An automatic tuning system for NP-hard algorithms in a cloud context

Yanik Ngoko
Qarnot Computing and LIPN, University of Paris 13
Paris, France
yanik.ngoko@qarnot-computing.com

Denis Trystram, Valentin Reis
LIG, University Grenoble Alpes
Grenoble, France
denis.trystram@imag.fr

Christophe Cérin
LIPN, University of Paris 13
Paris, France
christophe.cerin@lipn.univ-paris13.fr

***Abstract*—Traditional automatic tuning systems are based on an exploration-exploitation tradeoff that consists of: learning the behavior of the algorithm to tune on several benchmark (exploration) and then using the learned behavior for solving new problem instances. On NP-hard algorithms, this vision is criticizable because of: the hardness of finding a reference benchmark and the potential huge runtime of the exploration phase. In this paper, we introduce QTuning, a new automatic tuning system specially designed for NP-hard algorithms. Like traditional tuning systems, QTuning is based on benchmark runs. But, during the learning process, new benchmark data can always be introduced or existing ones removed. Moreover, the system mixes the exploitation and exploitation phases. The main contribution of this paper is to formulate the learning process in QTuning within an active learning framework. The framework is based on a classical observation made in optimization: namely, the efficiency of random search in regret minimization. We improve our random search algorithm in including a machine learning classification approach and a set intersection problem. Finally, we discuss about the experimental evaluation of the framework for the resolution of the satisfiability problem.**

***Keywords*-Automatic tuning; Random search; Active learning; Maximum Subset Intersection Problem.**

I. Introduction

The global objective of this study is to improve the performance of cloud-services in learning their optimal configuration in an in-situ context. We assume a cloud-service made of configurable time-consuming algorithms like exponential algorithms or NP-hard algorithms. We also assume that the configurable algorithmic parameters are well identified and that there exists benchmarks for their calibration. In strictly using the cloud resources, our objective is to build an automatic tuning system that will find the configurations that optimize the cloud-service on runtime and/or energy consumption.

The core problem that we address is not new. It was addressed in several past work with dense and sparse linear algebra algorithms [1], [2], sorting [2] or Fast Fourier Transform [3] algorithms. Fundamental principles and techniques for automatic tuning have also been formulated in methods, systems and frameworks [4], [5], [6]. Despite these contributions, our work brings several novelties when considering the state-of-art of automatic tuning techniques.

Traditional automatic tuning systems are of two kinds: *offline and runtime tuning* [7]. In both classes, the tuning is built upon an exploration-exploitation trade-off where from a parametrizable algorithm $A(.)$ and a reference benchmark B (set of problem instances), one *learns* the optimal parameters values to use in $A(.)$ such as minimizing the runtime or energy consumption induced by its processing. After this exploration phase, the optimal parameter values are used for solving other input instances of $A(.)$ (exploitation phase). In the case of offline tuning, the exploration includes several runs of $A(.)$ on entries of B (for performance profiling) and produces an optimal configuration $\vec{\theta}^{opt}$ that is used for running $A(.)$. In runtime tuning, instead of an optimal configuration, the exploration generates a *decision function* f that on any input I of $A(.)$ returns the optimal configuration ($\vec{\theta}^{opt} = f(I)$) to use in its processing. Offline and runtime tunings have been successfully applied to many computational problems. However, we do believe that these approaches are not suitable to NP-complete problems.

Our first disagreement comes from the fact that such an exploration-exploitation trade-off *requires a reference or representative benchmark* that characterizes the *key features* of the inputs accepted by $A(.)$. Unfortunately, on NP-hard problems, the great diversity of problem instances makes it hard to reach a consensus on reference benchmark. International competitions like the annual SAT competition [1] or DIMACS challenge tend to establish such references; but, one can notice that their reference benchmarks change every year. Our second disagreement comes from the fact that these two tuning techniques assume a profiling stage where performance of algorithms are estimated on a reference benchmark. On NP-hard problems, this stage could take several days, months or years.

In the light of these observations, we argue that the first difference between our work and prior ones is to think automatic tuning in an alternative exploration-exploitation trade-off, more suitable for NP-hard problems. We formulate our alternative approach as the *active learning perspective for automatic tuning*. In this vision, the tuning is not made of two separated phases of exploration and exploitation. Both phases co-exist such that the tuned system evolves through-

[1] http://www.satcompetition.org/

out the time with new optimal or sub-optimal configurations, discovered in the processing. In addition, we reject the idea of a reference benchmark and prioritize the one of dynamic training sets that users can configure and modify, on the fly, during the tuning.

The second difference between our work and existing ones is that we consider functioning cloud-services to tune in-situ. This means that the datacenter resources on which we operate is not dedicated to tuning. Therefore, the exploration process will evolve in an agile setting where resources are not always available or execution faults could occur.

Summarizing, in this paper, we introduce QTuning, a new automatic tuning service that implements the active learning perspective for tuning, in-situ, in a cloud context. The paper contribution focuses on the general architecture of the system and its active learning framework. On this last point, we formulate the challenge in exploration as a bi-dimensional random search problem that we address in solving a classification problem and an NP-hard set intersection problem. Finally, we provide an experimental evaluation of our framework.

The remainder of the paper is organized as follows. In Section II, we explain the practical motivation of our work. In Section III, we discuss the related work. In Section IV, we introduce the key automatic tuning concepts that are manipulated in QTuning. The architecture of the system and its active learning framework is presented in Section V. In Section VI, we discuss about optimization algorithms implemented in the framework. A practical validation is presented in Section VII and we conclude in Section VIII.

II. Motivation

Our initial motivation for designing QTuning was to improve the *Qombinatorics* cloud service, available in the Qarnot Computing cloud [2]. Qombinatorics is a SaaS developed by the Qarnot Computing Research team for the parallel resolution of NP-hard problems.

The system is deployed in the Qarnot Computing cloud: a heating cloud for HPC that innovated in designing special radiators (Qrads) that produce heat from computations. The Qarnot cloud infrastructure consists of the network of radiators that are deployed at homes and are managed by a private resource manager (Qware). In an economic viewpoint, the Qarnot business is based on two types of users: hosts interested in heating and HPC users interested in computing. The objective is to use requirements in computations for supplying those in heating. However, it is obvious to notice that such a system is not always balanced. Typically, in winter, it can happen that we do not have enough computations for heating. For these situations, the QTuning system is particularly interesting. The idea is to push profiling jobs for learning the behavior of services implemented in the cloud when radiators are idle. The conclusion of the learning are next used to improve the functioning of the cloud-services on user requests.

[2]http://www.qarnot-computing.com/

III. Related work

As stated in the introduction, automatic tuning is a well-investigated field. For a general survey on HPC systems based on automatic tuning, the interested reader might take a look at the general report of the 2014 Dagstuhl seminary [8]. It is also important to notice that nowadays, automatic tuning, as well as *parallelism*, more than in the past are considered by industries as one of the major strategy for the improvement of algorithmic performance [9].

In order to position our study, we classify automatic tuning systems based on two criteria, namely their genericity and their learning framework.

A. Problem-specific vs class-of-problems tuning

By problem-specific, we refer to an automatic tuning approach that was developed for specific problems. In the class of problem specific approaches, some convincing examples are the ATLAS [1] library for matrix multiplication, OSKI [10] on sparse matrix vector multiplication, the SPIRAL library [2] for sorting and the FFTW [3] for Fast Fourier transform. These works demonstrated the efficiency of key automatic tuning techniques such as offline and runtime tuning or the dynamic programming approach for algorithms cascading.

At this stage, we highlight that in comparison to these prior works, the tuning system that we propose is not restricted to a specific problem or type of parameters. Instead, we consider general techniques that can be used in a wide range of NP-complete algorithms.

There are several systems or frameworks that were proposed for automatic tuning. The AEOS approach [11] summarizes a set of general principles and a component architecture that were successfully applied for tuning several dense linear algebra kernel. The OpenTuner framework formulates a new language for general automatic tuning [12]. The system also innovates in implementing ensemble techniques for combining several automatic search algorithms. As OpenTuner, QTuning invests on search techniques for tuning exploration. The Periscope framework [13] is another general solution for automatic tuning in a distributed context. Similarly, QTuning is based on distributed search. However, our objective lies in diversifying the quality of local solutions that can be found during the search, in an online manner. ParamILS [14] is an automatic tuning system especially designed for tuning hard computational algorithms. Our exploration algorithms used in QTuning are inspired from the random search algorithm defined in ParamILS. We innovate in formulating this search as a combination between breadth and depth first searches for optimizing a regret function.

B. From supervised/active learning to active collaborative filtering

Pre-existing work such as the MATE system [15], the tuned plugin [3] or the Green framework [16] already envision automatic tuning as a long-term process. While these works invest on the idea of improving a system via monitoring and/or supervised learning, we consider that we can leverage some additional computing power to search for the best parameters.

Search strategies for building models are addressed by the active learning [17] literature. Previous work [18] uses active learning to build a surrogate model for runtime and power consumption prediction conditioned on parameter values. However, when tuning the parameters of a heuristic solver for a NP-Hard problem, the best values may vary from one problem instance to another, as illustrated by the powerful SAT solving systems SATZilla [19]. Contrary to QTuning, SATZilla is problem specific, as it requires the use of a vector of features describing the specific problem instance.

One line of research that fits more closely into our direction is Active Collaborative Filtering [20] (ACF) where a learning system intends to determine the ratings (here, *runtime*) users (*problem instances*) would give to products (*parameters*) based on similar users. In ACF, the learning system can ask about user's ratings. However, this work does not focus on producing the best rated item for a given new user, and rather optimizes a loss on the whole model.

Finally, let us recall that one of the earlier general automatic tuning systems is the old Algorithm selection framework of Rice [4]. This work innovated in introducing the key concepts that are nowadays used in most automatic tuning system. Our work is largely inspired by the Rice framework.

In the next section, we will introduce the key modeling concepts that are manipulated in QTuning. The objective is to give more details about the active learning vision implemented in QTuning.

IV. Model

In this part, we formally describe the automatic tuning problem addressed in QTuning. We first describe the general concepts and then the problem formulation according to the active learning framework.

A. Basic concepts

In QTuning, automatic tuning problems are formulated within the notion of tuning project. Such a project can be constrained or unconstrained. We explain these notions below.

Definition 4.1: **(Unconstrained tuning project)** We define an unconstrained tuning project as a tuple $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}))$. Here, σ is the cloud service to tune; $\vec{\theta}$ are the parameters to tune and $d(\vec{\theta})$ is the set of definition domains for each parameter $\theta_i \in \vec{\theta}$; B is the initial set of training data that will be used for the tuning.

[3]http://linux.die.net/man/8/tuned

An unconstrained tuning project defines a corpus of data to use for learning the behavior of a cloud-service. Once such a project is submitted, the system will start a learning process for finding the *best parameters* for running σ. Any submitted project can be modified, on the fly during the learning process. Such a modification can consist of the addition of a new entry in B, the deletion of an entry and the deletion of a parameter from $\vec{\theta}$.

Let us assume that $\vec{\theta} = (\theta_1, \dots \theta_n)$. In QTuning, an unconstrained tuning project defines a tuning problem whose solution is a *configuration* $\bar{\theta}_u \in D$ where $D = d(\theta_1) \times \dots \times d(\theta_n)$. In order to be suitable to a large class of cloud-service, QTuning makes few assumptions on the structure of D. Consequently, the search space can be extremely large. In some cases however, users might want to introduce *particular knowledge* that could reduce the size of the search space. For this purpose, QTuning supports the concept of *tuning constraint* defined as follows.

Definition 4.2: **(Tuning search space constraint)** We define a constraint of the search space of a tuning project as a tuple $C = ((\theta_i, v_i), (\theta_j, v_j), \dots (\theta_k, v_k))$. The constraint states that we do not authorize the assignments in which θ_i is assigned to v_i, θ_j to v_j etc.

Constraints can be defined in any tuning project. This leads to a constrained tuning project that we define as a tuple $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}), C)$ where C is the set of constraints of the project. In the following, we will only consider constrained tuning project because they generalize unconstrained ones.

For characterizing the solution of a tuning project, let us assume that on each benchmark entry $\beta_i \in B$ and assignment of parameters $\bar{p}_u$ we have an estimation $\Delta(\sigma, \bar{p}_u, \beta_i)$ of the average runtime required for processing β_i by σ. Then we have the following definition.

Definition 4.3: **(Solution of a tuning project)** The solution of a tuning project $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}), C)$ is the configuration $\bar{\theta}^o$ such that

$$\bar{\theta}^o = \arg \min_{\{\bar{\theta} | C \text{ are not violated }\}} \frac{1}{|B|} \sum_{\beta_i \in B} \Delta(\sigma, \bar{\theta}, \beta_i)$$

In other words, the solution of a tuning project is the vector of parameters that leads to the best average cost in the processing of the benchmark instances. This definition restricts our objective to the tuning for average runtime optimization. But, the conclusion of our work can be generalized to alternatives optimization like makespan or energy consumption minimization.

The basic concepts we introduced formalize the automatic tuning problem as encountered in several studies. QTuning extend these traditional notions in assuming a particular formulation of the tuning problem.

B. The min-regret tuning problem

Once a project $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}), C)$ is submitted in QTuning at an initial date t_0, the cloud-service σ can at anytime t request the best local configuration that the system produced. This configuration will be used for processing σ in parallel to the learning stage. Moreover, the project submitted at the date t_0 could be changed at any date according to the actions we define in the prior section. Therefore, in searching for the best configuration, the QTuning system must try at anytime to return a solution that is not too far from the optimal one, according to the knowledge it has on tuning projects.

Given a project Γ, let us assume that at the date t, its setting is defined as $\Gamma(t)$. Let us also assume that the optimal solution of $\Gamma(t)$ is $\bar{\theta}^o(t)$. In the min-regret tuning problem, we are interested in minimizing the regret, defined as the difference between the average cost for processing $B(t)$ with $\bar{\theta}(t)$ and the one cost for processing $B(t)$ with the best possible configuration. We formalize the regret at the date t_n as:

$$\int_{t_0}^{t_n} \left(Cost(\bar{\theta}(t)|\Gamma(t)) - Cost(\bar{\theta}^o(t)|\Gamma(t)) \right) dt$$

where,

$$Cost(\bar{\theta}(t)|\Gamma(t)) = \frac{1}{|B(t)|} \sum_{\beta_i \in B(t)} \Delta(\sigma, \bar{\theta}, \beta_i)$$

In comparison to classical automatic tuning problems, the min-regret problem brings two novelties: the notion of evolving tuning projects and the regret minimization. We end here the modeling of the automatic tuning problem. In the next section, we will describe how QTuning addresses the min-regret problem in a system and algorithmic viewpoint.

V. The QTuning architecture

A. General view

In Figure 1, we describe the typical deployment architecture for which the QTuning system have been built. QTuning is composed of an API (*QTuning.dll*) and a server program (*QTuning.d*). The QTuning API includes a set of functions that launch remote executions on the servers. The functions can request: the loading of a new tuning project (submission of $\Gamma(t)$), the modification of an existing one or the downloading of analytic results about a tuning project (typically, the downloading of $\theta(t)$). The interest in implementing these functions in an API is to allow any cloud-service to adapt its behaviour on the fly. In Figure 1 for instance, the cloud *service.d* could decide to request the best local configuration before processing any user request. It can also make these requests according to a specific statistic law.

Let us now assume that *service.d* pushed a tuning project Γ in *QTuning.d*. Then, *QTuning.d* will start its learning process for finding the configurations that minimize the regret. For this purpose, *QTuning.d* will frequently create *exploration jobs* whose objectives are to run *service.d* on distinct configurations. Created jobs are submitted to the cloud resource manager (RMS in Figure 1) for a processing on cloud resources (In the case of Qarnot, they are Q.rads). Performance results about these jobs are next collected and save in a Mongodb database [4]. They will be used for computing the best local configurations that was found in the learning process.

In this architecture, we chose the Mongodb database manager because it is adapted to the storage of massive data. This is expected in our active learning perspective where the benchmark can be extremely large. One can also remark that the deployment of QTuning requires that it supports the cloud resource manager interface. In its current version, the service was developed for the Qware platform of Qarnot Computing; but we envision to include the API of other resource managers. In what follows, we will now give a deeper presentation of the QTuning learning framework.

B. QTuning active learning framework

The QTuning addresses the min-regret tuning problem with an *active learning framework* based on four concepts: data exploration algorithms, data exploitation algorithms, exploration notifications and exploitation notifications. These concepts fit together as follows. Let us consider a tuning project $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}), C)$. The submission of such a project in the QTuning system will automatically cause an exploration notification. This notification implies to start a data exploration algorithm. The goal of a data exploration algorithm is to compute the costs $\Delta(\sigma, \bar{\theta}, \beta_i)$ induced by the run of $\sigma(\bar{\theta})$ on the benchmark entry β_i. At a moment in the system, we could then have a set of evaluations $\Delta(\sigma, \bar{\theta_1}, \beta_0), \ldots \Delta(\sigma, \bar{\theta_k}, \beta_n)$. If the service σ requests for the best local configuration, this will cause an exploitation notification. Such a notification will start a data exploitation algorithms that will return the best local configuration that could be derived from $\Delta(\sigma, \bar{\theta_1}, \beta_0), \ldots \Delta(\sigma, \bar{\theta_k}, \beta_n)$.

In the above scenario, we presented exploration and exploitation notifications as being caused by a project submission or a cloud-service request. In QTuning, there are other situations where these notifications are launched; but, this is not discussed in this paper. Data exploration algorithms are naturally embarrassingly parallel. It suffices to concurrently evaluate the $\Delta(\sigma, \bar{\theta_u}, \beta_i)$. The challenging question that this observation leads to is the one of deciding on the resource set to assign to exploration. Indeed, the more we have resources for exploration the more we can expect to minimize the regret. Nonetheless, we must not forget that σ must also service user requests and for this, will need computing resources. In its current implementation, QTuning only uses

[4]www.mongodb.org

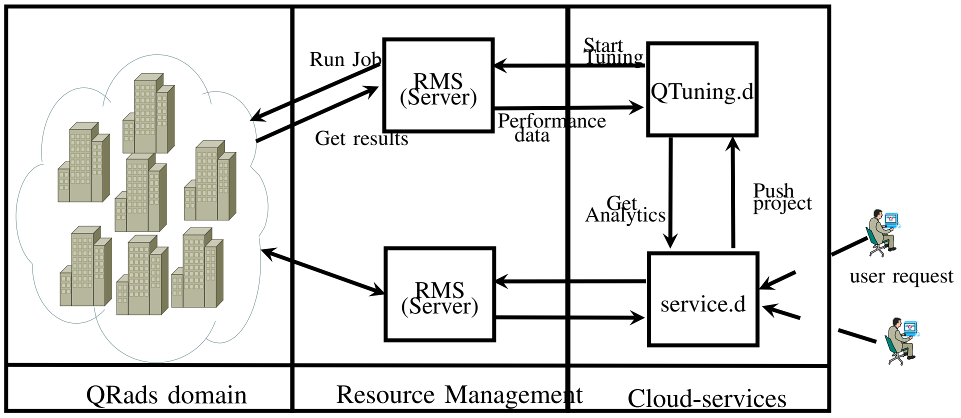


Figure 1: In-situ computing with QTuning

idle resources for exploration. In addition, the exploration could be interrupted at anytime if a user request is to be processed. Finally, a subset of resource is permanently allocated for data exploitation algorithms.

In Algorithm 1, we summarize the functioning of the active learning framework.

Algorithm 1 Active Learning framework

```
 1: while true do
 2:     ON_EXPLORATION_NOTIFICATION (e):
 3:     while e.exploration_continue() do
 4:         get the maximal number $max_J$ of resources available for exploration
 5:         get the maximal number $cur_J$ of resources used by exploration jobs
 6:         for all $j$ in $\{1, \ldots, max_J - cur_J\}$ do in parallel
 7:             Query an Oracle $O$ on the next configuration point $(\bar{\theta}, \beta)$ to evaluate
 8:             Submit an exploration job for $(\bar{\theta}, \beta)$ to the RMS
 9:             Wait for result
10:             Process and save the performance $\Delta(\sigma, \bar{\theta}, \beta)$ in the database
11:         end for
12:     end while
13:     ON_EXPLOITATION_NOTIFICATION (e):
14:     Retrieve the identifier $pid$ of the corresponding tuning project from $e$
15:     Load the performance data $\Delta(\sigma, \bar{\theta}_1, \beta_0), \ldots \Delta(\sigma, \bar{\theta}_k, \beta_n)$ of $pid$ from
        the database
16:     Process the analytic request formulated in $e$.
17:     return the result
18: end while
```

Algorithm 1 gives a simplified view of what is actually implemented. For instance, in the exploration phase, it does not handle the fact that QTuning actually manages several distinct tuning projects. Another limitation is that the execution of an exploration job could fail and this is supported in QTuning. However the presented view is enough for understanding the main principle of data exploration. On data exploitation, let us notice that until now, we mainly focused on a vision where exploitation consisted in retrieving the best local configuration. However, QTuning has a larger view. The idea is to process analytic requests that can consist in computing the standard deviation observed in a configuration or the percentage of the search space that was explored on a configuration. As for exploration, the exploitation scheme gives a simplified view of what is implemented. For example, QTuning generally anticipates requests on analytic results in computing them before (at line 10).

It might not be clear why we used the term active learning when considering the literature on the concept. Indeed, we can relate our framework to active learning algorithms studied in classification as follows. The *configuration points* $((\bar{\theta}, \beta))$ for which $\Delta(\sigma, \bar{\theta}, \beta)$ are our labelled data. The points without such an estimation are our unlabelled data. The best configuration is the classifier we are building. The objective in querying the oracle O is to state the *most important configuration point* that could serve for finding a configuration, close to the best one. Finally, as other active learning systems [17], we are motivated by the potential huge training time (estimation of $\Delta(\sigma, \bar{\theta}, \beta)$) and data to process.

We end here the general presentation of the QTuning framework. In the next sections, we will discuss in more details about data exploration and exploitation algorithms.

VI. Data exploration and exploitation in QTuning

This section aims at developing two points of the prior framework. The first is the different type of oracles that we can use in the exploration; the second one is how we compute the best configuration depending on the chosen oracle.

A. *The lexicographic oracle (Lex)*

The objective of QTuning oracles is to return the most important configuration point to evaluate. The lexicographic oracle works by ordering the configurations and benchmark according to the lexicographic order. The order is first applied on configurations and then on benchmark. For any query, it returns the next pair $((\bar{\theta}, \beta))$ in the lexicographic order that was not already evaluated.

For the lexicographic oracle, the computation of the best configuration (line 10 in the active learning framework) is simple: we cumulate the estimation made on a configurations and compare with the best result already found. This process is detailed in Algorithm 2.

Algorithm 2 Lexicographic Result Update

```
1: INPUT: A new result $\Delta(\sigma, \bar{\theta}_i, \beta_j)$, $\theta^o$ is the best local configuration and $\Delta(\theta^o)$ is its runtime on $B$.
2: Add $\Delta(\sigma, \bar{\theta}_i, \beta_j)$ to $\Delta(\theta_i)$
3: if $\Delta(\theta_i) > \Delta(\theta^o)$ then
4:     Mark all the pairs $(\theta_i, \beta_u)$ as explored
5: else
6:     if all the pairs $((\theta_i, \beta_u))$ where evaluated for $\beta_u \in B$ then
7:         Compare $\Delta(\theta^o)$ and $\Delta(\theta_i)$ and update the best local configuration
8:     end if
9: end if
```

As one can remark, the active learning framework with the lexicographic order leads to lexicographic exploration of the search space with the usage of a lower bound. This is a classical search approach that however is not optimized for the context we target. Indeed, for the minimization of the regret function, we need a oracle that takes care about the *informativeness* of the next configuration point: we do not have any element that suggests that the next lexicographic could be such a point.

B. Random oracle (Rand)

1) Random configuration choice and depth first search: Given a tuning project $\Gamma = (\sigma, B, \vec{\theta}, d(\vec{\theta}), C)$, whose search space is D, the random oracle considers that good configurations could be located *anywhere* in D. As the lexicographic oracle, the random oracle is based on ordering. In particular, it completely processes points of a configuration θ_i before starting those of a configuration $\theta_{i'}$. Differently from the lexicographic oracle, when for the current configuration θ_i we have all the $\Delta(\sigma, \bar{\theta}_i, \beta_j)$s, the next configuration $\bar{\theta_{i'}}$ from which we start is chosen randomly. In Figure 2, we graphically represent how this feature makes the random oracle different from the lexicographic one. With random iterations over configurations, we diversify the search walk. For regret minimization, we are convinced that this is better in the mean case. Let us now assume that the random oracle selected a θ_i for the generation of the next configuration points. A challenging question is to decide on the ordering of the first benchmark entries to evaluate. For example, must we start by the point (θ_i, β_0) or by the point (θ_i, β_1)?

2) Hard and soft instances: For deciding on the first benchmark entry to evaluate, we use an observation made on NP-hard problems. The idea is that instances of NP-hard problems are often classified as being hard or soft. The soft instances are the ones for which we can quickly find a good solution while hard instances are the ones for which algorithms runtime explode. Given a new configuration θ_i, we do believe that it will be better to start to evaluate firstly its hard instances. Indeed, if B_{hard} are those instances and θ_i a *weak* configuration, we can quickly fall in a setting where

$$\sum_{\beta \in B_{hard}} \Delta(\sigma, \bar{\theta}_i, \beta) > \Delta(\theta^o)$$

This means that in using the process of results updating described in Algorithm 2, we can quickly eliminate non-optimal configurations if we know hard instances.

Thus, given a configuration θ_i, the random starts by classifying the instances in hard and soft sets B_{hard} and B_{soft}. Then, it returns the configuration points with B_{hard} before those in B_{soft}. Within each class, the benchmark entries are selected randomly. This means that if $\beta_1, \beta_2 \in B_{hard}$ and $\beta_3 \in B_{soft}$, the oracle can return either (θ_i, β_1), (θ_i, β_2) (θ_i, β_3) or (θ_i, β_2), (θ_i, β_1) (θ_i, β_3). But we cannot have (θ_i, β_3), (θ_i, β_1) (θ_i, β_2). The only remaining question in this scheme is to be able to effectively build B_{hard} and B_{soft}. This is discussed in the next subsection.

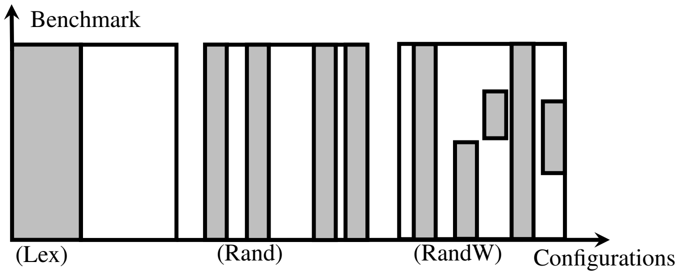


Figure 2: Lexicographic, random and random oracle with wisdom. *Lex* explores the search space in the lexicographic order. *Rand* and *RandW* randomly choose the next configuration. *RandW* differs from *Rand* on the fact that it extracts information from configurations that were partially explored.

3) A similarity approach for finding hard instances: The main assumption we made for detecting hard instances is that their *runtime will look similar on the configurations we processed*. Let us assume at the date t that we processed the set of configurations $\Theta(t)$. Let us also assume that there is a subset $\Theta_1(t) \subseteq \Theta(t)$ for which the evaluations were performed on all B [5]. We propose to characterize the runtime of each instance $\beta \in B$ as a vector

$$\Delta(\beta) = [\Delta(\sigma, \bar{\theta}'_0, \beta), \ldots \Delta(\sigma, \bar{\theta}'_{n_1}, \beta)]^T$$

where $n_1 = |\Theta_1(t)|$. Here, θ'_i are configurations in $\Theta_1(t)$ that we ordered randomly.

In other words, the runtime vector of a benchmark entry is the collection of runtime observed on $\Theta_1(t)$. Given these data, we propose to apply a $2-$Means clustering algorithm on the data B with the runtime vector and the euclidean distance. Finally, we conclude that hard instances are those, in the class, for which the mean value $\|\Delta(\beta)\|$ is the greatest. Indeed, it is intuitive to expect that on hard instances, most

[5]due to lower bound elimination we could strictly have $\Theta(t) \backslash \Theta_1(t) \neq \emptyset$

configurations take a huge runtime before finding the optimal solution.

Let us mention that the idea of distinguishing hard instances of NP-hard problem with a machine learning approach was also investigated in another paper [21]. But our approach completely differs from this proposition on the modeling of features. The *Rand* oracle uses the result update process of *Lex*. However, it is based on a deeper exploitation of the potential random distribution of best configurations. In addition, it exploits more past information generated in the exploration to take further decisions. But, this point could be improved as we will see in the sequel.

C. Random oracle with the wisdom of the losers (RandW)

With *Rand*, estimations made on $\bar{\theta}_0'$ could serve to guide the exploration of $\bar{\theta}_1'$, $\bar{\theta}_2'$ etc. This extraction of information is interesting, but there still remains a wastage. Indeed, data related to the *losing configurations* $\Theta_2(t) = \Theta(t) \backslash \Theta_1(t)$ are not exploited by the oracle [6]. This can be seen as a loss if we consider the processing cost that these evaluations required. The question then is: how could we exploit *this effort* for taking better decisions?

For this purpose, let us come back to the random oracle. At the beginning of the exploration of any configuration θ_i, the algorithm separates instances in hard and softs. Then, it randomly chooses instances in the hard class and then the ones of the soft class. This random choice could be improved in considering data of losing configurations. For example, being given the set hard instance B_{hard}, we could observe from data of the losers that a small subset B_{hard^+} of these data was generally enough to reach a lower bound violation. Then, instead of a random exploration of B_{hard}, it will be more efficient to randomly start by instances in B_{hard^+}. Globally, we can then reformulate the exploration of a configuration points as follows. We first start with the benchmark entries in $B_{hard^+} \cap B_{hard}$, then by the one in $B_{hard^+} \cap B_{soft}$ and finally the remaining instances in considering hard instances before soft ones. The question is: how to build B_{hard^+}?

1) Computation of hard instances as a subset intersection problem: We propose to consider B_{hard^+} as the set of benchmark entries for which there is a maximal number of losing configurations that were evaluated. The idea behind is that the more a same benchmark entry was evaluated in losing configurations, the more, we could believe that this benchmark entry causes lower bound violations.

More practically, at each date t, we assume that we know the size $K(t)$ of B_{hard^+}. For computing B_{hard^+}, we firstly associate each benchmark entry β with a configuration set $S(\beta)$ whose elements are all configurations $\bar{\theta} \in \Theta_2(t)$ for which there exists a computed estimation $\Delta(\sigma, \bar{\theta}, \beta)$. We then choose B_{hard^+} as the set of benchmark entries $\{\beta_1^+, \ldots, \beta_k^+\}$ such that $|\{\beta_1^+, \ldots, \beta_k^+\}| = K(t)$ and $|S(\beta_1^+) \cap \cdots \cap S(\beta_k^+)|$ is maximal.

With this formulation, we will decide on the construction of B_{hard^+} based on $|S(\beta_1^+) \cap \cdots \cap S(\beta_k^+)|.K(t)$ additional data. In the best case, when comparing with *Rand*, we will have $|\Theta_2(t)|.K(t)$ additional data that *we will transform in information*. This formulation has two challenges. The first one is that we need to tune $K(t)$. This is not discussed in this paper. The second challenge is the resolution of the intersection problem. We propose below a greedy approach.

2) Greedy algorithm for computing B_{hard^+}: One can easily notice that the computation of B_{hard^+}, as formulated, corresponds to the resolution of the Maximum K-intersection problem [22]. Indeed, this latter problem is defined as follows.

Given n sets $E_1, \ldots, E_m$ with elements over a universe $\Sigma = \{e_1, \ldots, e_n\}$, the goal is to select K subsets of $E_1, \ldots, E_m$ in order to maximize the size of the intersection of the sets. As shown in [22], this problem is NP-complete and cannot be approximated within an absolute error of $\frac{1}{2}n^{1-2\epsilon} + \frac{3}{8}n^{1-3\epsilon} - 1$ unless **P = NP**. However this intersection problem is close to coverage problems for which there exists a general heuristic framework. The first stage in this framework consists in ordering the sets $S(\beta)$ according to a pricing strategy. Then, the sets with higher prices are progressively chosen until we reach a subset intersection that cannot evolve. Using this idea, we propose in Algorithm 3 a greedy solution for building the B_{hard^+} set.

Algorithm 3 RandW Hard instance selection (RandW-HIS)

```
1: for $\beta \in \Theta_2(t)$ do
2:     Build the set $S(\beta)$.
3: end for
4: $C = \emptyset$; $Inter = \Theta_2(t)$
5: while $|C| < K(t)$ do
6:     Choose the set $S(\beta^+)$ s.t. $S(\beta^+)$ is not in $C$ and $|S(\beta^+) \cap Inter|$ is maximal;
7:     $C = C \cup S(\beta^+)$
8:     $Inter = Inter \cap S(\beta^+)$
9: end while
10: return $C$
```

We end here the description of the data exploration and exploitation in QTuning. An important point that we did not discuss is how we manage evolvability. For instance, what happens when between two dates t and $t+1$, we have $B(t) \neq B(t+1)$. We will not cover this point in this text. But, we can retain that in these cases, each oracle reconsiders the project $\Gamma(t+1)$ as a new project for which, we already have the estimates $\Delta(\sigma, \bar{\theta}, \beta)$ computed in $\Gamma(t)$ The next section is devoted to experiments.

VII. Experimental evaluation

A. Settings

In considering the architecture of Figure 1, we simulated the behavior of the active learning framework in a setting

[6]Losing configurations are those on which we did not entirely evaluated B because we detected before a lower bound violation.

where given a tuning project pushed at date 0, *service.d* requests a new configuration to *QTuning.d* every d seconds. We assume that between these d seconds, the active learning framework was able to launch exploration jobs and get the results on p configuration points. For obtaining the p points, the framework could have used the oracles above.

- *lex:* it corresponds to the *Lex* oracle.
- *rand:* the *Rand* oracle;
- *rand_sort:* the *Rand* oracle; but the group of hard and soft instances are each sorted in ascending order on the value of $\|\Delta(\beta)\|$;
- *rand_desc_sort:* the *Rand* oracle; but each group of hard and soft instance is sorted in descending order on the value of $\|\Delta(\beta)\|$;
- *rand_w:* the *RandW* oracle;
- *rand_w_noh:* the *RandW* oracle; but we choose B_{hard} randomly instead of using the 2-Means clustering.

For evaluating the oracles, we assumed three tuning projects based on the data of an international SAT competition [7]. In this competition, various solvers are evaluated on instances of the NP-hard problem SAT. The benchmark instances B and the configurations Θ of each tuning project are associated with a sub-competition. In each sub-competition, we fixed the values $\Delta(\sigma, \bar{\theta}_i, \beta_j)$ as the runtime of a SAT solver on a benchmark instance. Let us notice that all these runtime are available in the competition web site. Finally, the chosen sub-competitions are: Core solvers sequential on random satisfiable+unsatisfiable instances (c-SAT+UNSAT), Core solvers sequential on satisfiable instances (c-SAT) and Core solvers sequential on hard satisfiable+unsatisfiable instances (c-SAT+UNSAT—Hard).

B. Experimental plan

We did 7 series of experiments. In each, we computed the regret of the active learning framework for each of the oracles in a discrete windows of time $t_0, \ldots, t_n$. For the first 3 experiments, all oracles were evaluated assuming data of three sub-competitions. The evaluation for each competition included 300 sub-evaluations, were we changed the ordering of instances B and configurations Θ. The objective in these first experiments was to find the best oracle. In experiment 4, we compared variants of the *RandW* with different values of $K(t)$. Finally, in experiments, 5, 6 and 7, we compared all the oracles when the values of p are changed. The experiments plan are summarized in Table I

Exp.1	competition=c-SAT+UNSAT ($\|B\| = 300$, $\|\Theta\| = 28$), $d = 600$, $p = 100$, $t_0 = 0$, $t_n = 76.d$, $K(t) = 10$
Exp.2	competition=c-SAT ($\|B\| = 300$, $\|\Theta\| = 31$), $d = 600$, $p = 100$, $t_0 = 0$, $t_n = 45.d$, $K(t) = 10$
Exp.3	competition=c-SAT+UNSAT—Hard ($\|B\| = 300$, $\|\Theta\| = 31$), $d = 600$, $p = 100$, $t_0 = 0$, $t_n = 92.d$, $K(t) = 10$
Exp.4	competition=c-SAT+UNSAT, $d = 600$, $p = 100$, $t_0 = 0$, $t_n = 76.d$, $K(t) = 10, 20, 30, 40$
Exp.5	competition=c-SAT+UNSAT, $d = 600$, $p = 120$, $t_0 = 0$, $t_n = 76.d$, $K(t) = 10$
Exp.6	competition=c-SAT+UNSAT, $d = 600$, $p = 140$, $t_0 = 0$, $t_n = 76.d$, $K(t) = 10$
Exp.7	competition=c-SAT+UNSAT, $d = 600$, $p = 160$, $t_0 = 0$, $t_n = 76.d$, $K(t) = 10$

Table I: Experimental plan

C. Experimental results and analysis

In Figure 3, we present the mean regret we obtained. The first lesson we learn is that on Exp.1, Exp.2, Exp.3, the best oracle is *rand_desc_sort* and the worst is *lex*. Indeed, this oracle leads to the minimal regret within the time. In the plots in Figure 3, one can notice that the regret starts to grow quickly and then stabilizes. This is normal since within the time, we are approaching the optimal solution. These first experiments confirm two noticeable points: the interest of randomization for regret minimization (*lex* is the worst oracle) and the interest in processing instances with hard ones before soft ones instances.

The oracle *rand_desc_sort* was either followed by *rand*, or by *rand_w* and its variant. This means that there is an interest in using the wisdom of the loser. Since in the first experiments, we set $K(t)$ to 10, a question was then to see whether or not we could obtain better results for *wisdom of loser oracles* in changing the $K(t)$. The answer for this question is yes. As shown in Exp.4 there is an effective performance variation when $K(t)$ is changed. Consequently, for further work it is interesting to invest on how to decide the best value of $K(t)$. Finally, in the figures 3(e-f), we present what we obtained in changing the number of configuration points that we processed between the d seconds. The results confirm the trend we observed in the three first experiments.

VIII. Conclusion

In this paper, we introduced a new automatic tuning system for NP-hard algorithms in a cloud context. The proposed system is based on an active learning framework where we continuously learn the best configuration while servicing cloud requests with the best local results found. We proposed a naive oracle for the learning process and several improvements based on randomization, k-means clustering and the resolution of a set intersection problem. We then simulate the performance of our oracles based on runtime of algorithms solving the satisfiability problem. The experiments clearly demonstrated that our improved oracles are better in the practice. They also reveal the dominance of some oracles.

For continuing this work, we envision three main perspectives. The first one is to combine the different oracles in a general active learning framework. Here, we could

[7]http://satcompetition.org/edacc/SATCompetition2013/

use for example multi-armed bandit models. Our second perspective is to refine the classification of hard instances such as to define better techniques for ordering configuration points. Indeed, the experiments showed that the usage of sorting or the wisdom of the losers can improve the *Rand* oracle. Finally, we intend to define *filling techniques* such as to predict the value of configuration points with few processing. We do believe that given an instance, from the trace execution of a set of configurations on it, we can estimate the runtime it will have on new configuration.

References

[1] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimization of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[2] X. Li, M. J. Garzarán, and D. A. Padua, "A dynamically tuned sorting library," in *proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Jose, California: IEEE Computer Society, June 2004, pp. 111–124.

[3] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *proceedings of the International Conference on Acoustics, Speech and Signal Processing*. Seattle, Washington: ACM SIGARC, May 1998.

[4] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[5] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11.

[6] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "Siblingrivalry: Online autotuning through local competitions," in *International Conference on Compilers Architecture and Synthesis for Embedded Systems*, Tampere, Finland, Oct 2012.

[7] W. Tichy, "Auto-tuning parallel software: An interview with thomas fahringer: the multicore transformation (ubiquity symposium)," *Ubiquity*, vol. 2014, no. June, pp. 5:1–5:9, Jun. 2014.

[8] S. Benkner, F. Franchetti, H. M. Gerndt, and J. K. Hollingsworth, "Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401)," *Dagstuhl Reports*, vol. 3, no. 9, pp. 214–244, 2014.

[9] Oracle-Database, "Performance tuning guide," http://docs.oracle.com/cd/B28359_01/server.111/b28274.pdf, July 2008.

[10] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 16, 2005, pp. 521–530.

[11] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293 –312, feb. 2005.

[12] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe, "Opentuner: an extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, 2014, pp. 303–316.

[13] R. Mijakovic, A. P. Soto, I. A. C. Ureña, M. Gerndt, A. Sikora, and E. César, "Specification of periscope tuning framework plugins," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, 2013, pp. 123–132.

[14] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Int. Res.*, vol. 36, no. 1, pp. 267–306, Sep. 2009.

[15] A. Morajko, A. Martínez, E. César, T. Margalef, and J. Sorribes, "MATE: toward scalable automated and dynamic performance tuning environment," in *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II*, 2010, pp. 430–440.

[16] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *SIGPLAN Not.*, vol. 45, no. 6, pp. 198–209, Jun. 2010.

[17] B. Settles, *Active Learning*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. [Online]. Available: http://dx.doi.org/10.2200/S00429ED1V01Y201207AIM018

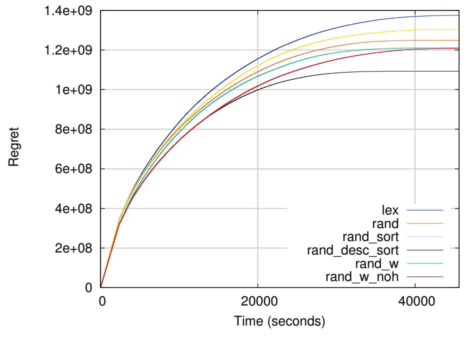
[18] P. Balaprakash, R. B. Gramacy, and S. M. Wild, "Active-learning-based surrogate models for empirical performance tuning," in *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, 2013, pp. 1–8.

[19] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of Artificial Intelligence Research*, pp. 565–606, 2008.

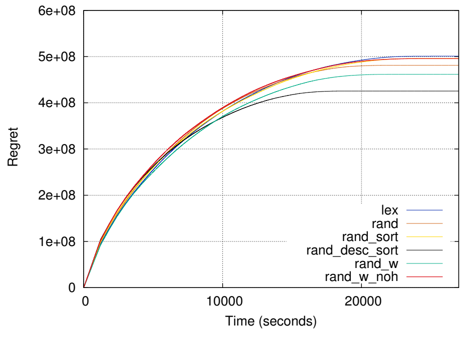
[20] C. Boutilier, R. S. Zemel, and B. M. Marlin, "Active collaborative filtering," *CoRR*, vol. abs/1212.2442, 2012. [Online]. Available: http://arxiv.org/abs/1212.2442

[21] T. Z. Zhe Wang and Y. Zhang, "Distinguishing hard instances of an np-hard problem using machine learning," http://cs229.stanford.edu/proj2013/WangZhangZhang-DistinguishHardInstancesOfAnNPHardProblem.pdf, 2013, online; Stanford University; accessed 8 october 2015.

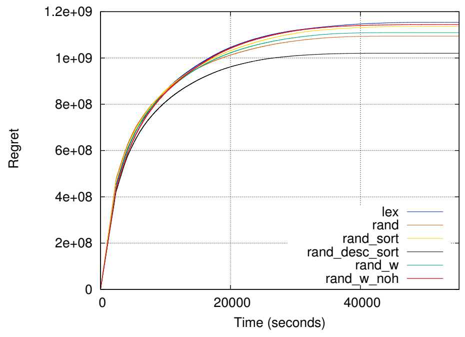
[22] M. Shieh, S. Tsai, and M. Yang, "On the inapproximability of maximum intersection problems," *Inf. Process. Lett.*, vol. 112, no. 19, pp. 723–727, 2012.



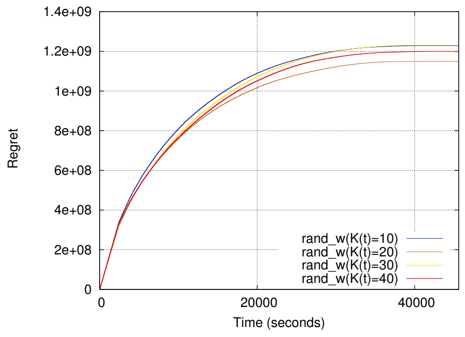
(a) Exp.1



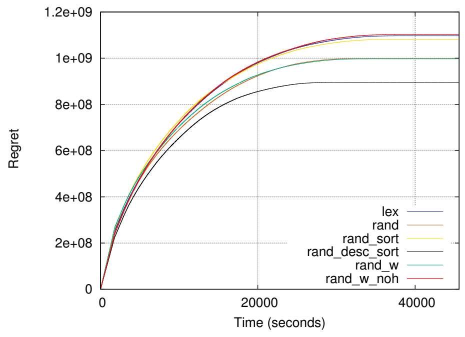
(b) Exp.2



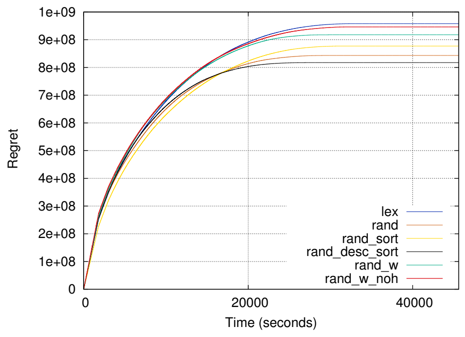
(c) Exp.3



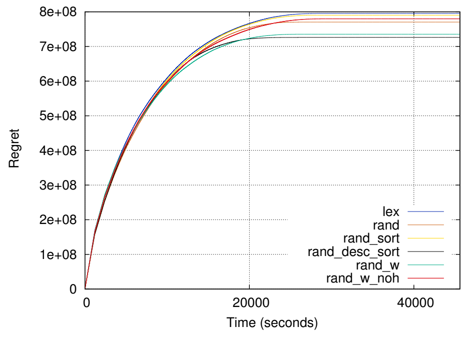
(d) Exp.4



(e) Exp.5



(f) Exp.6



(g) Exp.7

Figure 3: Mean regret obtained from the experiments whose settings are in Table I