Int. J. Appl. Math. Comput. Sci., 2019, Vol. 29, No. 2, 261–274
DOI: 10.2478/amcs-2019-0019

amcs

SOLVING SAT IN A DISTRIBUTED CLOUD: A PORTFOLIO APPROACH

YANIK NGOKO [a,b,*], CHRISTOPHE CÉRIN [b], DENIS TRYSTRAM [c]

[a]Department of Research and Development
Qarnot Computing, 40/42 Rue Barbés, Montrouge, France
e-mail: yanik.ngoko@qarnot-computing.com

[b]Department of Computer Science
University of Paris 13, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
e-mail: christophe.cerin@lipn.univ-paris13.fr

[c]Department of Computer Science
University of Grenoble Alpes, 700 avenue Centrale, 38401 Saint Martin, d'Héres cedex, France
e-mail: denis.trystram@imag.fr

We introduce a new parallel and distributed algorithm for the solution of the satisfiability problem. It is based on an algorithm portfolio and is intended to be used for servicing requests in a distributed cloud. The core of our contribution is the modeling of the optimal resource sharing schedule in parallel executions and the proposition of heuristics for its approximation. For this purpose, we reformulate a computational problem introduced in a prior work. The main assumption is that it is possible to learn optimal resource sharing from traces collected on past executions on a representative set of instances. We show that the learning can be formalized as a set coverage problem. Then we propose to solve it by approximation and dynamic programming algorithms based on classical greedy algorithms for the maximum coverage problem. Finally, we conduct an experimental evaluation for comparing the performance of the various algorithms proposed. The results show that some algorithms become more competitive if we intend to determine the trade-off between their quality and the runtime required for their computation.

Keywords: resource provisioning and scheduling, parallel distributed SAT, algorithm portfolio, maximum coverage problem.

1. Introduction

This paper deals with the parallelization of the classical satisfiability problem (SAT) (Kautz and Selman, 2007). Given a propositional formula defined as a conjunction of clauses, the objective in SAT is to state whether or not the formula is satisfiable. We are interested in a distributed solution that can be deployed in any volunteer computing-like framework such as the heating cloud[1] of the Qarnot computing company.[2]

SAT is a highly investigated NP-hard problem; it has several applications in combinatorial equivalence checking, automatic test generation, model checking, cryptography, etc. Since we are increasingly reaching the limitations of computability (see the work of Vardi (2014) for the end of Moore's law), we believe it is essential to develop new techniques for solving fundamental problems of computing such as SAT.

Several parallel solvers have been proposed for SAT. As mentioned in prior works (Martins *et al.*, 2012; Audemard *et al.*, 2014), they can be grouped into two classes: divide-and-conquer and portfolio based solvers. In the former the parallel algorithm operates by partitioning the search space of the formula to evaluate among several concurrent workers. GradSAT (Chrabakh and Wolski, 2003) and Psato (Zhang *et al.*, 1996) are some solvers that are based on this mechanism. In contrast, in portfolio solvers, the same search space can be assigned to concurrent workers. The particularity is that each worker

*Corresponding author

[1]A heating cloud is the one whose datacenters are units of a district heat network. See the work of Ngoko *et al.* (2018) for other examples.

[2]www.qarnot.com.

will use a different search strategy for finding the solution. At first sight, the lack of search space partitioning could be considered a weak point since we might have *redundant exploration* of the search space. But, let us observe that several portfolio solvers like Ppfolio (Roussel, 2011) or Penelope (Audemard *et al*., 2012) were awarded at the SAT competition[3] and sometimes they outperformed divide-and-conquer solvers.

In this paper, we focus on parallel portfolio approaches. The motivation for this choice is that, in the reference distributed cloud we target (the one of Qarnot), communications between nodes could be too slow (Internet-based architecture). Fortunately, in portfolio approaches, we can formulate efficient parallel algorithms that require scarce communication between the concurrent executions.

There exist two classes of parallel portfolio solvers for SAT. The first one is that of *pure portfolio* solvers, based on independent concurrent *sub-solvers*. This is the case of Ppfolio or those based on the more advanced resource sharing schedules studied by Goldman *et al*. (2012). A run of such solvers consists in concurrent execution of several SAT algorithms. The concurrent SAT algorithms do not *share knowledge* during their execution. An interruption signal is sent to each algorithm once a solution has been found in an algorithm. The second class of portfolio approaches is based on knowledge sharing. Solvers like Penelope or Plingeling (Biere, 2010) are based on sophisticated mechanisms, inspired by the conflict-driven clause learning (CDCL) technique (Silva and Sakallah, 1996), for sharing knowledge between sub-solvers. Unfortunately, such solutions are not necessarily suitable for distributed clouds in which the communication time could be important. Thus, the solver proposed in this paper is a pure portfolio one.

There exist distributed portfolio solvers that are close to what we propose in this work (Audemard *et al*., 2014; Goldman *et al*., 2012). Compared with them, we mainly contribute in three original points. Firstly, we introduce a new hierarchical scheme for the portfolio resolution of SAT in a cloud context. The parallel scheme is built to run in a cloud-service deployed in a volunteer-like cloud. The scheme includes two portfolio levels. At the upper level, the scheme consists of portfolio execution of parallel solvers that are run on cloud nodes. This level also includes a *planning engine* for deciding on the resource sharing schedule and an *execution engine* that deploys the solvers on a cloud and watches their execution to capture and react to messages, faults or errors. The parallel solvers of the upper level are based on an algorithm portfolio. More precisely, at the lower level, a parallel solver execution consists of the run of several distinct sequential SAT solvers in threads. Parallel solvers' executions are done in containers. It is important to note that we consider a cloud context where the resources are not strictly dedicated to our parallel scheme. Requests to a general cloud dispatcher must be done to deploy parallel SAT solvers.

Our parallel scheme can be seen as a pure portfolio of parallel solvers, run in clouds. The intelligence of our parallelization is in the modeling of optimal resource sharing to use between parallel solvers.[4] The optimal resource sharing schedule in our model is obtained by solving an NP-hard coverage problem similar to those introduced by Goldman *et al*. (2012). As in that work, the main assumption in the coverage problem is that we can learn the optimal schedule from traces collected in running solvers on a representative set of instances. However, while Goldman *et al*. (2012) propose to optimize resource sharing based on the average or maximal runtime of the representative instances, in this paper we propose to maximize the number of instances that are solved within a time limit.

Our second contribution is to propose algorithms for computing *good* resource sharing schedules in our parallel scheme. The proposed algorithms are based on the greedy resolution of the maximum coverage problem. In some cases, we provide theoretical guarantees on their quality. Finally, we evaluate the different algorithms we propose for the computation of optimal resource sharing. The experiments demonstrate that our algorithms largely overpass naive solutions. In addition, some of them are more competitive if we want to make a trade-off between the quality of results and the runtime.

As mentioned before, there are several parallel solvers proposed for SAT. A survey of the contributions can be found in the works of Holldobler *et al*. (2011) and Martins *et al*. (2012). Here, we focus on parallel distributed algorithms.

The Dolius solver (Audemard *et al*., 2014) is a parallel solver close to our target. It is based on a master/slave model and combines the divide-and-conquer and portfolio parallelism. The master starts its execution in dividing the SAT formula and sending it to workers. These later will process the formula with potentially different types of SAT solvers (portfolio). The parallel formulation we propose is close to the Dolius model. However, we differ in two aspects: firstly, we do not apply divide-and-conquer between workers to avoid communication costs that this will lead to. The workers in our model solve the same problem. Secondly, the portfolio construction in our solution is based on a new optimization model for resource sharing.

The remainder of this paper is organized as follows. Section 2 develops our parallel scheme for SAT. The

[3] www.satcompetition.org.

[4] This modeling is the main difference compared with other pure portfolio solvers like PPfolio.

presentation includes the Qarnot cloud architecture. In Section 4, we discuss algorithms for computing approximated or near-optimal resource sharing schedules. Section 5 is devoted to an experimental evaluation, and we conclude in Section 6.

2. Architecture

The Qarnot computing architecture is an Internet computing architecture whose main computing nodes are Q.rads. A Q.rad is a heater ($\simeq$ 500–700 W) whose heat sink consists of embedded microprocessors (in general 3–4 Intel i7 or AMD Pro-Rizen). Totally silent and based on free cooling, Q.rads belong to the category of *data furnace servers* disrupting volunteer computing, distributed cloud computing and edge computing (Ngoko *et al.*, 2018).

Qarnot developed Q.ware, a distributed cloud computing middleware that operates on top of Q.rads. Q.ware distinguishes two types of customers: those interested in heating (hosts of Q.rads) and those interested in distributed cloud computing (Internet customers). The main goal of Q.ware is to use the distributed computing requests for providing heat, where necessary, through Q.rads. One challenge in Q.ware is that the arrival law of heating requests will not necessarily follow the demand in computing. To cope with this difficulty, Q.ware uses several mechanisms, including the outsourcing to third party platforms, the reduction of Q.rad power consumption or the run of scientific/cryptocurrency mining workloads (Ngoko *et al.*, 2016) (when there are not enough computations).

Qsat was designed as a cloud-service for Q.ware. To understand its functioning, it is important to have basic notions regarding Q.ware. In this section, we briefly introduce the Q.ware general processing architecture. Then, we explain how Qsat interacts in this architecture.

2.1. Q.ware architecture. Going into details, Q.ware implements a decentralized computing architecture made of geo-distributed *compute clusters* connected to the Internet. The middleware was designed for batch and bare-metal processing with containers (mainly Docker). In Fig. 1, we describe an abstract view of the main components of a compute cluster.

In such a cluster, a distributed process will generally be triggered by a request sent by an Internet customer. The request can be of two types. Firstly, there are general computing requests; they ask to deploy a container or a set of containers that are not related to any Q.ware service. Here, we will typically have public Docker containers. General computing requests are routed towards a REST API server. The second type of requests covers those that are related to a cloud-service supported by Q.ware (in Fig. 1, service.d denotes such a service). An example of a cloud-service in Q.ware is the Qarnot 3D rendering service. Such requests are sent to the front-end server of the related cloud-service.

Whatever the request, its processing (at REST API or service.d) will consist of (i) creating a set of computational tasks and (ii) putting these tasks in the scheduling queue of the Q.node of the compute clusters in which we are. Here a task denotes an abstraction that mainly refers to a container environment,[5] a set of input files[6] and a command line to be executed on the input files and in the environment.

The Q.node uses a scheduling agent to dispatch tasks on Q.rads. For this purpose, the agent needs to have the state of Q.rads in order to elect nodes that are more suitable to receive computations (those where there is a heating demand). The states of Q.rads are aggregated through Q.boxes. These are servers that are typically deployed in buildings where there are Q.rads. In general, there is one Q.box per building and the Q.boxes are connected to Q.rads through a local network. With the states of Q.rads, the scheduling agent will use a greedy principle to assign tasks to Q.rads. Such an assignment will imply transferring the data of the task (environment, input files) to the Q.boxes where the processing will be done.

As already said, in the Qarnot platform, *Service.d* will typically (but not only) correspond to the 3D rendering service.[7] In our former work, we proposed to implement a tuning engine for SAT following this architecture (Ngoko *et al.*, 2016). Next, we explain how to implement service.d as a cloud-service for the distributed resolution of SAT.

2.2. Overview of Qsat. The Qsat service consists of two main components: a library and a server. The former is to be used in the client part (Internet customer). It provides functions for sending SAT computing requests in an appropriate format. The SAT server includes a front-end and a back-end for the processing of SAT requests.

Requests sent to Qsat are received by the front-end process. The front-end analyzes the request and, if there is no anomaly, it puts it in the queue of the back-end process. The back-end frequently monitors its queue to check whether or not there are new data. If that is the case, it uses a first come first served policy to process the data. The processing of each request consists of (i) creating a set of computing tasks for SAT processing, (ii) submitting the tasks to the scheduling agent and, finally, (iii) monitoring the execution. Our paper will not discuss the Qsat library or the front-end server. Instead, we will focus on the processing performed at the back-end level.

[5]The repository of the container image is defined by the customer.
[6]The input files will generally be uploaded to a CEPH storage.
[7]`https://render.qarnot.com/`.

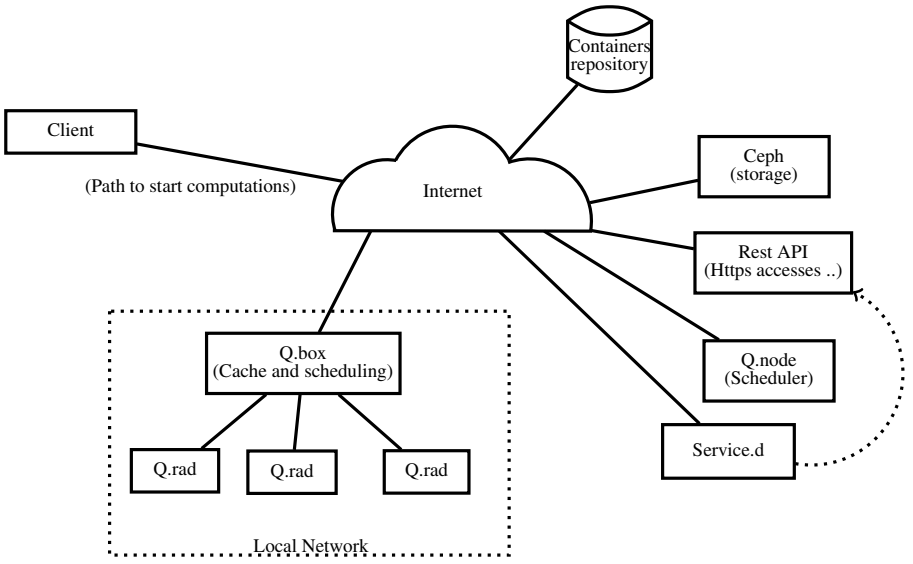


Fig. 1. Main components in the processing of a requests by Q.ware.

2.3. Back-end processing. Back-end processing is based on two engines: a planning and an execution one. While the goal of the former is to decide on a plan for solving SAT requests, that of the latter is to execute the plan. A request received by the planning engine can be defined as the triple $r = (\phi, m, T)$. Here, ϕ is a CNF SAT formula (Jackson and Sheridan, 2005), m—the maximal number of computing nodes to use in the processing and T—a timeout after which the processing will be interrupted.

For servicing r, the planning engine, which will compute a plan that consists of a set of tasks to deploy within the time. The tasks specified in the plan are based on a specific container image designed for Qsat. When creating the plan, the planning engine does not consider the real number of available computing nodes. It assumes that there are m computing nodes that are available.

Once the plans are computed, they are sent to the execution engine, which will create and submit tasks to the Q.node (following the plan). The execution engine will also frequently collect results from the deployed tasks. After a maximal processing time of T units on compute nodes, the result will be returned to the user.

The interest in defining a number of compute nodes and timeout is that users pay per compute nodes and time utilization. In the next section, we will provide a deeper description of a plan.

2.4. Qsat plans. We define a plan as a special version of a malleable resource sharing schedule (Goldman *et al.*, 2012). Below, we define such schedules.

Definition 1. (*Resource sharing schedule*) Consider a computing machine with m homogeneous computing resources. Let us also assume a set $\mathcal{H}$ of k parallel heuristics solving the same problem. We define a resource sharing schedule as a tuple $S = (S_1, \ldots, S_k)$ such that

$$\sum_{i=1}^{k} S_k \leq m.$$

Resource sharing schedules are used to combine several heuristics solving the same problem (parallel portfolio). Given a computational instance I to be solved and a resource sharing schedule S, I is solved by a concurrent run where each heuristic h_i is started on I with S_i computing units. All the executions are halted as soon as a heuristic finds a solution.

Definition 2. (*Malleable resource sharing schedule*) Consider a computing machine with m homogeneous computing resources. Let us also assume a set $\mathcal{H}$ of k parallel heuristics. We define a Q-phase, malleable resource sharing schedule as a tuple

$$\Gamma = [(S^1, \tau_1), \ldots, (S^Q, \tau_Q)],$$

where each S^i is a resource sharing schedule and $\tau_1 \geq 0$ is a time duration.

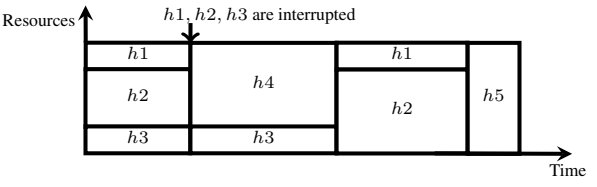


Fig. 2. Example of a malleable resource sharing schedule.

Q-phase malleable resource sharing is used to combine several resource sharing schedules for the solution of a problem. At date 0, we start the execution of the resource sharing schedule S^1. If in the interval $[0, \tau_1]$ a solution is found, then the execution is halted. Otherwise, one continues with the execution of the resource sharing schedule S^2. If we do not have a solution at date τ_Q, then we state that there is no solution.

In Fig. 2, we illustrate the run of a four-phase schedule that combines 5 heuristics on m computing units. In a malleable resource sharing schedule, each couple (S^u, τ_u) defines a phase. As one can notice, the same heuristic can be be used in different phases (its number of resources is not null in these phases). A question then is to know if we will continue the execution. This will be discussed further; for now, we can assume that we will generally restart heuristics between phases.

A phase is characterized by two values: a starting date and an ending date. The former is the date at which we can start the run of any solver of the first. The latter is the date at which there is at least one solver of the phase that is interrupted. In the schedule $\Gamma = [(S^1, \tau_1), \ldots, (S^Q, \tau_Q)]$, the starting date of the u-th phase is $\sum_{t=1}^{u-1} \tau_t$ and the ending date is $\sum_{t=1}^{u} \tau_t$.

A Qsat plan is a malleable portfolio schedule. But the inverse is not true. Indeed, in Qsat plans, we consider the following additional rules.

1. We assume that the computing units consist of multi-core processors.

2. We define $\mathcal{H}$ as a set of multi-core CDCL solvers. The solvers are based on a parametric version of the Penelope solver (Audemard *et al.*, 2012). In the classical version, for 8 threads, Penelope uses a specific selection of sequential SAT solvers that are concurrently parallelized with OpenMP directives. In the parametric version we created, one can dynamically define the selection of SAT solvers to be used. In addition, we created a Docker image for this version in order to be able to run it on the Qarnot cloud.

3. On each computing unit, we strictly assign at most one solver. This means that $S_u \in \{0, 1\}$.

4. We assume an input deadline T that defines the timeout for the processing of any instance. This means that
$$\sum_{q=1}^{Q} \tau_q \leq T.$$

Equipped with these definitions, we can summarize the processing in Qsat as follows. Given a request $r = (\phi, m, T)$, the planning engine will compute a schedule $\Gamma = [(S^1, \tau_1), \ldots, (S^Q, \tau_Q)]$ such that

$$\sum_{q=1}^{Q} \tau_q \leq T$$

and

$$\forall S^u, \sum_{i=1}^{k} S_i^u \leq m.$$

The schedule of Fig. 2 is not a Qsat plan. This is because some heuristics have more than one computing unit. In Fig. 3, we illustrate a simplified process view of the resolution of a SAT instance. Here, we assume that we have a 1-phase schedule and $m = 3$ computing units.

A more complex plan is considered in Fig. 4(a). This plan describes a concurrent execution of 9 parallel CDCL solvers (P_1–P_9) within the time. Given this schedule, the execution engine will proceed as follows. Firstly, it will create three tasks, each corresponding to the CDCL solvers P_1, P_3, P_5, and we will submit them to the scheduling agent. Three containers will be deployed to run the CDCL solvers. After a maximal duration of t_1, the execution engine will interrupt the three running tasks. If before this date a solver finds a solution, the others will be interrupted; otherwise, the execution engine will wait for the deadline t_1. If no solution is found, the execution engine will next create and submit three other tasks, each corresponding to the run of P_2, P_4 and P_6. These tasks will run until a deadline set to $t_2 - t + 1$. If we still do not have a result, the process will be repeated for the last three sets of portfolio solvers. Once a solver finds a solution or after the timeout, the execution engine collects the results and returns them.

As already stated, the CDCL solver we consider corresponds to a parametric version of the Penelope solver, and is a multithreaded portfolio solver for SAT. Consequently, there are two levels of concurrency in a Qsat plan. The upper level, where we have a concurrent execution of parallel solvers, and the lower, where there are multithreaded executions of SAT solvers.

It is important to note that, due to the overhead induced by resource management, the execution of the resource sharing model described in Fig. 4(a) may correspond to what is depicted in Fig. 4(b). This is due to the fact that the scheduling agent must find free nodes before deploying submitted tasks. An important issue for the execution engine is, given t_1, to decide on the effective date t_1' at which it will submit a task for the execution of P_2. In our implementation, we propose to adopt an opportunistic rule: *As soon as the execution of a parallel solver P_i is ended, the execution engine submits the solver P_i' that follows P_i according to the ordering defined in the resource sharing schedule*. In doing so, we do not apply the deadline t_1 to a phase but, individually, to the solvers. Thus, solvers of different phases can run

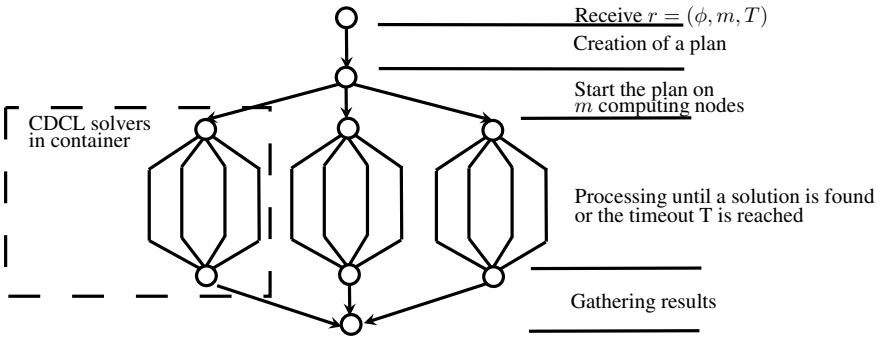
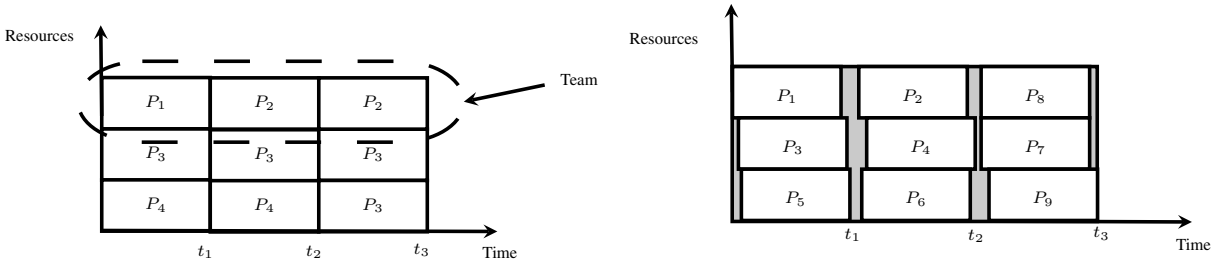


Fig. 3. Simplified execution graph of Qsat with 3 processors (4 cores each).



(a) theoretical resource sharing schedule (b) practical execution

Fig. 4. Example of resource sharing schedules with three 9 parallel solvers.

concurrently, which will differ from the original resource sharing. In all cases, however, if a solver finds a solution, we interrupt the executions.

The execution engine also supports a fault-tolerance mechanism whose idea is simple: if the execution of a solver fails, it is restarted automatically. Thus, our parallelization ensures that, even when faults occur, the whole processing is not necessarily penalized: we have independent parallel solvers; the failure of one will not damage the other runs.

We have explained the concept of a Qsat plan and its execution. To end this section, we propose to consider the notion of a team, which we will use in the design of malleable resource sharing schedules.

2.5. Solver deployment and teams. Until now, we have assumed that heuristics are restarted between phases. In Fig. 2, this means that between the first and the second phase, we restart h_3. In practice, however, this might not be a good option. Indeed, if the solvers are not randomized, it is not interesting to halt the execution on a compute node for restarting again the same solver on the same computing units. The general rule that we will adopt in the execution engine is that *if between consecutive phases there is a solver on one computing unit, we do not restart it, but we continue the execution*.

The question of restarting is related to another problem we have not discussed: in each phase, on which computing units do we deploy each server? Let us note that given $S_i^1 = 1$, we have m distinct computing units on which we can start the heuristic h_i. To address the question of the deployment, we propose the following rule: *if between consecutive phases there is a solver on one computing unit, we ensure that the solver will not be restarted. Otherwise, we randomly pick the computing unit of any solver*.

With these considerations, we can then see a malleable resource sharing schedule $\Gamma = [(S^1, \tau_1), \ldots, (S^Q, \tau_Q)]$ as a tuple $\Gamma' = (\alpha_1, \ldots, \alpha_m)$, where each α_i corresponds to the assignment of heuristics that we will have on the computing unit i. In other words, a malleable resource sharing schedule with Q phases is a set of malleable schedules on its different computing units. In this projection, we will call each α_i a team. In Fig. 4(a), we provide an example of a team with 3 phases. At each time instant $t \in [0, \sum \tau_u]$, we assume that $\alpha_i(t)$ defines the solver that is run. For the team that we refer to in Fig. 4(a), $\alpha_1(0) = P_1$ and $\alpha_1(t_1) = P_2$

We end here the presentation of the Qsat architecture. Next, we will discuss the algorithmic design of Qsat plans.

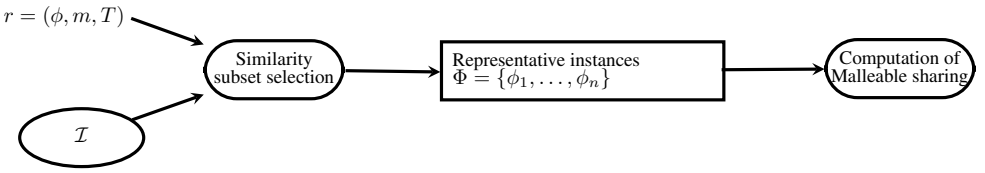


Fig. 5. Stages in the generation of a malleable schedule.

3. Generation of plans

3.1. Overview. To generate plans, the planning engine uses a basis $\mathcal{I}$ of SAT instances. $\mathcal{I}$ defines the *know-how* of the engine on the resolution of SAT. For any portfolio solver $P_i \in \mathcal{H}$ and any computational instance I, this know-how is modeled by two values: $cost(P_i, I)$ and $cover(P_i, t)$. While the former is the average running time required by P_i to process I, the latter is the subset of $\mathcal{I}$ that can be solved by P_i in spending at most t time units. $cost(P_i, I)$ and $cover(P_i, t)$ are estimated in an off-line setting, when there is no request sent to Qsat. By definition, $cover(P_i, t)$ is computed from $cost(P_i, I)$.

Both $cost(P_i, I)$ and $cover(P_i, t)$ are used in the decision on the most suitable malleable resource sharing schedule. More precisely, given a request $r = (\phi, m, T)$, the planning engine will start by computing the subset $\Phi \subseteq \mathcal{I}$ of the n instances that are *most similar to* I (similarity is discussed further on). For this purpose, a distance function is used. Here, n is an internal parameter of Qsat. From the running times $cost(P_i, I)_{i=1,\ldots,k, I \in \mathcal{I}}$, the planning engine then builds the malleable resource sharing schedule that minimizes the maximal running time in the resolution of any instance in Φ.

The two stages described below are illustrated in Fig. 5. To compute the similarity between SAT instances, any SAT instance is also described as a set of features, represented as a multi-dimensional real vector. The dimensions correspond to different measures, like the number of variables of the SAT instance, the number of clauses, the number of variables per clauses, etc. Generally, the different measures of this vector correspond to a subset of the features used in the Satzilla solver (Xu *et al.*, 2008). Given the feature representation of SAT instances, the similarity between them is computed with the Euclidean distance: the lower the distance between two instances, the more similar they are.

3.2. Computation of resource sharing schedules. Once the subset Φ is determined, the planning engine generates the resource sharing schedule by solving the following problem:

INPUT: We assume a finite set of solvers $\mathcal{H}$, a request $r = (\phi, m, T)$ and the set of representative instances Φ. At each discrete time unit $t \in \{1, \ldots T\}$, we have for each $P_i \in \mathcal{H}$ the values of $cover(P_i, t)$.

QUESTION: Let $\Gamma(\mathcal{H}, T)$ be the set of resource sharing schedules whose maximal timeout is T. For each schedule $\Gamma^j \in \Gamma(\mathcal{H}, T)$, let $cover(\Gamma^j) \subset \Phi$ be the subset of instances for which the run of Γ^j can state whether or not we have a satisfiability result in at most T time units. The objective is to choose the schedule Γ^{opt} such that

$$opt = \arg \max_{1 \leq j \leq |\Gamma(\mathcal{H},T)|} |cover(\Gamma^j)|.$$

This computational problem is inspired by the discrete resource sharing problem ($dRSSP$) introduced in a prior work (Goldman *et al.*, 2012). However, it completely differs from our prior problem formulations, where we focused on the minimization of the running times. We adopt this formulation because it is more suitable for a cloud context where users are more interested in putting maximal time limits for the execution in order to minimize the price to pay.

The idea in this problem formulation is to choose the plan that can solve the maximum number of Φ instances within T time units. This is motivated by the fact that, if ϕ is *similar* to Φ, then we are computing a schedule that will maximize the probability to have an answer for r before T time units. The NP-hardness of the problem can easily be established from a reduction to the maximum coverage problem (Hochbaum and Pathria, 1998). In the next section, we will consider the resolution of our resource sharing problem.

4. Heuristics for approximating optimal resource sharing

We address the resource sharing problem with the multi-phase approach introduced by Goldman *et al.* (2012). In this approach, instead of considering all possible schedules, we restrict the solution to a subset of schedules that we refer to as *phase-schedules*. For such schedules, we assume a maximal number of phases for resource sharing schedules. The optimal solution is found in an iterative process where local optimizations are used to build phases. The design of phase-schedules is discussed in subsequent sections.

4.1. 1-Phase schedules. We define a 1-phase schedule as a schedule $\Gamma = (\alpha_1, \ldots, \alpha_m)$ on which the following rule is satisfied: $\forall t, t' \in \{1, \ldots, T\}, \alpha_i(t) = \alpha_i(t')$.

If we restrict the resource sharing problem of Section 3.2 to 1-phase schedules, then the resulting computational problem can be mapped onto the maximum coverage one (Hochbaum and Pathria, 1998). Indeed, an instance of the maximum coverage is given by an integer k and a collection of sets $C = \{C_1, \ldots, C_m\}$. The objective is to select at most k sets C' such that $\left|\bigcup_{c_i \in C'} c_i\right|$ is maximized. The mapping with the design of a 1-phase schedule is simple. Given any resource sharing schedule $\Gamma = (\alpha_1, \ldots, \alpha_m)$, it suffices to assume that, in each team α_i, there is a unique solver $P_u \in \mathcal{H}$ that is always executed. We then build the optimal schedule Γ^{opt} by selecting the subset of m solvers $P'_1, \ldots, P'_m$ for which $\left|\bigcup_{P'_i} cover(P'_i, T)\right|$ is maximized.

Let us assume that $q = |\mathcal{H}|$ and $n = |\Phi|$. Thus we have the following result.

Theorem 1. *There is an* $1 - 1/e$ *approximation of the optimal* 1*-phase schedule in* $O(nqm)$.

The proof follows from the fact that the greedy algorithm for maximum coverage has this approximation ratio.

4.2. 2-Phase schedules. In a 2-phase schedule, there exists at most one date at which the solver of a team is switched. In Fig. 6, we provide an illustration of such a schedule. 2-phase schedules generalize 1-phase ones. Nevertheless, we propose to approximate optimal 2-phase schedules in using again the greedy solution for the maximum coverage problem. The main observation that supports this view is that we can again formulate the construction of the optimal 2-phase schedule as the solution of a maximum coverage problem.

Indeed, let us consider a 2-phase schedule $\Gamma = (\alpha_1, \ldots, \alpha_m)$. By definition, there might exist a date t where, for a team α_i, $\alpha_i(t) \neq \alpha_i(t+1)$. We capture this with the notation $\alpha_i = [(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$. It states that the execution of α_i consists in running the solver $P_{\alpha_i^1}$ from 0 to $t_{\alpha_i^1}$ and then running $P_{\alpha_i^2}$ from $t_{\alpha_i^1} + 1$ to T. In the case where $t_{\alpha_i^1} = 0$, we have a 1-phase schedule. With this basis, we propose a process with four stages for the design of 2-phase schedules.

The first stage of the process consists of generating all the possible combinations with the structure $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$. Each of these combinations corresponds to a potential team execution. Let $\mathcal{H}^2$ be the set of combinations we generated. In the second stage, we compute the instances that each $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$ solves. For this, we use the formula

$$cover([(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]) = cover(P_{\alpha_i^1}, t_{\alpha_i^1}) \cup cover(P_{\alpha_i^2}, T - t_{\alpha_i^1} + 1).$$

In the third stage, we iterate over the *possible ending dates* of the first phase and apply the greedy algorithm of the maximum coverage one. More precisely, at the iteration $t \in \{0, \ldots, T\}$ in this stage, we select the subset $\mathcal{H}^2(t)$ defined by the schedules $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$ such that $t_{\alpha_i^1} = t$. Given these schedules, we next address the local problem of choosing the m schedules (each schedule is run by a team) of this subset that collectively ensure that a maximum number of instances are covered. From the result of the prior section, one can notice that this local problem corresponds to the maximum coverage problem restricted to $\mathcal{H}^2(t)$. In the last stage of the algorithm, we choose over all possible ending dates (for the first phase), the schedule that leads to the maximal coverage.

It is easy to notice that the theoretical guarantee of the greedy resolution of the maximum coverage problem will characterize the process we describe. On the contrary to the construction of a 1-phase schedule, the time complexity of the process is more important because we have more schedules to explore.

We proposed to build 2-phase schedules in solving several 1-phase scheduling problems. After the computation of $\mathcal{H}^2$, we iterated over the ending dates of the first phase because in 2-phase schedules we have at most one ending date where the schedule of any team is changed. In practice, it might be interesting to consider situations where phases are not related to all teams but are to be considered *per team*. This means that we could have two teams for which the ending date of the first phase is different. We will name such schedules *extended* 2*-phase schedules*. One interest in such schedules is that we could use a simplified version of the previous process for their design. This is discussed below.

4.3. Extended 2-phase schedules. For the design of extended 2-phase schedules, we consider the following stages: (i) generation of possible combinations $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$, (ii) computation of the instances that each $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$ solves and (iii) selection of the m schedules (among the generated ones) when covering the maximal number of instances. This stage is achieved with the greedy algorithm of the maximum coverage problem.

For the generation of all possible combinations $[(P_{\alpha_i^1}, t_{\alpha_i^1})(P_{\alpha_i^2}, T)]$ (where $t_{\alpha_i^1} \in \{0, \ldots, T\}$ and $P_{\alpha_i^1}, P_{\alpha_i^2} \in \mathcal{P}$), we propose to use a brute force approach. In the worst case, this will lead us to

$$\sum_{t=0}^{T} \binom{q}{1} \cdot \binom{q-1}{1}$$

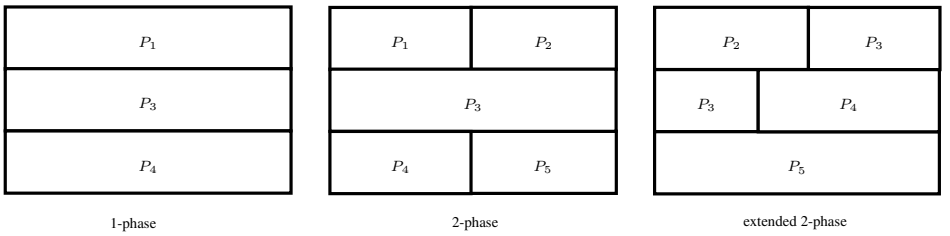


Fig. 6. Examples of schedules.

combinations. Therefore, we have the following result.

Theorem 2. *There is an* $1 - 1/e$ *approximation of the optimal extended* 2*-phase schedule in* $O(nq^2Tm)$.

The interest in extended 2-phase schedules is that, though they generalize 2-phase schedules, we have for their construction a simplified process. Indeed, after the generation of candidate schedules that can be run in teams, we do not need to iterate over ending dates before applying the greedy algorithm. As we will see in the experiments, this can lead to a significant runtime gain. In the next section, we will consider the case of k-phase schedules, $k > 2$.

4.4. k-Phase schedules. The solution we proposed for 2-phase schedules can be generalized to k-phase schedules. We will thus have up to $O(q^kT)$ possible schedules per team. This might be too much to store in memory. Below, we propose an alternative based on dynamic programming.

4.4.1. Principle. The dynamic programming solution operates on a two-dimensional array $sol(:)$ of $k \times T$ entries. The dimensions of the arrays are the duration of the schedule and the number of its phases. An entry $sol(h, t)$ refers to the best local solution that uses h phases and is run until the maximal date t. In the first iteration of the algorithm, we build the column $sol(1, :)$. In the second, we build $sol(2, :)$ and we repeat until we have $sol(k, :)$. The column $sol(1, :)$ is computed by applying the greedy algorithm for 1-phase schedules with different values of the maximal runtime. The second column refers to 2-phase schedules. However, we do not generate it from the greedy algorithm we defined in Section 4.2. Instead, we use a completion algorithm that computes any $sol(h, t)$ from $sol(h - 1, t)$. The idea of this algorithm is to *complete* schedules obtained at phases $h - 1$ with a supplementary phase that ends at date t. An illustration of the dynamic programming execution is given in Fig. 7. We will now focus on the completion algorithm.

4.4.2. Completion algorithm. At the iteration $u \in \{1, \ldots, T\}$ of the dynamic programming algorithm, let us assume that the objective is to build $sol(h, u)$. The completion algorithm will start by selecting the local solutions $sol(h - 1, t)$, where $t \leq u$. Then, the algorithm computes the set of instances solved by this local solution and removes them from Φ. Let Φ' be the remaining instances. The iteration continues with the updates of the values of $cover(P_i, t)$ for instances in Φ'. The update task consists in reducing from the P_is runtime the processing time already invested in $sol(h - 1, t)$. Finally, the completion algorithm runs the 1-phase approximation algorithm for finding the best phase-schedule to process Φ' assuming the updated runtime. The resulting 1-phase schedule is added as a supplementary phase at the end of the execution of $sol(h - 1, t)$ and saved as $sol(h, u, t)$. For deriving $sol(h, u)$, the completion algorithm iterates over different values of $t \leq u$ and returns the schedule $sol(h, u) = sol(h, u, opt)$ such that

$$|cover(sol(h, u, opt))| = \max_{t \leq u}\{|cover(sol(h - 1, u, t))|\}.$$

We end here the presentation of the resource sharing algorithm used in Qsat. In the next section we report an experimental evaluation of the algorithms.

5. Experimental evaluation

We report two series of experiments. The objective in the first one is to compare the quality of results of the different resource sharing algorithms. In the second series, we are interested in estimating the runtime overhead of the execution engine. It is important to note that the different algorithms we consider are implemented in the Qsat planning engine. However, for the sake of reproducibility, in this section we will present the results of their simulation on a public SAT benchmark.

5.1. Comparison of the resource sharing algorithms. These experiments consisted of simulations based on the

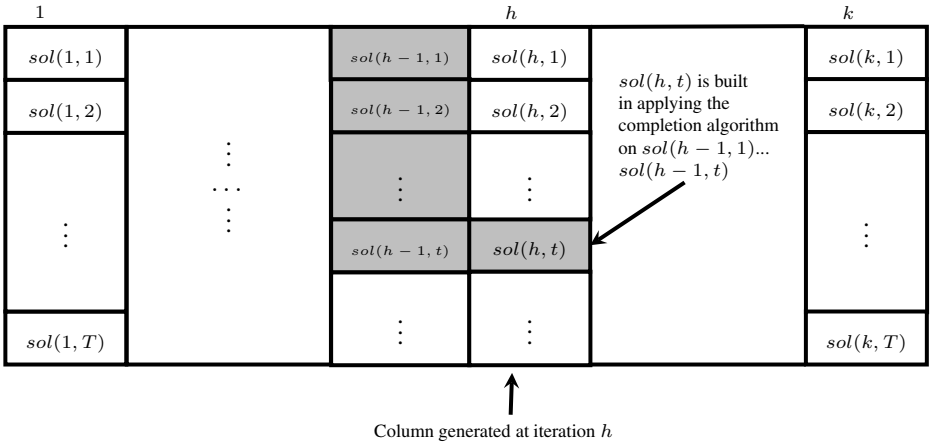


Fig. 7. Dynamic programming. The element $sol(h, t)$ is generated at iteration h of the algorithm. Its computation is based on the first t elements of the column $h - 1$.

Table 1. Experimental plan.

Ref.	Parameters
C1	name = Core_Solvers_Parallel_Application_SATUNSAT, $n = 300$, $q = 12$
C2	name=Core_Solvers_Parallel_Hard-combinatorial_SATUNSAT, $n = 300$, $q = 10$
C3	name = Parallel_Application_SATUNSAT, $n = 300$, $q = 14$
C4	name = Parallel_Hard-combinatorial_SATUNSAT, $n = 300$, $q = 14$
C5	name = Parallel_Random_SAT, $n = 225$, $q = 6$
Exp.	**Parameters**
exp.1	$T = 500$, $m = 4$, instances and portfolio solvers: C1-C5
exp.2	$T = 1000$, $m = 4$, instances and portfolio solvers: C1-C5
exp.3	$T = 2000$, $m = 4$, instances and portfolio solvers: C1-C5
exp.4	$T = 500$, $m = 5$, instances and portfolio solvers: C1-C5
exp.5	$T = 1000$, $m = 5$, instances and portfolio solvers: C1-C5
exp.6	$T = 2000$, $m = 5$, instances and portfolio solvers: C1-C5

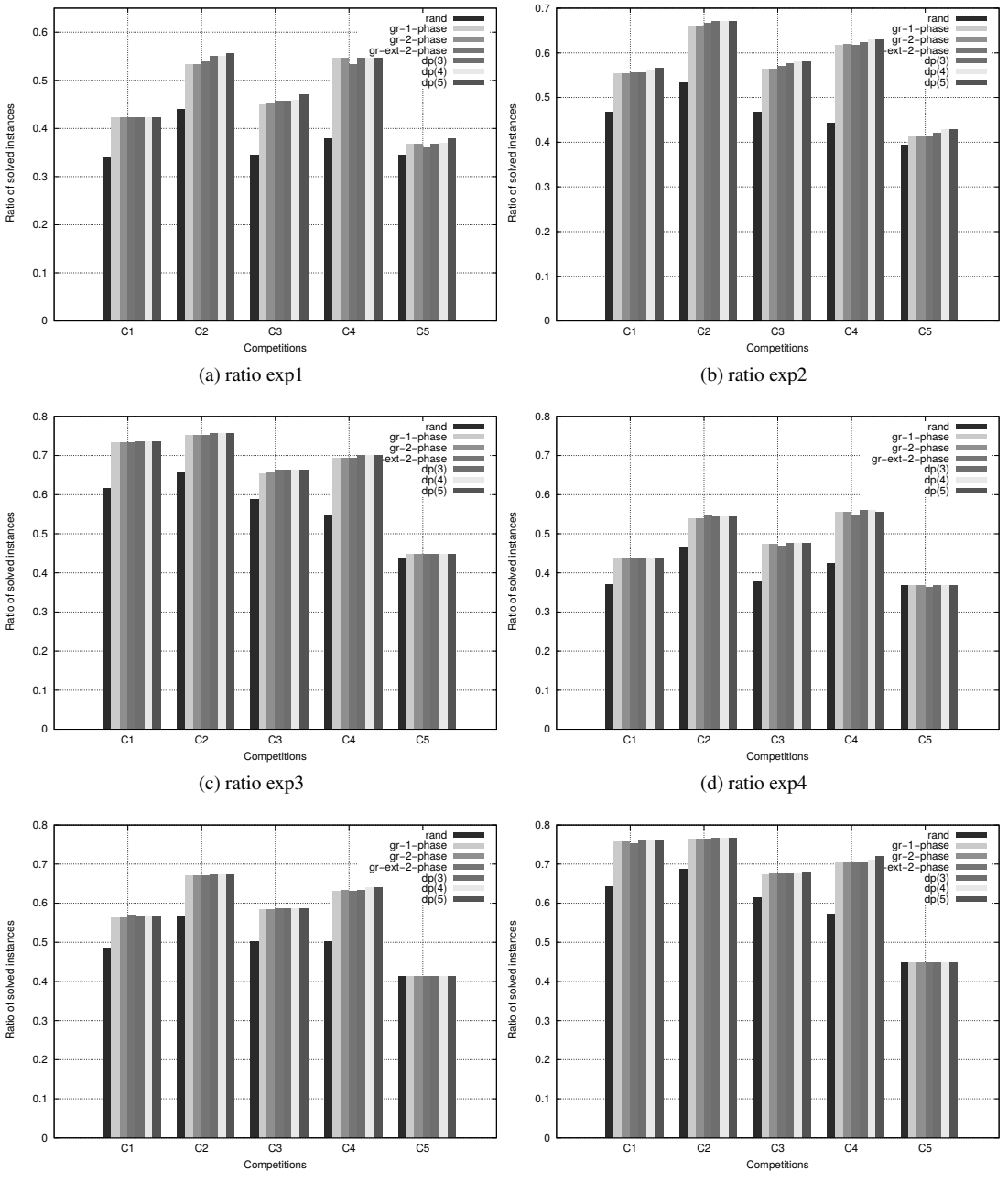
data of the 2013 international SAT competition.[8] The selected data are the results of 5 parallel SAT competitions whose names and settings are given in Table 1. The values chosen for T *capture* the median CPU times of the best, average and worst solvers of the competition. The values of m were chosen to obtain solutions of the dynamic programming algorithm in *reasonable* computing times.

Within these results, for each competition, we have the runtime measured by evaluating a finite set of SAT instances with a finite set of parallel SAT solvers. For the simulation, we assumed that, in each competition, the evaluated parallel SAT solvers are our set of solvers ($\mathcal{H}$) and the instances used for the evaluation are the representative data from which we want to build an optimal resource sharing schedule. Then, we performed 6 experiments, where we estimated the runtime and quality of the results of the resource sharing heuristics proposed in this paper. In each experiment, for the resource sharing problem to be solved, we fixed values of T and m. The choices are summarized in Table 1. Finally, by the quality of results for a heuristic, we mean the ratio between the number of instances that the heuristics can solve (for the resource sharing problem considered) versus the total number of instances.

In Fig. 8, we present the quality of the results observed in the different experiments. Here, $dp(k)$ refers to the run of the dynamic programming algorithm for k-phase schedules. Here *rand* refers to the random 1-phase schedule that we build in randomly choosing the m portfolio solvers. The quality of results is the mean value obtained from 300 runs.

Regarding the quality of results, one can notice that the greater the value of T, the higher the quality of results. This is expected since, the greater T, the more instances

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



(a) ratio exp1

(b) ratio exp2

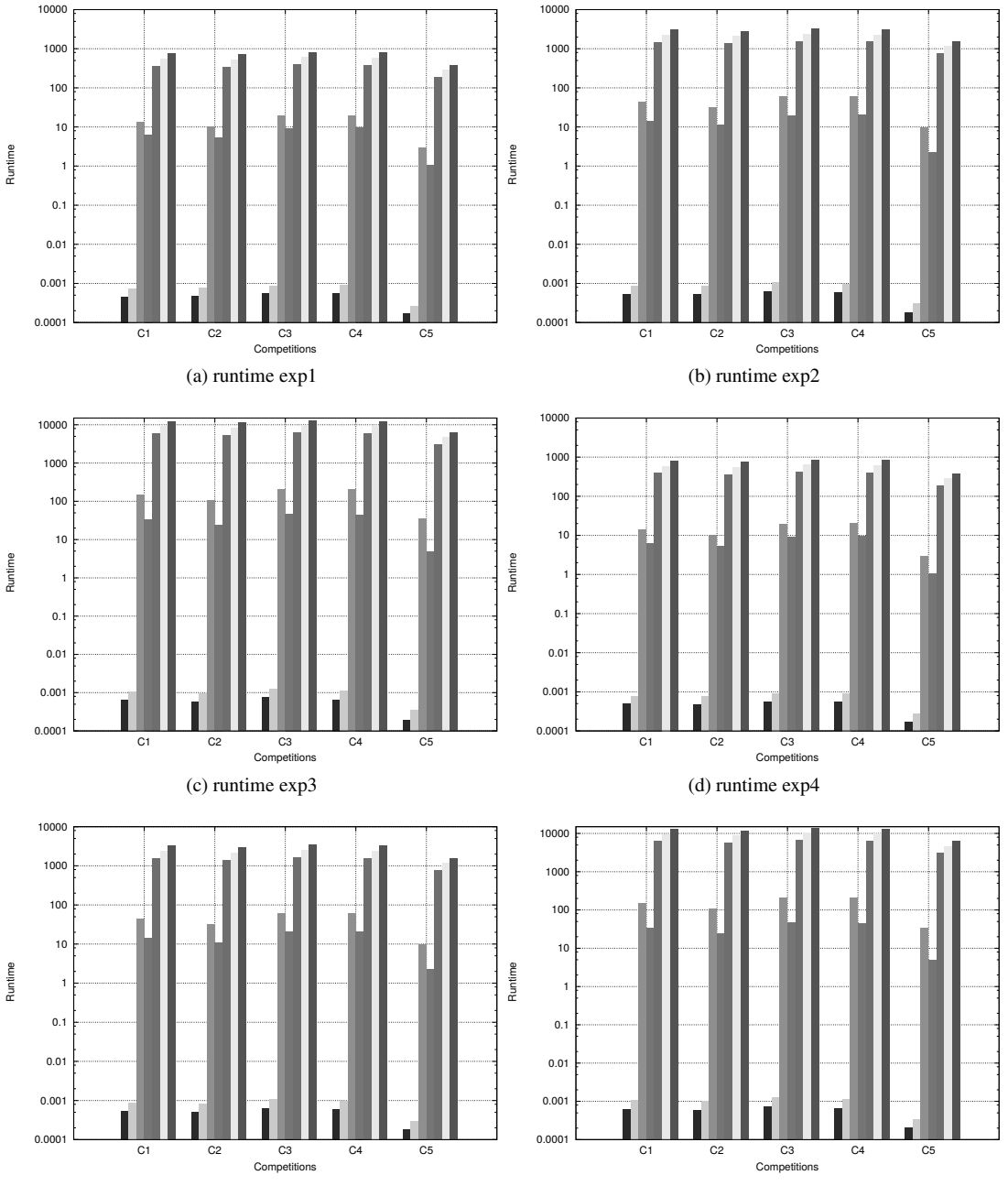
(c) ratio exp3

(d) ratio exp4

(e) ratio exp5

(f) ratio exp6

Fig. 8. Quality of results of the different heuristics.



(a) runtime exp1

(b) runtime exp2

(c) runtime exp3

(d) runtime exp4

(e) runtime exp5

(f) runtime exp6

Fig. 9. Runtime of the heuristics (in seconds; we consider the heuristics of Fig. 8).

we can solve. We also observed the same trend on the value of m, but the progression was less significant.

The second lesson of these experiments is that there is a clear difference between the random construction of the schedules and the other heuristics we proposed in this paper. This justifies the algorithmic contributions we made in this paper. Nonetheless, the difference is more effective on the competitions C1 and C4 than C5. This is because there is less performance diversity in C5: solvers in this competition were in most cases unable to solve the input SAT instances.

The third lesson is that, in general, the greater the number of phases, the better the quality of results we could expect. This was predicted, but the performance variations we observed were not important.

The fourth lesson we learned is that if dynamic programming was more efficient in terms of the quality of results, its runtime made it hardly exploitable in an online setting. As show in Fig. 9, generating dynamic programming solutions took several hours. In comparison, the greedy heuristics (and in particular the greedy-1-phase) were more competitive. Regarding the runtime, let us mention that the measurements were done with $Qsat$.

Summarizing, these experiments show that the best options for the implementation of $Qsat$ consist in using the greedy 1-phase algorithm or the extended 2-phase solution. Indeed, on representative SAT data (extracted from the official SAT competition), these two heuristics take only few seconds for generating solutions that are competitive in terms of quality. In addition, we could not tolerate a huge runtime in the determination of the optimal resource sharing schedule since, in the request/answer model we considered, the user expects a solution in a maximal runtime. This conclusion, however, does not mean that the other heuristics could not be interesting in some cases. For instance, if we consider instead a setting where the optimal resource sharing schedule is to be computed offline (as in the work of Goldman *et al.* (2012)), these later solutions could be exploited since we could have *enough time* for building the optimal results.

6. Conclusion

In this paper, we introduced a new distributed service for the resolution of SAT. The proposed service is based on an algorithm portfolio. It features two components: a planning engine, which computes an optimal plan (or resource sharing) for the parallelization, and an execution engine, which runs this plan. Both the engines work with a resource sharing model that we introduced in our prior work (Goldman *et al.*, 2012), but not for a distributed cloud context. In particular, we explained how to transform a resource sharing schedule into a *pack of docker containers*. Moreover, we introduced novel algorithms for the generation of near-optimal resource sharing schedules. We also proposed an experimental evaluation where we analyzed the quality of results and the runtime of our algorithms. The results showed that some of the proposed algorithms have a better trade-off between the quality of results and the runtime.

For continuing this work, we intend to introduce rescheduling in $Qsat$. As stated before, the schedule submitted by $Qsat$ will not always be executed as is done by the Qarnot scheduler. For handling these cases, a good approach consists in enhancing the execution engine in $Qsat$ for detecting *deviations* in the execution of the resource sharing schedule. Once these deviations are identified, a rescheduling process will be launched.

Acknowledgment

This work was funded by the Greco project[9] of the French National Agency for Research (ANR).

References

Audemard, G., Hoessen, B., Jabbour, S., Lagniez, J.-M. and Piette, C. (2012). PeneLoPe, a parallel clause-freezer solver, *SAT Challenge 2012: Solver and Benchmarks Descriptions, Helsinki, Finland*, pp. 43–44.

Audemard, G., Hoessen, B., Jabbour, S. and Piette, C. (2014). Dolius: A distributed parallel SAT solving framework, *5th Pragmatics of SAT Workshop POS-14, Vienna, Austria*, pp. 1–11.

Biere, A. (2010). Lingeling, plingeling, PicoSAT and PrecoSAT at SAT Race 2010, *Technical report*, Johannes Kepler University, Linz.

Chrabakh, W. and Wolski, R. (2003). Gridsat: A Chaff-based distributed SAT solver for the grid, *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC'03, Phoenix, AZ, USA*, pp. 37–50, DOI: 10.1145/1048935.1050188.

Goldman, A., Ngoko, Y. and Trystram, D. (2012). Malleable resource sharing algorithms for cooperative resolution of problems, *Congress on Evolutionary Computation World Congress on Computational Intelligence, Brisbane, Australia*, pp. 1438–1445.

Hochbaum, D.S. and Pathria, A. (1998). Analysis of the greedy approach in problems of maximum k-coverage, *Naval Research Logistics* **45**(6): 615–627.

Holldobler, S., Manthey, N., Nguyen, V.H., Stecklina, J. and Steinke, P. (2011). A short overview on modern parallel SAT-solvers, *2011 International Conference on Advanced Computer Science and Information System (ICACSIS), Jakarta, Indonesia*, pp. 201–206.

Jackson, P. and Sheridan, D. (2005). Clause form conversions for boolean circuits, *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing, SAT'04, Vancouver, BC, Canada*, pp. 183–198.

[9] `https://anr-greco.net/`.

Kautz, H. and Selman, B. (2007). The state of SAT, *Discrete Applied Mathematics* **155**(12): 1514–1524.

Martins, R., Manquinho, V. and Lynce, I. (2012). An overview of parallel SAT solving, *Constraints* **17**(3): 304–347.

Ngoko, Y., Saintherant, N., Cérin, C. and Trystram, D. (2018). Invited paper: How future buildings could redefine distributed computing, *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS 2018, Vancouver, BC, Canada*, pp. 1232–1240.

Ngoko, Y., Trystram, D., Reis, V. and Cérin, C. (2016). An automatic tuning system for solving NP-hard problems in clouds, *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS 2016, Chicago, IL, USA*, pp. 1443–1452.

Roussel, O. (2011). Description of Ppfolio, `https://www.cril.univ-artois.fr/~roussel/ppfolio/solver1.pdf`.

Silva, J.a.P.M. and Sakallah, K.A. (1996). GRASP—a new search algorithm for satisfiability, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design, ICCAD'96, San Jose, CA, USA*, pp. 220–227.

Vardi, M.Y. (2014). Moore's law and the sand-heap paradox, *Communications of the ACM* **57**(5): 5–5.

Xu, L., Hutter, F., Hoos, H.H. and Leyton-Brown, K. (2008). Satzilla: Portfolio-based algorithm selection for SAT, *Journal of Artificial Intelligence Research* **32**: 565–606.

Zhang, H., Bonacina, M.P. and Hsiang, J. (1996). PSATO: A distributed propositional prover and its application to quasigroup problems, *Journal of Symbolic Computation* **21**(4–6): 543–560.

Yanik Ngoko received his BSc in computer science from the University of Yaoundé I (UYI), Cameroon, his MSc in parallel and numerical computing also from the UYI, and his doctorate in computer science from the Grenoble Institute of Technology, France (2010). From 2011 to 2014, he was a postdoctoral researcher, first at the university of São Paulo and then at the University of Paris 13. Since October 2014, he has been a research scientist at Qarnot Computing and an associate member of the Computer Science Laboratory of Paris Nord (University of Paris 13). His research interests include parallel and distributed computing, web services, cloud computing, and applications of edge computing to IoT.

Christophe Cérin has been a professor of computer science at the University of Paris 13, France, since 2005. In 2015, he was involved in an infrastructure project related to big data and high performance computing for e-sciences for Paris-Sorbonne University. At Paris 13, he chairs the board for the cluster computing facility available to all campus scientists. His current industrial experience includes serving as a local chair for the Wolphin project (Alterway, Gandhi, Objectif Libre and Paris 6). His recent industrial experience has been for the Wendelin and Resilience projects related to cloud and big data. He is also active with RSU (Rèves de Scènes Urbaines—industrial demonstrator for sustainable cities), Qarnot Computing and Umanis. His research focuses on high performance computing, including grid and cloud computing, and he develops middleware, algorithms, tools and methods for distributed systems.

Denis Trystram has been a professor of computer science at the Grenoble Institute of Technology since 1991 (now a distinguished professor). He was a senior member of Institut Universitaire de France from 2010 to 2014. In 2011 he obtained a Google research award for his contributions in the field of multi-objective optimization. Denis is leading a research group on optimization of resource management for parallel and distributed computing platforms in a joint team with Inria. Since 2010, he has been the director of the international Master program in computer science at the University of Grenoble Alpes. He has been recently elected the director of the research pole in maths and computer science in that university.

Received: 26 July 2018
Revised: 10 February 2019
Accepted: 2 March 2019