Combining Multiple Heuristics*

Tzur Sayag[1], Shai Fine[2], and Yishay Mansour[3]

[1] School of Computer Science, Tel Aviv University, Tel Aviv, Israel. Email: tzurs@post.tau.ac.il
[2] IBM Research Laboratory in Haifa, Israel. Email: shai@il.ibm.com
[3] School of Computer Science, Tel Aviv University, Tel Aviv, Israel. Email: mansour@post.tau.ac.il

Abstract. In this work we introduce and study the question of combining multiple heuristics. Given a problem instance, each of the multiple heuristics is capable of computing the correct solution, but has a different cost. In our models the user executes multiple heuristics until one of them terminates with a solution. Given a set of problem instances, we show how to efficiently compute an optimal fixed schedule for a constant number of heuristics, and show that in general, the problem is computationally hard even to approximate (to within a constant factor). We also discuss a probabilistic configuration, in which the problem instances are drawn from some unknown fixed distribution, and show how to compute a near optimal schedule for this setup.

1 Introduction

Many important optimization and decision problems are computationally intractable and are sometimes even hard to approximate. The computational aspect of these problems is often depressing from the theoretical perspective. In reality however, in many cases, some of these hard problems are central and require a solution. Given that an efficient algorithm for these problems is (probably) out of the question, practical research is devoted to the discovery of heuristic methods which would optimistically be efficient over a large variety of problem instances.

One example of the above is the constraint satisfaction problem (SAT), which is highly important in the field of hardware and software verification. The fact that in general it admits no efficient algorithm has not stopped people from routinely solving impressively large instances of SAT problems. Quite to the contrary, the research community has come up with a large variety of heuristic methods for solving SAT problems (see [10,9,6] for example). A live proof of the efforts and interest in these problems is the fact that there are even organized competitions between the different academic and industrial implementations (see http://www.satcompetition.org/). One major difficulty concerning SAT solvers as well as heuristic methods for other problems is that the running time over a particular problem instance cannot be guaranteed. For this reason, once a

* This research was supported on part by an IBM faculty award.

heuristic consumes too much resources (e.g., time or space) one might suspend it and start running a different heuristic instead. The main goal of our model is to abstract this switching between different heuristics for solving the same instance of the problem.

Our model includes k distinct heuristics. Given a problem instance each heuristic has a different cost for solving it (for simplicity, the cost can be thought as the time needed by the heuristic to compute a solution). The cost of solving an instance of a problem is the total cost invested until one of the heuristics terminates with a solution. If we had only one problem instance then it would clearly be optimal to simply run the best heuristic (assuming it was known). We however, are interested in a case where we have n different problem instances, and the same schedule is used for all the problem instances. The total cost of a set of problem instances is the sum of the costs of the various problem instances. As a solution, we study two types of fixed schedulers. The first is a *resource sharing* scheduler that devotes a fixed fraction of its resources to each heuristic until some heuristic terminates with a solution. The second is a *task switching* scheduler in which only one heuristic is executed in each time unit. In both models, the schedulers are kept fixed for all problem instances, and the optimization problem is to find an optimal fixed scheduler for a given set of problem instances (and their related costs).

Our main result is an algorithm that computes an optimal resource sharing schedule in time $O(n \log n)$ for two heuristics (e.g., $k = 2$) and $O(n^{k-1})$ for a constant $k \geq 3$. For computing an optimal task switching schedule we briefly report an $O(n^{k+1})$ time algorithm. Note that both algorithms run in polynomial time algorithm for a constant number of heuristics. We also show that in general finding the optimal resource sharing schedule is NP-hard even to approximate. To make our setup more realistic, we study a probabilistic model in which the problem instances are drawn from some fixed unknown distribution. In this setup, one can search for a near optimal scheduler by sampling a number of problem instances and finding an optimal scheduler for them (in such case, the assumption that the cost of every heuristics over each problem instance in the sample is known in advance, becomes reasonable). Conceptually, in the initial learning phase we invest in learning the cost of each heuristic and the goal is to select a near optimal fixed scheduler. For the probabilistic model we show that if the number of problem instances (i.e., n) is a large enough sample, then, with high probability, the performance of any scheduler on the sample is close to its expected cost on the distribution (simultaneously for all schedulers). We note that since the number of schedulers might be infinite, we employ tools from computational learning theory to ensure the uniform convergence of the best scheduler on the sample to the optimal scheduler for the (unknown) distribution.

Our scheduling models have a similar flavor to the classical job scheduling models [11,8]. The problem instances can be viewed as different jobs while the heuristics can be viewed as different machines. Since each pair of heuristic and problem instance have some arbitrary cost, the setup resembles the parallel unrelated machines model of job scheduling. The main conceptual difference lies in the optimization function, which as we will see, is not linear, and hence, introduces new computational challenges.

Another related research topic is competing with the best expert in an online setting (e.g., [3]). In that setting one competes with the best single expert, which is in our case the best single heuristic. One should note that the performance of the best single heuristic can be significantly inferior even with respect to a round robin task switching (or equal weight resource sharing) schedule. This is since each heuristic might have a problem instance on which it performs very poorly. We note that both of our scheduling models do compare, implicitly, to single best heuristic (by simply assigning all the weight to one heuristic). However, we would like to stress that our scheduling models encompass a much wider class of schedulers, which is especially beneficial for cases in which problem instances have significantly different processing costs by various heuristics.

The rest of this paper is organized as follows. Section 2 introduces the model and the two scheduling schemes. The algorithm for the resource sharing scheduling problem appears in Section 3 in which we also state the hardness of approximation result. Section 4 introduces and discusses the probabilistic setup. We conclude with some empirical results in Section 5. Due to space limitations the proofs are deferred to the full version of the paper.

2 Model

We consider computing solutions to a set J of n problem instances using k different heuristics h_j, for $j \in [1, k]$. The cost (e.g., time) of solving the problem instance x_i using heuristic h_j is $\tau_{i,j}$, therefore, the input to the problem is modelled by a *processing cost matrix* $\mathcal{T} = [\tau_{i,j}] \in (\mathbb{R}^+)^{n\times k}$.

The *resource sharing* model encapsulates a scenario in which we execute all the heuristics concurrently, while devoting to each heuristic a fixed fraction of the resources. A *resource sharing* scheduler has a vector of shares $s \in \triangle_k$, where $\triangle_k = \{(s_1, \ldots, s_k) : \sum_{i=1}^{k} s_i = 1; s_i \in [0, 1]\}$. Conceptually, s_i is the fraction of resources we devote to execute heuristic h_i. Assume that heuristic h_i has cost τ_i, this implies that if we invest a total cost of $c = \tau_i/s_i$ then heuristic h_i receives τ_i resources and terminates with a solution. For the global cost we consider the first heuristic that terminates, which for a given problem instance x and schedule s we call the *completing heuristic*. Formally, the cost of a resource sharing schedule s for a problem instance x with costs $\tau = (\tau_1, \ldots, \tau_k)$ is given by

$$cost_{rs}(x, s) := \min_{1\le j\le k} \frac{\tau_j}{s_j}$$

and the completing heuristic is $\arg\min_{1\le j\le k} \frac{\tau_j}{s_j}$.

The cost of a resource sharing schedule s for a set of problem instances J is $\sum_{x\in J} cost_{rs}(x, s)$. The optimization task is, given a processing cost matrix $\mathcal{T} \in (\mathbb{R}^+)^{n\times k}$, compute a schedule s^* of minimum cost, i.e., $cost_{rs}(\mathcal{T}, s^*) = \min_{s\in\triangle_k} cost_{rs}(\mathcal{T}, s)$.

The *task switching* model divides the resources (e.g., time) to units and at each unit assigns all the resources to a single heuristic. (We assume here that all the costs are also integral number of units.) A *task switching* scheduler for k heuristics is a function $W(t) : \mathbb{N} \mapsto [1, k]$. The cost of a task switching scheduler W for a problem instance x with processing costs $\mathcal{T} = (\tau_1, \ldots \tau_k)$ is defined recursively,

$$cost_{ts}^{W}(x,t) = \begin{cases} 1 + cost_{ts}^{W}(\mathcal{T} - e_{W(t)}, t+1), & \forall i, \tau_i > 0 \\ 0, & \text{otherwise.} \end{cases}$$

where $\boldsymbol{e}_1, \ldots \boldsymbol{e}_k$ be the standard unit vectors in $\mathbb{R}^k$. In other words, we count the total number of resource units until one of the heuristics, say h_i, was executed at least τ_i times and thus terminates with a solution. Again, the cost for a set of problem instances J with processing cost matrix $\mathcal{T}$ is the sum of costs, i.e., $cost_{ts}(\mathcal{T}, W) = \sum_{x \in J} cost_{ts}^{W}(x, 0)$.

To see the difference between the performance of the different schedulers, consider the following processing cost matrix:

$$\mathcal{T} = \begin{pmatrix} 2 & 10 \\ 10 & 1 \end{pmatrix}.$$

The best single heuristic has cost 11. The optimal resource sharing model is not the trivial $(0.5, 0.5)$ but rather $(2 - \sqrt{2}, \sqrt{2} - 1)$ which has cost of about 5.8284. The optimal task switching schedule has cost 4.
The following lemma relates the optimal cost of the two models,

Lemma 1. *Let s be a resource sharing schedule, for any processing cost matrix $\mathcal{T}$ there is a task switching schedule W such that $cost_{ts}(\mathcal{T}, W) \leq cost_{rs}(\mathcal{T}, s)$*

We can also compare the performance of the above fixed scheduling schemes with the optimal dynamic scheme that selects the best heuristic for each problem instance, i.e., $best(\mathcal{T}) = \sum_{i=1}^{n} \min_{k \geq j \geq 1} \{\tau_{i,j}\}$.

Theorem 2. *Let $s^* = s^*(\mathcal{T})$ ($W^* = W^*(\mathcal{T})$, respectively), be an optimal resource sharing (task switching) scheduler for the processing cost matrix $\mathcal{T}$. The following bounds hold: (1) For any $\mathcal{T}$ we have $\frac{cost_{ts}(\mathcal{T}, W^*)}{best(\mathcal{T})} \leq \frac{cost_{rs}(\mathcal{T}, s^*)}{best(\mathcal{T})} \leq k$. (2) There is a matrix $\mathcal{T}$ such that $\frac{cost_{rs}(\mathcal{T}, s^*)}{best(\mathcal{T})} \geq \frac{cost_{ts}(\mathcal{T}, W^*)}{best(\mathcal{T})} \geq \frac{k+1}{2}$.*

3 Resource Sharing Scheduler

In this section we present an algorithm for the resource sharing scheduling problem, whose performance is stated in the following theorem,

Theorem 3. *Given a set of problem instances J and $\mathcal{T}$, the related processing cost matrix, an optimal resource sharing schedule can be computed in time[4] $O(n \log n)$ for $k = 2$, and time $O(n^{k-1})$ for a constant $k \geq 3$*

[4] We remark that our "O" notation has a constant that depends exponentially on the number of heuristics k.

Algorithm overview: Consider a set of problem instances J. Fix a resource sharing schedule s, and observe that for each $x \in J$, the minimum cost for x is attained on one of the k heuristics, i.e., the completing heuristic. The key idea of the algorithm is to divide the space of possible resource sharing schedules into disjoint cells with the following special property. In each such cell c, for each problem instance $x \in J$ and for any schedule $s \in c$, the same heuristic achieves the minimum cost, i.e., for every $x \in J$ the completing heuristic is the same for every $s \in c$. This implies that we can rewrite the total cost function for any schedule $s \in c$ as $cost_{rs}(\mathcal{T}, s) = \frac{A_1}{s_1} + \frac{A_2}{s_2} + \ldots + \frac{A_k}{s_k}$, where $A_1, \ldots, A_k$ are constants which depend only on the cell c. The main benefit is that we can efficiently minimize such a function over $s \in c$. This suggests that an influential complexity term would be the number of cells that we need in order to cover the entire space of resource sharing schedules.

For the resource sharing schedules, we show that the cells are obtained from hyperplane arrangements, for which there is a well established theory that allows us to bound the number of cells and their complexity. In addition, it allows us an efficient traversal of the cells.

Our arguments are organized as follows: we first introduce some basic facts about hyperplane arrangements, which are the main theoretical tool that we use. We then introduce a specific hyperplane arrangement we use to analyze our cost function. Finally, we describe the algorithm that finds an optimal schedule s^* and state its time complexity.

Hyperplane Arrangements: Hyperplane arrangements are well studied objects in computational geometry (see [4,7,5]). Informally, given a finite collection H of hyperplanes in $\mathbb{R}^d$, the hyperplanes in H induce a decomposition of $\mathbb{R}^d$ into connected open cells. We denote the arrangement of H by $\mathcal{A}(H)$. We now briefly review some definitions and basic results regarding hyperplanes arrangements. A *hyperplane* $H \subset \mathbb{R}^d$ is defined by an equation of the form $L(x) = b$ where $L : \mathbb{R}^d \to \mathbb{R}$ is a linear functional and b is a constant. A *hyperplane arrangement* $\mathcal{A}$ is a collection of hyperplanes in $\mathbb{R}^d$. Such a collection defines *cells* or *regions* (sometimes also called *faces*), namely the connected components of the set $\mathbb{R}^d - \cup_{H \in \mathcal{A}} H$. More formally, let $H_1, H_2, \ldots H_n$ be the set of hyperplane in $\mathcal{A}$. For a point p define

$$u_i(p) =: \begin{cases} +1, \text{ if } p \in H_i^+; \\ 0, \text{ if } p \in H_i; \\ -1, \text{ if } p \in H_i^-, \end{cases}$$

for $1 \leq i \leq n$. The vector $u(p) = (u_1(p), u_2(p), \ldots u_n(p))$ is called the position vector of p. If $u(p) = u(q)$, then we say that the points p and q are *equivalent*, and the equivalence classes thus defined are called the *cells* of the arrangement $\mathcal{A}$. A *d-dimensional cell* in the arrangement is a maximal connected region of $\mathbb{R}^d$ that is not intersected by any hyperplane in H. An *i-dimensional cell* in $\mathcal{A}(\mathcal{H})$, for $1 \leq i \leq d-1$ is a maximal connected region in the intersection of a subset of the hyperplanes in H, that is not intersected by any other hyperplane in H. Special names are used to denote i-cells for special i values: a 0-cell is called a

vertex, a 1-cell is an *edge*, $(d-1)$-cell is a facet and a d-cell is called a *cell*. For each (d-dimensional) cell c let $f_i(c)$ denote the number of i-dimensional cells on the boundary of c. We refer to $f(c)=\sum_{i=0}^{d-1} f_i(c)$ as the *complexity of the cell* c. The *complexity of an arrangement* $\mathcal{A}$ is defined as $\sum_{c\in\mathcal{A}} f(c)$.
The following results from [5] and [2] bound the hyperplane arrangement complexity.

Theorem 4 ([5]). *Let $\mathcal{A}$ be an arrangement of n hyperplanes in $\mathbb{R}^d$, for a constant $d\geq 1$. Then, the number of cells is $O(n^d)$, $\sum_{c\in\mathcal{A}} f(c)=O(n^d)$ and for every cell $c\in\mathcal{A}$ we have $f(c)=O(n^{\lfloor\frac{d}{2}\rfloor})$.*

Representing Hyperplane Arrangements: The common data structure that represents an arrangement is the incidence graph $I(\mathcal{A})$. (We briefly describe it and refer the interested reader to [5] for a more detailed exposition.) The incidence graph $I(\mathcal{A})$ contains a node $v(c)$ for each i-cell of $\mathcal{A}$; if two i-cells c and c' are incident upon each other then $v(c)$ and $v(c')$ are connected by an edge. When implemented, each node $v(c)$ of an incidence graph has a record that contains some auxiliary information (see [5]) among which is the coordinates of some point $p(v(c))$ inside the cell c and two lists containing pointers $L(v(c))$ to the sub-cells[5] and to the super-cells of c. (When convenient, we will use the term cell to represent the data structure $v(c)$.) Note, that the size of the representation of a cell $c\in\mathcal{A}$, is $O(f(c))$. The following theorem from [5] builds the incidence graph for a hyperplane arrangement.

Theorem 5 ([5], Theorem 7.6). *Let $\mathcal{A}$ be an arrangement of n hyperplanes in $\mathbb{R}^d$, for a constant $d\geq 2$. There is an algorithm that constructs the incidence graph $I(\mathcal{A})$ in time $O(n^d)$.*

Two Heuristics: Before we describe our algorithm for the general case of $k\geq 3$ heuristics, it is instructive to review the above definitions and the main steps of our algorithm for the simple case of two heuristics. Further, this case is of interest since it does not admit the same time complexity result we obtain later for $k\geq 3$, instead, we have an $O(n\log n)$ algorithm for this case.

Recall that for $k=2$ we have a processing cost matrix $\mathcal{T}\in(\mathbb{R}^+)^{n\times 2}$, and we seek to find a resource sharing schedule s^* such that $cost_{rs}(s^*)=min_{s\in[0,1]}\sum_{i=1}^{n}\min(\frac{\tau_{i,1}}{s},\frac{\tau_{i,2}}{1-s})$. (Note that for $k=2$ we have effectively only one parameter for the scheduler.) We describe our algorithm in terms that would be useful for the general case of $k\geq 3$. Our algorithm iterates the high dimensional cells of the arrangement one by one, minimizing the cost function for the cell and its sub-cells (we slightly abuse the notation here and use sub-cells to denote all cells of lower dimension that are on the boundary of c).

We now describe the above procedure in more detail (for the special case of $k=2$). Given a set of problem instances J and $\mathcal{T}$ the related processing

[5] A cell c' is said to be a *sub-cell* of another cell c if the dimension of c' is one less then the dimension of c and c' is contained in the boundary of c. Similarly, a cell c' is a *super-cell* of c if c is a *sub-cell* of c'.

cost matrix, we construct a hyperplane arrangement of dimension $k-1(=1)$ consisting of $n+2$ hyperplanes of dimension 0 (i.e., points). Our hyperplanes (i.e., the points) define where the cost of two heuristics is identical. Namely, the hyperplanes:

$$H_i : \frac{\tau_{i,1}}{s} = \frac{\tau_{i,2}}{1-s},$$

for $i = 1, \ldots n$. Additionally, we add the hyperplanes $s = 0$ and $s = 1$. The resulting hyperplane arrangement $\mathcal{A}(\mathcal{T})$ has two types of cells: at most $n+2$ points (which are the 0-cells) that lie on the segment $[0,1]$ and at most $n+1$ line segments (which are the 1-cells). It is easy to see that the line segments are exactly the cells we are after. Specifically, for each schedule s inside each such line segment or cell c, the minimal cost for each problem instance $x \in J$ is attained by the same heuristic. Our algorithm now needs to traverse the cells and write the cost function for each cell. Computing A_1 and A_2 for a specific cell directly requires to iterate through the problem instances and decide which of the two heuristics has the minimal cost (for all schedules inside c), we can then accumulate the relevant quantities to obtain A_1 and A_2 for the cell. This procedure takes $O(n)$ steps for each cell, and since there are $O(n)$ cells, this gives an $O(n^2)$ algorithm.

To improve the time complexity, observe that for schedules that are in the left most line segment $[0,a]$, for all problem instances the second heuristic is the completing heuristic. In the segment $[a,b]$, which is just to the right of $[0,a]$, only the problem instance corresponding to the point a changes its completing heuristic. Thus, given the cost function for $[0,a]$ we can use a simple update rule and in time $O(1)$ write the cost function for $[a,b]$. This method takes $O(n)$ steps for the first segment and only $O(1)$ for each of the remaining segments. To support this update scheme, we need a simple data structure that supports a sorted traversal of the cells. This adds a factor of $O(n \log n)$ time which turns out to be dominant in our algorithm.

Having shown how to write the cost function for each line segment, it is left to verify that the resulting functions can be easily minimized. Direct calculation shows that a local minimum for $\frac{A}{s} + \frac{B}{1-s}$ is attained at $\bar{s} = \sqrt{A}/(\sqrt{A}+\sqrt{B})$. This means that for each of the $O(n)$ segments, we consider at most three points as possible global minimum ($\bar{s}$ and the two end points – the subcells of the interval). The following lemma summarizes the above,

Lemma 6. *Given a processing cost matrix $\mathcal{T}$, an optimal resource sharing schedule can be found in time $O(n \log n)$ for $k = 2$.*

Hyperplane Arrangement for Resource Sharing: We now return to the general case of a constant $k \geq 3$ and show that the ideas used for $k = 2$ can be applied here as well. One difference is that instead of sorting the cells we use Theorem 5 and a special data structure that allows the cell traversal that is required for the efficient update of the cost function.

Given the processing cost matrix $\mathcal{T}$ we construct a hyperplane arrangement $\mathcal{A}(\mathcal{T})$ consisting of $\binom{k}{2} \cdot n = O(k^2 \cdot n)$ hyperplanes. We first consider the set of

$O(k^2 \cdot n)$ functionals indexed by $1 \leq i < j \leq k$ and $1 \leq r \leq n$. They are define by

$$L_{i,j,r}(s_1, \ldots, s_{k-1}) = \frac{s_i}{\tau_{r,i}} - \frac{s_j}{\tau_{r,j}}$$

Note that when $j = k$ we interpret s_k as $1 - s_1 - ... - s_{k-1}$. We consider the hyperplane $H_{i,j,r}$ in $\mathbb{R}^{k-1}$ defined by $L_{i,j,r}(x) = 0$. We also consider the hyperplanes $s_i = 0$ for $1 \leq i \leq k$ and the hyperplane $s_1 + ... + s_{k-1} = 1$. Thus the arrangement $\mathcal{A}$ consists of $N = \binom{k}{2}n + k + 1 = O(k^2 n)$ hyperplanes in $\mathbb{R}^{k-1}$.

The Cost Function in Each Cell: To motivate the choice of this particular hyperplane arrangement we study the cost function on each cell $c \in \mathcal{A}(\mathcal{T})$. The key observation is that for each schedule $s \in c$, exactly one unique heuristic has minimal cost for $x \in J$, which is true by definition of c. (If for some problem instance $x_r \in J$ there are two schedules $s_i, s_j \in c$ with different completing heuristics, then the hyperplane $L_{i,j,r}$ must intersect the cell.) The following lemma formalizes this argument.

Lemma 7. *In each cell c of the arrangement $\mathcal{A}(\mathcal{T})$ the function $cost_{rs}(\mathcal{T}, s)$ has the form $\frac{A_1}{s_1} + ... + \frac{A_k}{s_k}$. Moreover, given a cell c, we can obtain this representation (i.e., the constants $A_1, \ldots A_k$) in time $O(kn)$.*

To support a more efficient update of the cost functions for the cells, we define a *partition graph* $\mathcal{P}(\mathcal{A})$, which we use to efficiently compute the cost functions in the simplified form for the highest dimensional cells,

Definition 8. *For $I(\mathcal{A}(\mathcal{T}))$, the incidence graph of $\mathcal{A}(\mathcal{T})$, we define the* partition graph $P(\mathcal{A}(\mathcal{T})) = (V, E)$ where $V = \{v(c) \in I(\mathcal{A}(\mathcal{T})),\ \text{s.t. } dim(c) = k - 1\}$ (i.e., V has a vertex representing each highest dimensional cell in $\mathcal{A}(\mathcal{T})$), and an edge $(u, v) \in E$ if there is a path (u, w, v) in the incidence graph and $dim(w) = k - 2$ (i.e., two vertices $v(c), v(c')$ are connected in $\mathcal{P}(\mathcal{A}(\mathcal{T}))$ if there is a cell of dimension $k-2$ that is on the boundary of c and c' in the arrangement). As an auxiliary information, each vertex v in the partition graph $\mathcal{P}(\mathcal{A}(\mathcal{T}))$ points to the corresponding vertex in $I(\mathcal{A}(\mathcal{T}))$.

Lemma 9. *Given an incidence graph $I(\mathcal{A}(\mathcal{T}))$, (1) there is an algorithm that constructs its partition graph $\mathcal{P}(\mathcal{A}(\mathcal{T}))$ in time $O(n^{k-1})$ and, (2) $\mathcal{P}(\mathcal{A}(\mathcal{T}))$ is connected.*

Recall that for any cell Lemma 7 computes the values of $A_1, \ldots A_k$ in time $O(kn)$. To reduce the time complexity, we note that given these values for an initial $(k-1)$-cell, along with the partition graph, it is straight forward to compute these constants for a neighbor $(k-1)$-cell in $O(k)$ time. This implies that this computation, for all cells, can be done in time $O(k \cdot n^{k-1}) = O(n^{k-1})$. The idea is that we use the partition graph to update the cost function for a neighboring $(k-1)$-cell (i.e., a highest dimensional cell). Once the constants for a $(k-1)$-cell are known, we can compute the cost function in any subcell of $(k-1)$-cell using an algebraic manipulation taking advantage of the additional hyperplane equation that is satisfied on the boundary (the details are deferred to the full version of this paper).

Next, we show how to find the minimal value of the specific cost function in a given cell,

Observation 10. *Given an i-cell c and the constants $A_1, \ldots A_d$ of the cost function $cost_{rs}(\mathcal{T}, s) = \sum_j A_j/s_j$, there is an algorithm that rejects all schedules s inside c as possible minimal points of $cost_{rs}(\mathcal{T}, s)$ and works in time $O(f_{i-1}(c))$ where $f_{i-1}(c)$ denotes the number of faces on the boundary of c.*

Our algorithm visits each $(k-1)$ dimensional cell c (a vertex in the partition graph) and finds the minimal value of the specific cost function inside c and on its boundaries (which are sub-cells that are reachable from c in the incidence graph). The correctness follows from the fact that we visit all the $(k-1)$-cells, and test each of them along with its sub-cells.

Theorem 11. *There is an algorithm that finds an optimal resource sharing schedule in time $O(n^{k-1})$, for a constant $k \geq 3$.*

We also derive an algorithm to compute the optimal task switching schedule. Our algorithm uses a dynamic programming like approach, and the following theorem summarizes our result,

Theorem 12. *Given a set of problem instances J and $\mathcal{T}$, the related processing cost matrix, an optimal task switching schedule can be found in time $O(n^{k+1})$.*

Both our algorithms are polynomial only for a constant k. For an arbitrary k we show a hardness result. Namely, we show that finding a near optimal resource sharing schedule is NP-hard. First we formalize the related decision problem.

Definition 13 (RS). *The (decision version) of a resource sharing scheduling problem is a pair $(\mathcal{T}, C)$ where $\mathcal{T} = [\tau_{i,j}] \in \mathbb{R}^{n \times k}$ is a processing time matrix and $C \in \mathbb{R}$ is a goal cost. The question is whether there is a resource sharing scheduler $s \in \triangle_k$ with $cost_{rs}(\mathcal{T}, s) = \sum_{i=1}^{n} \min_{j=1}^{k}(\frac{\tau_{i,j}}{s_j}) \leq C$*

Using a reduction from 3-dimensional matching (3-DM) we show that the resource sharing scheduling problem is NP-hard to approximate.

Theorem 14. *RS is NP-complete, and there is a constant $\alpha > 1$ such that it is hard to approximate RS to within α, i.e., to find a schedule s such that $cost_{rs}(\mathcal{T}, s) \leq \alpha \cdot cost_{rs}(\mathcal{T}, s^*)$, where s^* is the optimal resource sharing schedule.*

4 Probabilistic Model

The main reason for studying a resource sharing or a task switching schedules is the fact that we hope to be able to fix such a schedule, based on a small number of problem instances, and expect it to behave similarly on future unseen problem instances. A crucial point is that once a schedule was fixed, the various processing costs of the problem instance are not needed in order to run the schedule. This methodology can be viewed as having a *learning phase* (in which we determine the schedule) and an *execution phase* (where we use the schedule). For the above methodology to work, we need that the performance in the learning phase and execution phase is similar, something which is not always guaranteed.

In this section we extend our scheduling models and describe a probabilistic model of generation of problem instances, this helps demonstrate the feasibility of the above methodology. Namely, first observing a small sample of problem instances (for each problem instance in the sample we have full information about all its processing costs), and then, computing an optimal schedule (resource sharing or task switching). The main contribution of this section is that we show that if the sample size is large enough then the observed cost on the sample is close to the expected cost on the distribution (simultaneously for all schedulers). The results and techniques that we use in this section are in the spirit of the classical uniform convergence results in computational learning theory (see [1]).

We first define the probabilistic model. Let $\mathcal{X}$ be the set of all possible problem instance and $\mathcal{H}$ be the set of all heuristics. For each pair of heuristic $h \in \mathcal{H}$ and problem instance $x \in \mathcal{X}$ let $\tau_{x,h} \in [1, B]$ be the cost of heuristic h on problem instance x, where B is an upper bound on the processing cost. (Note that the processing costs are arbitrary!)

Let Q be some fixed unknown probability measure on $\mathcal{X}$. When we draw a problem instance $x \in \mathcal{X}$, we are given an access to an oracle that given a heuristic $h \in \mathcal{H}$ returns $\tau_{x,h}$. Let $z = (x_1, \ldots x_m) \in \mathcal{X}^m$ be a *sample* of m problem instances identically distributed drawn independently from Q.

Let $\mathcal{W}$ be the set of all possible task switching schedules for k heuristics, i.e., $\mathcal{W} = \{W : W \in [1, kB] \mapsto [1, k]\}$ for problem instances in $\mathcal{X}$. (Note that any problem instance in $\mathcal{X}$ is completed after at most kB steps.)

Slightly abusing our previous notation, let $cost_{ts}(Q, W) = E_{x \sim Q}\left[cost_{ts}(W, x)\right]$ be the expected processing cost of W under the distribution Q. Similarly, let $cost_{ts}(z, W) = \frac{1}{m}\sum_{i=1}^{m} cost_{ts}(W, x_i)$ be the average cost of schedule W on the set of problem instances z. Finally, denote by $W^* = \arg\min_W cost_{ts}(Q, W)$ the optimal schedule with respect to the probability distribution Q and by $\hat{W}^*(z) = \arg\min_W cost_{ts}(z, W)$ the optimal task switching schedule on the set of problem instances z. Our main result is that with high probability, simultaneously for all schedules, the error by estimating the cost of a schedule on the sample is at most ε.

Theorem 15. *Let Q be a fixed distribution over $\mathcal{X}$ and $\epsilon, \delta > 0$ be fixed parameters. There exists a constant $m_0(\epsilon, \delta) = O(\frac{k^3 B^3}{\epsilon^2}\log(k/\delta))$, such that for any sample z of size $m \geq m_0(\epsilon, \delta)$, with probability at least $1 - \delta$ we have, $|cost_{ts}(z, W) - cost_{ts}(Q, W)| \leq \varepsilon$, for all $W \in \mathcal{W}$.*

An immediate implication of Theorem 15 is that computing an optimal task switching for a sample results in a near optimal task switching schedule. Note that our result has the accuracy (denoted by ε) converging to zero at rate of $O(\frac{1}{\sqrt{m}})$, as in the classical convergence bounds in computational learning theory [1]. We also show a similar result for the case of resource sharing scheduling,

Table 1. Resource Sharing vs Best Heuristics on SAT instances

Data Set	instances	#Variables	Best Heu.	2^{nd} Best	RS	Ratio
ADD	441	40,000	287,579,911	740,846,088	286,561,199	0.99
MADD	162	40,000	4,414,139,866	8,401,158,224	2,981,405,433	0.67
ADDI	281	1,500	1,178,621	1,178,621	1,178,621	1.00
MULI	265	$< 5{,}000$	541,765	541,765	541,765	1.00

Theorem 16. *For any fixed probability distribution Q on $\mathcal{X}$, any $\varepsilon, \delta \in (0,1)$, there is a constant $m_0 = \tilde{\Omega}(\frac{k^2B^2}{\varepsilon^2} \ln \frac{1}{\delta})$* [6] *such that with probability at least $1-\delta$, for any sample z of size $m > m_0(\varepsilon, \delta)$ we have, $|cost_{rs}(z, s) - cost_{rs}(Q, s)| \leq \varepsilon$, for every resource sharing schedule s.*

5 Empirical Results

The original motivation for this line of research was related to scheduling multiple SAT heuristics and improving over the performance of the best single heuristic. In the experiment design here we considered only two heuristics and tested on real verification data the performance of the optimal resource sharing schedule. Even in this simple case one can see the potential for improvements using our methodology. We note that these results are not obtained using the sampling probabilistic algorithm but instead we calculate the exact optimal resource sharing scheduler to emphasize the performance boost.

As input to our experiment we used real data that was generated to test the performance of several SAT solvers. Each data set contains several hundred problem instances that were generated for property testing of hardware circuits composed of various adders and multipliers (The data sets contain real SAT verification problems provided with the curtsey of IBM® research lab). In addition to the different type of circuits, each data sets is also different in the size of the problems (i.e. the number of variables) they contain. Since all these SAT solvers are DPLL based, we used the number of branch actions as the performance measure of a SAT solver on a problem instance (this measure has a very high correlation with real computation time and has the benefit of being machine independent).

To obtain these empirical results we picked the two best heuristics for each data set and computed the optimal resource sharing scheduler, we then compare its performance to these two best heuristics. Table 1 summarizes the performance of the optimal resource sharing compared to the top two heuristics. Clearly, the resource sharing algorithm is never worse than the best heuristic. In some cases it is significantly better (for the MADD set, the optimal resource sharing schedule has a cost of only 67% of the best heuristic, a very significant improvement).

[6] We use the notation $\tilde{\Omega}(f) = \Omega(f \log^c f)$, for some constant $c > 0$.

For the two small sets, the optimal resource sharing schedule is simply the best heuristic (which explains why the ratio is precisely 1), for the two larger data sets the improvements are significant.

References

1. M. Anthony and Peter L. Bartlett, *Neural network learning: Theoretical foundations*, Cambridge University Presss, 1999.
2. Boris Aronov, Jiri Matousek, and Micha Sharir, *On the sum of squares of cell complexities in hyperplane arrangements*, SCG '91: Proceedings of the seventh annual symposium on Computational geometry (New York, NY, USA), ACM Press, 1991, pp. 307–313.
3. Nicolò Cesa-Bianchi, Yoav Freund, David P. Helmbold, David Haussler, Robert E. Schapire, and Manfred K. Warmuth, *How to use expert advice*, Journal of the ACM **44** (1997), no. 3, 427–485.
4. Mark de Berg, Marc van Kreveld, Mark Overmars, and Ptfried Schwarzkopf, *Computational geometry algorithms and applications*, second, revised ed., ch. 8, pp. 172–180, Springer, 2000.
5. Herbert Edelsbrunner, *Algorithms in combinatorial geometry*, Springer-Verlag New York, Inc., New York, NY, USA, 1987.
6. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin, *Evaluating search heuristics and optimization techniques in propositional satisfiability*, Proceedings of International Joint Conference on Automated Reasoning (IJCAR'01) (Siena), June 2001, to appear.
7. D. Halperin, *Arrangements*, Handbook of Discrete and Computational Geometry (Jacob E. Goodman and Joseph O'Rourke, eds.), CRC Press LLC, Boca Raton, FL, 2004, pp. 529–562.
8. Dorit S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*, PWS Publishing Co., Boston, MA, USA, 1997.
9. O. Kullmann, *Heuristics for SAT algorithms: Searching for some foundations*, September 1998, 23 pages, updated September 1999 w.r.t. running times, url = http://cs-svr1.swan.ac.uk/ csoliver/heur2letter.ps.gz.
10. Chu-Min Li and Anbulagan, *Heuristics based on unit propagation for satisfiability problems*, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97) (Nagoya, Japan), August 23–29 1997, pp. 366–371.
11. M. Pinedo, *Scheduling theory, algorithms and systems*, Prentice-Hall International Series in Industrial and Systems EngineeringPrentice-Hall: Englewood Cliffs, NJ, 1995.