$ is a proper subset of I.

Proof. Because the classification is valid,

$$\begin{aligned} u_a(x_o) &\geq u_b(x_o) \\ u_a(x_p) &\geq u_b(x_p). \end{aligned}$$

¿From the homogeneity assumption, we get that

$$\begin{aligned} \eta_a(x_o)\rho_a(r_a) &\geq \eta_b(x_o)\rho_b(r_b) \\ \eta_a(x_p)\rho_a(r_a) &\leq \eta_b(x_p)\rho_b(r_b). \end{aligned}$$

Hence

$$\begin{aligned} \frac{\rho_b(r_b)}{\rho_a(r_a)} &\leq \frac{\eta_a(x_o)}{\eta_b(x_o)} \\ \frac{\rho_b(r_b)}{\rho_a(r_a)} &\geq \frac{\eta_a(x_p)}{\eta_b(x_p)}, \end{aligned}$$

and the lemma follows. □

As a consequence, we have the following lemma.

Lemma 4.4. *Let the resource utility functions be homogeneous. For any two instances x_o, x_p, and algorithms a and b, without loss of the generality*

$$\frac{\eta_a(x_o)}{\eta_b(x_o)} > \frac{\eta_a(x_p)}{\eta_b(x_p)}.$$

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 4.3 leads to

$$\frac{\eta_a(x_o)}{\eta_b(x_o)} \leq \frac{\eta_a(x_p)}{\eta_b(x_p)},$$

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

$i = 0$
$U^* = 0$
Initialize split points $(p_1, \ldots, p_{\binom{n}{2}}) = (0, \ldots, 0)$
for all $(p_1, \ldots, p_{\binom{n}{2}}) \in (0 \ldots m)^{\binom{n}{2}}$ **do**
  Create W from $p_1, \ldots, p_{\binom{n}{2}}$
  if W is valid **then**
    $U \leftarrow \max_s U(s, W)$
    $U^* \leftarrow \max\{U, U^*\}$
  end if
end for

Figure 3: Optimal CMA.

Then, from Lemma 4.4, for each schedule there is a *split point* D, for which *no* instances with $\frac{\eta_a(x)}{\eta_b(x)} > D$ are assigned to b and *no* instances with $\frac{\eta_a(x)}{\eta_b(x)} < D$ are assigned to a. This property can be easily used to limit the number of all valid classifications. The following theorem states this fact precisely.

Theorem 4.5. *The number of valid classifications of instances to problems, when RAF are homogeneous, is at most*

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

Proof. We can see from Lemma 4.4 that every valid classification has at least one set of $\binom{n}{2}$ split points. 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 algorithms. Let these algorithms be a and b. This makes the split point between a and b different for each classification. Because there are at most

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

possible split-point sets, the theorem follows. □

As a result of the above theorem, we propose an Optimal Classification Maximization Algorithm (OCMA), depicted in Figure 3. It iterates through all split-point sets, creating a classification for each one, and calculating the optimal schedule for each classification. A classification is created from a split-point set by assigning an instance to an algorithm if and only if it is assigned to the algorithm for each split-point. Notice that the conditions on valid schedules are necessary, not sufficient. Therefore, the algorithm possibly also checks classifications that are not valid.

Theorem 4.6. *Optimal CMA finds a schedule with the optimal utility with complexity:*

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

Proof. The utility is optimal because each valid schedule has a unique split-point set by Lemma 4.4, and it is sufficient to enumerate all valid schedules by Lemma 4.2. The complexity is evident from the total number of split-point sets and Theorem 3.3. □

¿From the above analysis we have an algorithm that is polynomial in the number of instances and exponential in the number of algorithms. Moreover, the OCMA enumerates also classifications that are not valid. Thus it is questionable whether the exponential complexity of the OCMA is caused by its inefficient enumeration or whether the number of valid classifications is exponential. We show, to address this issue, that simply enumerating all valid classifications cannot lead to a polynomial algorithm, and later in Section 5, we also show that the general problem of finding the optimal schedule for a set of homogeneous algorithms is NP-hard.

Proposition 4.7. *The number of valid classifications may be exponential in the description of the problem, even when using decision algorithms.*

Proof. To prove the proposition, we show an example with an exponential number of valid classification. We construct a scheduling instance with $n+1$ algorithms and n instances for any n. Let the algorithms be $A = \{a_0 \dots a_n\}$ and instances $\mathcal{X} = \{x_1 \dots x_n\}$. Since we are dealing with decision algorithms, the utility on the instances is determined by the solution time $\tau_a(x)$ on each instance x. We define these solution times as follows:

$$\begin{aligned} \tau_0(x_i) &= 1 \quad i = 1, \dots, n \\ \tau_i(x_i) &= 1 \quad i = 1, \dots, n \\ \tau_j(x_i) &= 2 \quad i, j = 1, \dots, n \text{ and } i \neq j \end{aligned}$$

Now we show that any classification of instances to algorithm a_1 is possible, thus creating at least 2^i distinct classifications. Let $\sigma(x_i)$ be the indicator function whether x_i is assigned to algorithm a_0. For arbitrary σ, we can define schedule s as follows to achieve the proper instance assignment:

$$\begin{aligned} s[0] &= \epsilon' * \frac{1}{n+1} \\ s[j] &= \epsilon * \frac{1}{n+1} \quad \sigma(j) = 0 \\ s[j] &= \phi * \frac{1}{n+1} \quad \sigma(j) = 1, \end{aligned}$$

where

$$\begin{aligned} \phi < \epsilon' < \epsilon < 2\epsilon' \\ \epsilon' + (n - |\sigma|) * \epsilon + |\sigma| * \phi = n + 1. \end{aligned}$$

Such ϵ, ϵ', and ϕ always exist and the conditions ensure that the schedule sums to 1. The classification of this schedule clearly satisfies σ, since:

$$\begin{aligned}
\sigma(x_i) = 1 \quad &\Rightarrow \quad \frac{\tau_0(x_i)}{s[0]} = \frac{n+1}{\epsilon'} < \frac{n+1}{\phi} = \frac{\tau_i(x_i)}{s[i]} \\
\sigma(x_i) = 1 \quad &\Rightarrow \quad \frac{\tau_0(x_i)}{s[0]} = \frac{n+1}{\epsilon'} < \frac{n+1}{\frac{\epsilon}{2}} = \frac{2(n+1)}{\epsilon} = \frac{\tau_j(x_i)}{s[j]} \\
\sigma(x_i) = 0 \quad &\Rightarrow \quad \frac{\tau_0(x_i)}{s[0]} = \frac{n+1}{\epsilon'} > \frac{n+1}{\epsilon} = \frac{\tau_i(x_i)}{s[i]}.
\end{aligned}$$

□

As a result, just enumerating the valid schedules cannot lead to a polynomial algorithm. Thus, the exponential complexity of the OCMA cannot be resolved by a better enumeration of valid schedules.

5 Complexity Analysis

This section describes the complexity analysis of the SPPA scheduling problem. Specifically, the SPPA scheduling problem is the question whether there is a schedule s such that $U(s) \geq K$. We show that this problem is NP hard.

Definition 5.1. In a *0-1 knapsack problem* a set of positive integer pairs $\{(c_i, w_i) \mid i = 1, \ldots, N\}$ is given. The problem is whether there exists a subset T such that $\sum_{i \in T} c_i \geq C$ and $\sum_{i \in T} w_i \leq W$.

The 0-1 knapsack problem is NP complete [10, 11]. We assume below that a description of the utility function is included in the scheduling problem.

Theorem 5.2. *The scheduling problem is NP-hard even for homogeneous utility functions.*

The proof of this theorem may be found in Appendix A.1.

We proved NP-hardness for the problem of finding schedules for algorithm with general resource utility functions. While this result is not directly applicable to decision algorithms, it indicates that homogeneity of the resource utility functions is not sufficient to make the problem easier. The main difficulty in complexity analysis of scheduling decision algorithms is that it is not a combinatorial problem, because their resource utility functions are continuous.

6 Theoretical Generalization Bounds

This section describes the generalization properties of the SPPA approach. Since the schedules are based on a training set of instances, it is important to be able to predict

how well the SPPA may perform on $\mathcal{X}$, the domain of all problem instances. We prove only the worst case bounds.These bounds have mostly theoretical significance as they show interesting asymptotic generalization behavior. The bounds in this section are motivated by the Probably Approximately Correct (PAC) learning framework [9].

The main goal is to show that the number of training instances required to learn an SPPA that performs well on all instances is reasonably small. Because training instances are drawn randomly, the bounds only hold with a certain probability. Note that the bounds apply to the meta-level scheduling algorithm, such as CMA, that optimizes the SPPA schedule.

In this section, we assume that all SPPAs we refer to are composed of the same set of algorithms; they only differ in the actual schedules. Notice that since an SPPA also contains the schedule, it behaves just as a simple algorithm. Thus, the utility of an SPPA on the training set is its *empirical mean utility*, defined as:

$$U_m(s) = \frac{1}{m} \sum_{i=1}^{m} U(s, X_i),$$

where X_i is a random variable, representing an instance randomly drawn from $\mathcal{X}$. Notice that $U_m(s)$ is a random variable.

We define two properties of SPPAs, *generalization* and *optimality*. An SPPA learning algorithm generalizes well, when the utility on all instances is close to the utility on the training set. An SPPA learning algorithm is optimal, if the optimal SPPA on the training set is close to the optimal result on the set of all instances. These properties are formalized by the following definition.

Definition 6.1. We say that an SPPA learning algorithm *mean-generalizes*, if for any $0 < \epsilon$ and $0 < \delta < 1$ it outputs an SPPA $s \in S$, for which

$$\mathbf{P}\left[U_m(s) - \mathbf{E}\left[U(s, X)\right] > \epsilon\right] \leq \delta.$$

Let the globally optimal algorithm be:

$$s^* = \arg\max_{s \in S} \mathbf{E}\left[U(s, X)\right].$$

We say that an SPPA learning algorithm is *mean optimal*, if for all $0 < \epsilon$ and $0 < \delta < 1$ it outputs a schedule s

$$\mathbf{P}\left[\mathbf{E}\left[U(s^*, X)\right] - \mathbf{E}\left[U(s, X)\right] > \epsilon\right] \leq \delta.$$

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

The following theorem probabilistically bounds the expected performance of an SPPA on a sample of size m.

Theorem 6.2. *Let the resource utility functions be homogeneous. Let* $\widehat{u}$, $\widehat{\eta}$, $\widehat{\rho}$ *be the maximal values of corresponding functions for the given set of algorithms. In addition, let* $\frac{m*\epsilon^2}{\widehat{u}^2} > 2$. *Then, the generalization probability is*

$$\begin{aligned}&\mathbf{P}\left[\sup_{s\in S}|U_m(s) - \mathbf{E}\left[U(s,X)\right]| > \epsilon\right]\\ &\leq \quad 8(m+1)^{\binom{n}{2}}2^n \exp\left(\frac{-m\epsilon^2}{32\widehat{u}}\left(\frac{1}{\widehat{\rho}\widehat{\eta}n}\right)^2\right).\end{aligned}$$

The proof of this theorem is quite technical and it is in Appendix A.2.

Theorem 6.3. *Let the assumptions of Theorem 6.2 hold. Also let*

$$z = (\widehat{\rho}\widehat{\eta}n)^2\widehat{u}.$$

An algorithm that finds a static schedule by maximizing the empirical mean utility will find the ϵ *mean optimal SPPA with probability at least* $1-\delta$ *using at most*

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

samples.

Proof. The proof is based on Problem 12.5 from [4]. Let

$$d = \frac{\epsilon^2}{\widehat{u}}\left(\frac{1}{\widehat{\rho}\widehat{\eta}n}\right)^2.$$

Further, whenever $m \geq \frac{512\binom{n}{2}}{d}\ln\frac{256\binom{n}{2}}{d}$ we have the following bound

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

The theorem follows straight forward from the bound. □

Remark 6.4. A tighter bound could be obtained using Vapnik-Chervonenkis results that assume that the training set can be learned with zero training error, as in Section 12.7 of [4].

a	$\mathbf{E}[u(I)]$	$\mathbf{Var}[u(I)]$	$\mathbf{E}[u(I_S)]$	$\mathbf{Var}[u(I_S)]$
zChaff	372.3	2633.3	251.1	2140.2
relsat	715.0	3599.2	595.9	3274.3
relsat	951.0	4198.5	833.4	3934.4
sato	994.4	4103.7	877.0	3833.7
SPPA	233.2	2005.6	77.1	1000.5

Table 1: Results of a few best-performing algorithms from Sat-Ex and the best SPPA after 200 runs of CMA.

7 Application

We examined the applicability of the SPPA approach to combining 23 state-of-the-art algorithms for the satisfiability problem (SAT). The SAT problem is to determine whether a formula in propositional logic is satisfied for at least one interpretation. It offers a general framework for problem solving and planning [7]. The performance results of the algorithms were taken from [13]. The results are from runs of the algorithm on 1,303 instances. The cutoff execution time was 10,000 seconds. For the purpose of evaluation, we used a solution time of 20,000 seconds for instances that were not solved within the 10,000-second limit to emulate the possible expected solution time. The choice of this value did not have a significant impact on the results.

The set of all available instances from [13] is denoted as I. The set of all instances that are solvable by at least one available algorithm is denoted as I_S. The best performing algorithm from the group was zChaff, with average execution time 372 seconds on I and 251 on I_S. The performance of the four fastest algorithms on the set is summarized in Table 1.

We tested CMA for both I and I_S. In both cases the CMA-calculated SPPA significantly outperforms the best algorithm for the instance set (zChaff). The performance of the SPPA was derived analytically from the available performance data of individual algorithms. In the case of I, the mean execution time was lower by 37%, and the standard deviation was lower by 24%. In the case of I_S, the mean execution time was lower by 68% and the standard deviation was lower by 55%. The impressive results on I_S were mainly due to the fact that each instance is solved by at least one algorithm, thus reducing the very long runtime for those instances. The results are summarized in Table 1.

Specific schedules that were obtained are shown in Table 2. We used all 23 available algorithms for calculating the optimal schedules, but only the listed algorithms had non-zero processor share for the two schedules. It is interesting that though zChaff is the fastest algorithm, the fraction of processor it uses is very small in both schedules. The intuitive explanation is that zChaff is fast on instances that are hard to solve, but for the rest, it is

Algorithms	$\mathcal{P}_1$	$\mathcal{P}_2$
eqsatz	0.857	0.104
nsat	0.002	0.002
ntab-back2	0.030	0.026
heerhugo	0.004	0.004
satz	0.014	0.739
zChaff	0.093	0.125
Performance	233.2	294.8

Table 2: Summary of two SPPAs calculated using CMA with a different starting classification. $\mathcal{P}_1$ is the best SPPA we obtained using CMA. $\mathcal{P}_2$ is a locally optimal SPPA obtained by CMA from a different initialization.

I'	$\mathbf{E}[U \mid I']$	$\mathbf{E}[U \mid I]$	$\mathbf{E}[U \mid I'']$	$\mathbf{E}[u(a) \mid I'']$
I_1	399.8	387.4	359.4	373.2
I_2	225.1	418.8	856.1	280.4
I_3	380.1	311.7	268.7	310.9
I_4	228.5	297.3	340.6	390.5
I_5	225.2	259.7	281.4	375.0

Table 3: Performance of SPPA U on the training set I and test set $I'' = I \setminus I'$. The schedules were obtained by CMA for I'. The performance of zChaff, the overall fastest SAT algorithm, is denoted as $u(a)$.

a little slower than some other algorithms.

Certainly, the fact that we used the same set for obtaining the optimal SPPA and for evaluation adds a significant bias toward the method. Nevertheless, it provides a useful optimistic limit on the possible gain by using multiple algorithms, which is substantial. To address this issue, we also evaluated the generalization properties experimentally. Unfortunately, the small size of the overall instance database limited the significance of these results. But we managed to demonstrate the benefit of SPPAs in this case as well. We randomly choose subsets of all instances and used CMA to find the locally best algorithm for the set. Then, we evaluated the performance on the set of all algorithms. The instance sets of size 400 were I_1 and I_2. Instance sets of size 800 were I_3, I_4, and I_5. The results are shown in Table 3. On 4 out of 5 training sets, the best SPPA significantly outperformed zChaff on the test set.

8 Conclusions

The main idea of Parallel Portfolios of Algorithms is to concurrently run several algorithms for the same problem and to tune their performance by controlling the distribution of processor time to each algorithm. We focus in this paper on static schedules, in which processor shares are constant during the execution. In this case, it is possible to devise a fast, but suboptimal algorithm for calculating schedules. The results on SAT problems demonstrate the effectiveness of the approach for practical applications.

As shown in [12], it is also possible to find optimal non-static schedules in which a change of resource allocation function is allowed in discrete intervals. The optimality is with regard to expected computation time. These schedules may be obtained by solving a corresponding Markov Decision Process, which is generally more complex than calculation of static schedules. They can be shown to be optimal with regard to expected computation time. However, they do not show a significant improvement over static schedules on the presented SAT problem.

The schedule optimization problem (2) is in fact an instance of the general class of concave minimization problems. These are hard mathematical programming problems, but with a large number of sophisticated algorithms. A good overview may found in [6]. An application of these techniques to the portfolio optimization problem may lead to interesting results.

While we did not consider randomized algorithms here, the same analysis applies to expected utility of randomized algorithms. Though, it would not take advantage of the varied performance of a single algorithm on the same instance over multiple runs. Interestingly, randomized schedules do not increase the SPPA utility because the utility of a stochastic schedule is just a convex linear combination of the utilities of some deterministic schedules. One advantage of our approach is that it lends itself naturally to parallelization as long as the number of processors does not exceed the number of algorithms.

An interesting enhancement would be to incorporate an evaluation function that predicts the performance of each algorithm on each instance. Such prediction functions have been used effectively by [1, 8] in other contexts.

Acknowledgements

Support for this work was provided in part by the National Science Foundation (Grant No. 0328601) and by the Air Force Office of Scientific Research (Grant No. FA9550-05-1-0254). Any opinions, findings, conclusions or recommendations expressed in this manuscript are those of the authors and do not reflect the views of the US government.

References

[1] Andrew Arnt, Shlomo Zilberstein, and James Allen. Dynamic Composition of Information Retrieval Techniques. *Journal of Intelligent Information Systems*, 23(1):67–97, 2004.

[2] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2003.

[3] T. L. Dean. Intractability and Time-Dependent Planning. In *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, 1986.

[4] Luc Devroye, Laszlo Gyorfi, and Gabor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, 1996.

[5] Carla Gomes and Bart Selman. Algorithm Portolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[6] Reiner Horst and Hoang Tuy. *Global optimization: Deterministic approaches*. Springer, 1996.

[7] Henry Kautz and Bart Selman. Unifying SAT-based and Graph-based Planning. In *Proceedings of the 16th International joint Conference on Artificial Intelligence*, pages 318–325, 1999.

[8] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. Boosting as a Metaphor for Algorithm Design. In Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, pages 899–903, 2003.

[9] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Book Co, 1997.

[10] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[11] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Dover Publications, Inc, 1998.

[12] Marek Petrik. Learning Parallel Portfolios of Algorithms. Master's thesis, Comenius University, Bratislava, Slovakia, 2005.

[13] Laurent Simon. The Sat-Ex Site. Http://www.lri.fr/~simon/satex, 2005.

[14] Shlomo Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. Ph.D. Dissertation, University of California Berkley, 1993.

Appendix A. Proofs of Theorems

A.1 Proof of Theorem 5.2

Proof. We show that we can transform an arbitrary 0-1 knapsack algorithm to a scheduling problem instance in polynomial time. The main idea of the reduction is to create an instance for each object from the knapsack problem, and determine the subset T based on the classification of the instances to various algorithms. Notice that we do not assume that the algorithms are decision algorithms, instead they may have arbitrary resource utility functions. For the knapsack problem used in the reduction, we use the same notation as in Definition 5.1.

The scheduling problem is constructed as follows. Let the set of all instances be $I = \{x_i \mid i = 0, \ldots, N\}$ and the set of all algorithms $A = \{a_j \mid j = 0, \ldots, N\}$. Since the utility functions are homogeneous, the utility function for each algorithm a_j has the following form:

$$u_j(r, x) = \rho_j(r) * \eta_j(x).$$

Let the functions ρ_j be stepwise constant, with step-length ϵ. Then define an $\epsilon * w_j$ for each function such that it is constant for all $r_j > \epsilon * w_j$. Thus, we define ρ_j for all $j > 0$ as:

$$\begin{aligned} \rho_j(r) &= f\left(\left\lfloor \frac{r}{\epsilon} \right\rfloor\right) \quad \text{when } r < \epsilon * w_j, \\ \rho_j(r) &= f(w_j) \quad \text{otherwise,} \end{aligned}$$

where $f(z)$ is an arbitrary concave function. A possible example is $f(z) = 1 - \frac{1}{w}$. The resource utility function ρ_0 is defined as:

$$\begin{aligned} \rho_0(r) &= 0 \quad \text{when } r < \epsilon \\ \rho_0(r) &= 1 \quad \text{otherwise} \end{aligned}$$

As defined above, the domain of these functions is $[0, 1]$. We choose $\epsilon = \frac{1}{W+1}$.

For $i, j = 1, \ldots, N$, let the instance utility function be:

$$\begin{aligned} \eta_j(x_i) &= \frac{c_i}{f(w_i) - f(w_i - 1)} \text{ when } i = j \text{ and } j > 0 \\ \eta_j(x_i) &= 0 \text{ otherwise} \end{aligned}$$

And we also define the special case, for $i, j = 1, \ldots, N$

$$\begin{aligned} \eta_0(x_0) &> \max\{c_i \mid i = 1, \ldots, o\} \\ \eta_j(x_0) &= 0 \text{ for } j > 0 \\ \eta_0(x_i) &= \frac{c_i f(w_i - 1)}{f(w_i) - f(w_i - 1)} \end{aligned}$$

These values are well-defined because f is an increasing function.

The main idea behind the utility assignment is to ensure that assigning less that $w_i\epsilon$ resources for algorithm a_i does not influence the utility of the schedule, because the utility of a_0 will be greater on this instance. However, assigning at least $w_i\epsilon$ leads to an increase of the schedule utility by c_i. The fact that the function levels at $w_i\epsilon$ ensures that assigning more than that to an algorithm does not have any effect. The construction above is clearly polynomial. Now we proceed to prove that solving the above constructed scheduling problem solves the knapsack problem.

First, take a schedule s to the scheduling problem with utility $U(s)$, we show that there exists a subset T with

$$\sum_{i\in T} c_i = U(s) - \sum_{i=1}^{N} \eta_0(x_i) - \eta_0(x_0). \tag{6}$$

We can assume that at least ϵ is allocated to algorithm a_0. Otherwise, it would be possible to increase the value of the schedule by assigning it at least epsilon, because solving instance x_0 has higher utility than solving any other instance. Moreover, we can assume that all resource allocations are multiplies of ϵ, otherwise it is possible to just round it to the smaller value, since the utility on the interval does not change. Let A be the set of algorithms except a_0 that have at least one assigned instance. Clearly, total resources assigned to these algorithms do not exceed $W\epsilon$, and thus neither the total weight of objects corresponding to these instances exceeds W. The utility from these instances is equal to $K + \sum_{i\in A} z_i$, so clearly the cost of the corresponding objects is K.

Reversely, we show that for any knapsack set T there exists a schedule that satisfies (6). Take a subset of objects with weight at most W and cost at least C. By assigning w_i of the resources to each of the algorithms that correspond to the instances, these algorithms have a higher utility on them than a_0. Additionally, a_0 will have assigned ϵ of the processor time. Thus, the utility of the schedule is $\sum_{i=1}^{N} c_i + \sum_{i=1}^{o} z_i + Z$. This proves the theorem. □

A.2 Proof of Theorem 6.2

Proof. We prove the theorem in 4 steps. All steps, except the 3rd one are very similar to the proof of Theorem 12.4 (Glivenko-Cantelli) from [4].

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

$$\mathbf{P}\left[\sup_{s\in S} |U_m(s) - \mathbf{E}\left[U(s,X)\right]| > \epsilon\right] \leq 2\mathbf{P}\left[\sup_{s\in S} |U_m(s) - U'_m(s)| > \frac{\epsilon}{2}\right]. \tag{7}$$

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

$$\begin{aligned}
&\mathbf{P}\left[\sup_{s\in S}|U_m(s) - U'_m(s)| > \frac{\epsilon}{2}\right] \\
&\quad > \quad \mathbf{P}\left[|U_m(s^*) - U'_m(s^*)| > \frac{\epsilon}{2}\right] \\
&\quad > \quad \mathbf{P}\left[|U_m(s^*) - \mathbf{E}[U(s^*, X)]| > \epsilon, |U'_m(s^*) - \mathbf{E}[U(s^*, X)]| < \frac{\epsilon}{2}\right] \\
&\quad = \quad \mathbf{E}\left[\mathrm{I}\{|U_m(s^*) - \mathbf{E}[U(s^*, X)]| > \epsilon\}\,\mathbf{P}\left[|U'_m(s^*) - \mathbf{E}[U(s^*, X)]| < \frac{\epsilon}{2}\,\middle|\, X_1, \ldots, X_m\right]\right].
\end{aligned}$$

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

$$\begin{aligned}
&\mathbf{P}\left[|U'_m(s^*) - \mathbf{E}[U(s^*, X)]| < \frac{\epsilon}{2}\,\middle|\, X_1, \ldots, X_m\right] \\
&\quad \geq \quad 1 - \frac{\mathbf{E}[U(s^*, X)]\,(1 - \mathbf{E}[U(s^*, X)])}{\frac{m\epsilon}{2}} \\
&\quad \geq \quad 1 - \frac{\widehat{u}^2}{m\epsilon^2} \geq \frac{1}{2},
\end{aligned}$$

whenever $\frac{m*\epsilon^2}{\widehat{u}^2} > 2$. This shows (7).

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

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

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

$$\sup_{s\in S}\left|\sum_{i=1}^{m}(U(s, X_i) - U(s, X'_i))\right|$$

is the same as distribution of

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

Thus by STEP 1

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

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

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

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

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

Since, there are at most $(m+1)^{\binom{n}{2}}$ possible instance to algorithm classifications (Thm. 4.5) represented by W, we obtain

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

where s defines only the resource assignments, not the classification. Function k, and m_j are defined the same way as in Section 3. 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

$$\begin{aligned}
&(m+1)^{\binom{n}{2}}\sup_W\mathbf{P}\left[\sup_{s\in s}\left|\sum_{j=1}^{n}\rho_j(s[j])\sum_{i=1}^{m_j}\delta_{k(i,j)}\eta_j(X_{k(i,j)})\right|>\frac{\epsilon}{4}\,\middle|\,X_1,\ldots,X_m\right]\\
&\leq\ (m+1)^{\binom{n}{2}}\sup_W 2^n\sup_s\mathbf{P}\left[\left|\sum_{j=1}^{n}\rho_j(s[j])\sum_{i=1}^{m_j}\delta_{k(i,j)}\eta_j(X_{k(i,j)})\right|>\frac{\epsilon}{4}\,\middle|\,X_1,\ldots,X_m\right]\\
&\leq\ (m+1)^{\binom{n}{2}}2^n\mathbf{P}\left[\left|\sum_{j=1}^{n}\widehat{\eta}\widehat{\rho}\sum_{i=1}^{m}\delta_i\right|>\frac{\epsilon}{4}\,\middle|\,X_1,\ldots,X_m\right].
\end{aligned}$$

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. Then

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

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