Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-020-09704-4

Real-time solving of computationally hard problems using optimal algorithm portfolios

Yair Nof[1] · Ofer Strichman[1]

Published online: 28 August 2020
© Springer Nature Switzerland AG 2020

Abstract

Various hard real-time systems have a desired requirement which is impossible to fulfill: to solve a computationally hard optimization problem within a short and fixed amount of time T, e.g., $T = 0.5$ seconds. For such a task, the exact, exponential algorithms, as well as various Polynomial-Time Approximation Schemes, are irrelevant because they can exceed T. What is left in practice is to combine various anytime algorithms in a parallel portfolio. The question is how to build such an optimal portfolio, given a budget of K computing cores. It is certainly not as simple as choosing the K best performing algorithms, because their results are possibly correlated (e.g., there is no point in choosing two good algorithm for the portfolio if they win on a similar set of instances). We prove that the decision variant of this problem is NP-complete, and furthermore that the optimization problem is approximable. On the practical side, our main contribution is a solution of the optimization problem of choosing K algorithms out of n, for a machine with K computing cores, and the related problem of detecting the minimum number of required cores to achieve an optimal portfolio, with respect to a given training set of instances. As a benchmark, we took instances of a hard optimization problem that is prevalent in the real-time industry, in which the challenge is to decide on the best *action* within time T. We include the results of numerous experiments that compare the various methods. Hence, a side effect of our tests is that it gives the first systematic empirical evaluation of the relative success of various known stochastic-search algorithms in coping with a hard combinatorial optimization problems under a very short and fixed timeout.

Keywords Algorithm portfolios · NP-optimization · Real-time

Mathematics Subject Classification (2010) 68T20

✉ Yair Nof
yair.nof@gmail.com

Ofer Strichman
ofers@ie.technion.ac.il

[1] Information Systems Engineering, Technion, Haifa, Israel

1 Introduction

Hard real-time systems frequently have a seemingly impossible requirement: To solve within a short, fixed amount of time T (e.g., $T = 0.5$ second), a computationally hard optimization problem. For example, a centrally-controlled team of robots playing soccer need to make split-second decisions, but those decisions are subject to hard constraints such as not walking into each other or not hitting a teammate with the ball. Since there are multiple players and each has multiple options, the number of combinations grow exponentially with the number of players. Another example is a fleet of drones with a central control. The fleet has to fly in an urban area and hence react to the environment, and make decisions in real-time. Those decisions are subject to both hard constraints (e.g., not to collide) and to soft objectives, such as minimizing the time to complete the mission. Here again the number of possible moves of the system grow exponentially with the number of drones and other players in the environment.

The fixed time-bound on solving such problems can emanate from the interaction with the physical world: the other team and the ball in the case of the soccer robots; or various hazards in the case of the urban drone-fleet; it can also emanate from the control system itself, if it operates in a clocked closed feedback-loop.

Although for some hard optimization problems there are known polynomial-time approximations schemes (PTAS) [31], which can guarantee a $1 - \epsilon$ proximity to the optimal solution,[1] such solutions are generally irrelevant because they can exceed the hard time-bound T. Moreover, the run-time increases when the precision parameter ϵ becomes smaller.

What is left, then, is to measure the relative success of various *anytime* algorithms in solving such a problem. However, simply counting the times that a given algorithm was able to satisfy the hard constraints is not sufficiently refined, because even the hard constraints can be prioritized (i.e., weighted) among themselves, and in comparison to the soft constraints. This internal prioritization reflects what is done in practice with such systems. In the following discussion we describe this concept more formally.

Our focus is on optimization problems that their decision variant is complete in NP (henceforth, NP-optimization problems—see [4] for examples); By definition, those can be reduced to the NP-complete Constraint Satisfaction Problem (CSP),[2] which gives us a unified starting point. CSP has an optimization variant called the Constraint Optimization Problem (COP) [5], in which the goal is to satisfy the constraints while maximizing some objective function. It is common to distinguish between *hard* and *soft* constraints in COP, where each of the latter is associated with a weight that reflects the 'reward' for satisfying it. Every solution has to satisfy the hard constraints, and an optimal solution has to additionally maximize the reward by satisfying soft constraints.

We now define a *fixed-time variant* of an NP-optimization problem. Given the fixed time-bound T our goal is to find algorithms that their solution is as good as possible at time T. This, by definition, implies that we cannot guarantee that our solution satisfies all the hard constraints of the original optimization problem, and we therefore need to prioritize them by giving them weights. In other words, we need to turn the hard constraints into soft constraints. This gives rise to the following definition:

[1]Throughout this work we refer to maximization problems. The definitions for minimization problems are similar: in this case it will be a $1 + \epsilon$ proximity.

[2]CSP generalizes the propositional satisfiability problem (SAT), but is still in NP. It allows variables with any finite discrete domain (rather than SAT's restriction to the Boolean domain), and a rich modeling language.

Definition 1 (**The fixed-time variant of an NP-optimization problem**) Given a

- NP-hard optimization problem P cast as a COP, and
- weights to the hard constraints, reflecting their importance relative to each other and the original soft constraints (if there are any),

let $soft(P)$ denote a problem identical to P except that all the hard constraints are turned into soft constraints with the given weights. Given a time limit T, the *fixed-time* variant of P, denoted $FT(P)$, is the problem of finding a solution within time T to $soft(P)$.

Here we focus on a particular variant of such an optimization problem, that is concerned with decisions that a system is making about its next action (or actions), in real-time. This variant is prevalent in the real-time industry. In such a system the hard constraints can typically be divided into *safety* and *liveness* constraints (this dichotomy is well-known in the verification community). Informally the former states what should not happen, whereas the latter states what should happen. Going back to our drones example, a safety constraint is that the drones avoid close interactions with obstacles and other drones. On the other hand a liveness constraint is that the drones reach a certain location.

Generally it is easy to satisfy the safety constraints while violating the liveness constraints: simply do nothing. In our drones example, the safety constraints can be satisfied by giving an instruction to the drones to hover in place; in the robot team example, they can be instructed to not move. These are examples of 'no-action' decisions, which must be available for solving the fixed-time variant of hard decision problems. Hence, given a suggested solution that does not satisfy the safety constraints, we can always change it so it satisfies those constraints by choosing the 'no-action' option. Accordingly, the search heuristic is constructed such that taking a no-action reduces the value of the suggested solution. Systems that do not have this property, i.e., they have hard safety properties which cannot be satisfied by relaxing a hard liveness constraint, are not suitable for our framework.

The main contribution of the article is that it solves the problem of *optimizing a parallel portfolio of algorithms*, for a given number K of available computing cores, and the related problem of finding the lowest value of K for which we can get an optimal portfolio. We prove that the decision variant of this problem is NP-complete in Section 4.2, and that the optimization variant is approximable in Section 4.3, and hence belongs to the APX complexity class.

A side effect of our tests is that it also gives us the first systematic empirical evaluation of the relative success of known anytime algorithms in coping with a hard combinatorial optimization problems under a very short and fixed timeout (the algorithms were implemented in a unified framework and their parameters were automatically tuned for the set of instances, to make the comparison as fair as possible).

We continue in the next section by briefly describing a set of anytime algorithms that cope with the fixed time-variant of COPs. Most of these algorithms are adaptation of known stochastic search algorithms and their combinations. In Section 3 we describe empirical evidence of the relative performance of the various algorithms.

In Section 4 and Section 5, which describe the main contribution of this article, we suggest methods to construct optimal parallel portfolios of the algorithms, based on modeling the selection optimization problem and solving it with a Satisfiability Modulo Theories (SMT) engine. In Section 6 we discuss empirical results comparing parallel portfolios constructed in different ways, and we conclude in Section 7. All the experiments in this article are based on instances that model an NP-hard optimization problem that roughly models the

urban drones-fleet example that was mentioned in the beginning of this section. We tested overall 1000 instances, in two batches, which we call ‘normal’ and ‘hard’, the latter having the property of having a much larger number of drones to coordinate.

2 The applied algorithms

A detailed description of the algorithms mentioned below appear in the first author’s thesis [26]. Here we will only describe them briefly and also describe the framework that we used for implementing them.

2.1 Local search

Local search is a heuristic framework for solving hard optimization problems. Local search is relatively simple to implement, output a stream of solutions without a setup time, and contains a rich toolbox of parameterized algorithms. Local search meta-heuristics can be easily adapted to many concrete algorithms [13], and there are also many high-level modeling languages and libraries that are useful during implementation e.g., LOCALIZER [23], EASYLOCAL++ [6], and COMET [11]. Local search moves between neighboring candidate solutions using an evaluation function, and stops if an optimal solution is found or the time bound is reached. The *Random Walk* (RW) [32], *Stochastic Hill-Climbing* (SHC)[8], *Tabu Search* (TS) [9] and *Simulated Annealing* (SA) [17] are such local-search-based algorithms that we experimented with.

2.2 Non-local-search

When the candidate solutions are sampled not necessarily following the neighboring relation, the search is not considered to be local. The *Random Search* (RS)[2], and *Cross Entropy Method* (CE) [28] are such non-local-search algorithms that we experimented with, although they are still stochastic. We also experimented with a *Greedy* algorithm and combining it with the other algorithms (this turned out to be a winning strategy as we will see).

2.3 Greedy variants

Our basic greedy algorithm is simple: For some variable ordering, for each variable:

- Choose the value that increases the most the value of the objective function, while still satisfying all the constraints under the assignment to the variables that precede it in the given order.
- In case no such value exists, assign a ‘no-action’ value. This value trivially satisfies the constraints for this variable.

The greedy algorithm yields one solution and stops. Since in our experiments it typically terminates before the timeout, it can be useful to use the remaining time for improving its solution. Below we describe various combinations that we experimented with.

Greedy loop Perform multiple iterations of the greedy algorithm, each time using a different (random) variable order.

Hybrid: greedy + local-search Use the solution of the greedy algorithm as an initial solution for any of the local-search algorithms above. Thus, local-search attempts to improve the solution given by the greedy algorithm.

Hybrid: greedy + cross-entropy In the cross entropy method each value has a certain probability to be chosen, and that value is adapted throughout the algorithm. Initially this distribution is uniform. Here we experimented with a variant that assigns some weight $0 < w < 1$ (a parameter) to each of the values in the initial solution by the greedy algorithm, whereas other values divide the remaining $1 - w$ weight among themselves. This biases the distribution towards the values of the greedy solution.

2.4 An algorithmic framework

Algorithm 1 describes the meta-heuristic that we used, which we call *Fixed-Time Search*. Each of the algorithms mentioned above is derived from it. First, the algorithm is initialized with the given initial solution. Then, we iterate until the given time limit was reached, or when the stopping criterion $Stop()$ was met (e.g., a convergence criterion was met). In each iteration, $ChooseCandidate(Current)$ returns a candidate solution, which is a function of the current solution (e.g., a neighbor of $Current$, according to some neighboring relation). For making the candidate solution our current solution, the acceptance criterion $Accept(Candidate)$ is required (e.g., the candidate solution is at most $1 + \epsilon$ times worse than the current solution). When the current solution is better than the best solution according to the $Value()$ function (which is tailored for the specific COP), we update our new best solution with the current one. If the restart criterion $RestartNow()$ is met (e.g., there was no improvement in the current solution for X iterations), then the new current solution becomes what $Restart()$ returns (e.g., a new random solution). By using a unified framework for all the algorithms, we achieve a relatively fair comparison between them. In all the following references to COP we refer to FT(COP) (see Definition 1).

Algorithm 1 $FixedTimeSearch$(Initial Solution $Init$, Time T).

```
Current ← Init
Best ← Current
while not (Stop() or timeout T reached) do
    Candidate ← ChooseCandidate(Current)
    if Accept(Candidate) then
        Current ← Candidate
        if Value(Best) < Value(Current) then
            Best ← Current
    if RestartNow() then
        Current ← Restart()
return Best
```

3 Empirical results

We generated random inputs for our empirical evaluation, based on a problem similar to the drone navigation problem that we mentioned in the introduction.

We implemented all the algorithms from Section 2 in C++ and ran experiments serially on a Dell Vostro 3360 with Core i7-3517U at 1.9-2.4Ghz and 6GB of RAM, using the same timeout.

Most of the algorithms that we tested have parameters, and those can be tuned automatically for the problem and timeout at hand, using any one of the methods described in [1, 15, 21], or [16]. We chose to experiment with the latter: it describes *ParamILS*, which is a widely used and cited tool for parameter tuning.

Our objective is to get the best quality at a given timeout T. *ParamILS* tunes an algorithm by a local-search over the space of its parameters, using a training set of instances. Since *ParamILS* uses a seed for its local-search, we used it with two different seeds to produce two different versions of the same algorithm. After tuning, *ParamILS* evaluates the quality of the tuned algorithm by running it over a test set of instances. Normally this process, to be effective, requires a long run-time, but since in our case we tune it for a short run, our algorithms can be tuned very effectively in a reasonable amount of time.

We tuned our algorithms for best quality at $T = 0.1$ second, and then ran all of them on the 'normal' batch of 500 random inputs. The results are shown in Fig. 1. We can see that the hybrid solutions lead the chart. When running on the batch of 500 harder instances the hybrid algorithms still lead the chart, although, as expected, there is some differences in the relative competitiveness of the algorithms – see Fig. 2.

As a side note, let us mention that even when tuned for $T = 1$ second, the improvement is reflected at any given time in the range 0..1 sec. This can clearly be seen in Fig. 3, where the tuned version of the algorithm dominates the non-tuned one, in two separate algorithms, along the time-line. The figure shows the simulated annealing algorithm, as one example from our tested algorithms, with and without a greedy pre-run, before and after the tuning. As a reference, we show the anytime behavior of a random search (RS).

4 Constructing an optimal parallel portfolio

Using a *parallel portfolio* of algorithms [14] can significantly improve the performance relative to a single algorithm [10]. The idea of a *static* portfolio [27] is to run several algorithms in parallel, on separate cores, and after the time bound T has elapsed, take the best result achieved by any of those algorithms. A *dynamic* portfolio is a portfolio being able to adapt per input [22]. Two automated methods for constructing a static portfolio are described in [12]. These methods integrate the tuning process with the construction of the portfolio, and were used to reduce the computation time of SAT-solving. The first method was to treat a portfolio of algorithms as one algorithm with a configuration space of all its components and tune this one algorithm to get the best quality. The second and less exhaustive method, was a greedy method of using a highly parameterized single algorithm, starting with an empty portfolio and then tuning one algorithm instance at a time, to identify the best parameters we can, allowing to improve the quality of the growing portfolio. Clearly this is not optimal since the results depend on the order of tuning.

We will focus on static parallel portfolios, composed of algorithms that have been already tuned. We assume that the algorithms are executed independently. The problem that we solve is to choose the algorithms, given that we cannot run all of them because of an insufficient number of computing cores. The choice is made based on the results achieved for a training set and the hope is that the optimization performed for this set will prove itself effective also with respect to future instances. We proceed with formal definitions of two variants of this problem.

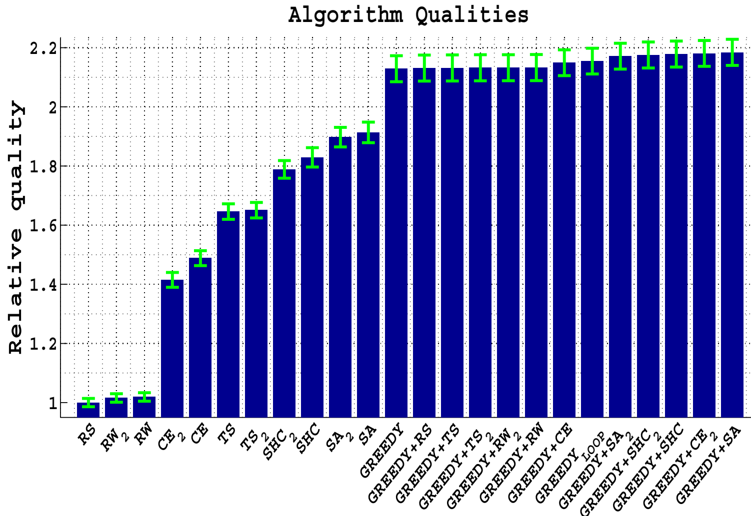

Fig. 1 Quality of various algorithms, divided by the worst quality (of Random Search) after tuning for $T = 0.1$ sec, over 500 instances (the 'normal' instances)

4.1 The *K*-algorithms cover problem

We begin with a definition of a generic problem of choosing an algorithm portfolio. In Definitions 3 and 4 we will demonstrate particular uses of it.

Definition 2 (**The** K**-Algorithms Cover Problem**) An instance of the K-algorithms cover problem is a 5-tuple $\langle S, I, M, O, K \rangle$, where:

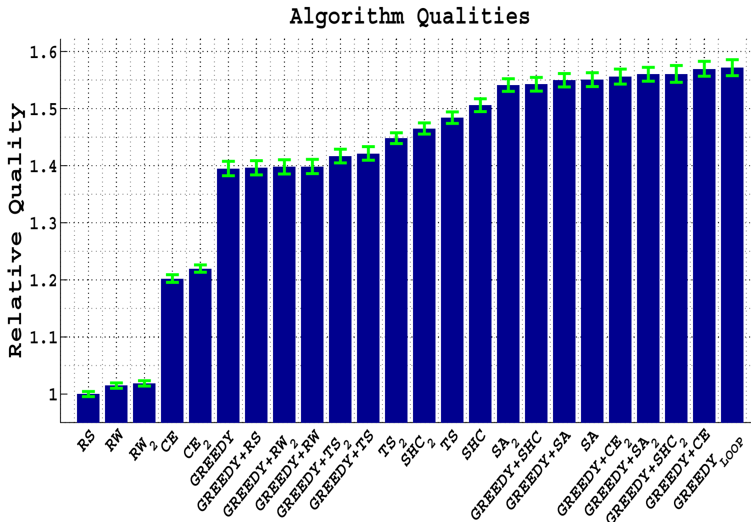

Fig. 2 Quality of various algorithms, divided by the worst quality (of Random Search) after tuning for $T = 0.1$ sec, over 500 instances (the 'hard' instances)

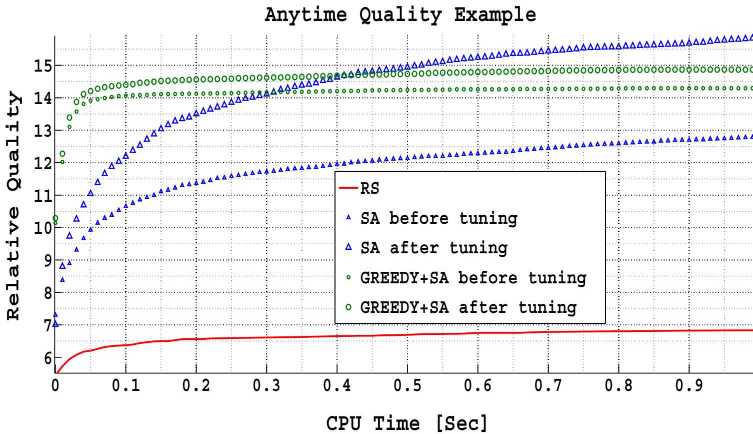

Fig. 3 Anytime quality of simulated annealing, divided by the quality of random search at its first solution, untuned vs. tuned for $T = 1$ sec, over 50 instances

- S is a set of n algorithms,
- I is a set of input instances for S,
- $M : S \times I \mapsto \mathbb{R}$ is the quality of the solution that each algorithm in S generates for each instance from I,
- O is the objective (a definition of what constitutes an optimal portfolio), and finally
- $K < n$ is the maximal number of algorithms that can be chosen from S.

A solution to the instance $\langle S, I, M, O, K\rangle$ is a parallel portfolio of K algorithms from S that optimizes O.

Definition 2 leaves open the choice of the objective O. We will now show two different choices of O that we experimented with.

Definition 3 (The K-Algorithms Max-Sum Problem) An instance of the *K-algorithms max-sum problem* is a K-algorithms cover problem with the following objective O:

$$O \doteq \arg\max_{s \subset S:|s|=K} \left(\sum_{i \in I} \left(\max_{A \in s} M(A, i) \right) \right) . \tag{1}$$

In words, the objective here, is to find a set $s \subseteq S$ such that $|s| = K$, which maximize the portfolio's sum of qualities across instances.

Example 1 Let $\langle S, I, M, O, K\rangle$ be an instance of the K-Algorithms Cover Problem, with its first three components S, I, M defined as follows:

- $S = \{A_1, A_2, A_3\}$, thus $n = 3$
- $I = \{i_1, i_2, i_3, i_4, i_5, i_6, i_7\}$
- M is defined using the following table:

O is given in (1). Let $m(s)$ denote the sum in (1), i.e.,

$$m(s) = \sum_{i \in I} \left(\max_{A \in s} M(A, i) \right) . \tag{2}$$

Table 1 Results of 3 algorithms over 7 instances

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
A_1	1	4	3	4	3	4	3
A_2	1	3	4	3	4	3	4
A_3	3	3	3	3	3	3	3

If $K = 1$ we have:

$$m(\{A_1\}) = \sum_{i \in I} \left(\max_{A \in \{A_1\}} M(A, i) \right) = \sum_{i \in I} M(A_1, i) = 22 \tag{3}$$

Similarly, $m(\{A_2\}) = 22$ and $m(\{A_3\}) = 21$. Thus, according to (1), either $\{A_1\}$ or $\{A_2\}$ is the best 1-portfolio in this case.

If $K = 2$, we have:

$$\begin{aligned} m(\{A_1, A_2\}) &= \sum_{i \in I} \left(\max_{A \in \{A_1, A_2\}} M(A, i) \right) = \\ &= 1 + 4 + 4 + 4 + 4 + 4 + 4 = 25 \end{aligned} \tag{4}$$

and similarly $m(\{A_1, A_3\}) = 24$ and $m(\{A_2, A_3\}) = 24$. Hence, according to (1) the best 2-portfolio is $\{A_1, A_2\}$

Definition 4 (**The K-Algorithms Min-Max-Gap Problem**) An instance of the K-algorithms min-max-gap problem is a K-algorithms cover problem with the following objective O:

$$O \doteq \arg\min_{s \subset S : |s| = K} \left(\max_{i \in I} \left(\min_{a \in s} Gap(a, i, S) \right) \right) \tag{5}$$

Where

$$Gap(a, i, S) = \max_{A \in S}(A, i) - M(a, i) \tag{6}$$

In words, the objective here is to minimize the portfolio's worst gap to the optimal algorithm in S, across all instances.

Example 2 Using Table 1 of Example 1, we demonstrate the meaning of O as defined in (5). Let $m(s)$ denote the internal expression in (5), i.e.,

$$m(s) = \max_{i \in I} \left(\min_{a \in s} Gap(a, i, S) \right) . \tag{7}$$

In Table 2 we show the values of $Gap(a, i, S)$ as computed based on the values of $M(a, i)$ in Table 1, using (6).

Table 2 *Gap*, as defined in (6), with respect to the values in Table 1

	i_1	i_2	i_3	i_4	i_5	i_6	i_7
A_1	2	0	1	0	1	0	1
A_2	2	1	0	1	0	1	0
A_3	0	1	1	1	1	1	1

– For $K = 1$ we have:

$$m(\{A_1\}) = \max_{i \in I}\left(\min_{a \in \{A_1\}} Gap(a, i, S)\right) = \max\{2, 0, 1, 0, 1, 0, 1\} = 2\,. \tag{8}$$

Similarly $m(\{A_2\}) = 2$ and $m(\{A_3\}) = 1$. Hence, according to (5) the best 1-portfolio is $\{A_3\}$.

– For $K = 2$, we have

$$m(\{A_1, A_2\}) = \max_{i \in I}\left(\min_{a \in \{A_1, A_2\}} Gap(a, i, S)\right) = \max\{2, 0, 0, 0, 0, 0, 0\} = 2 \tag{9}$$

and similarly $m(\{A_1, A_3\}) = 1$ and $m(\{A_2, A_3\}) = 1$. Hence, according to (5) either $\{A_1, A_3\}$ or $\{A_2, A_3\}$ is the best 2-portfolio in this case.

4.2 The *K*-Algorithms Max-Sum problem is NP-complete

We will now prove the NP-completeness of the *K-Algorithms Max-Sum Problem (KAMSP)*, by showing that it generalizes the *Maximum Coverage Problem (MCP)* (we will use tagged variables to define the MCP problem):

Definition 5 (**Maximum Coverage Problem (MCP)**) – $S' = \{S'_1, S'_2, \ldots, S'_m\}$ is a finite collection of sets of elements, and

– K' is a positive integer

A solution to *MCP* is a subset $s' \subseteq S'$ of sets, such that $|s'| \leq K'$ and the number of covered elements $\left|\bigcup_{S'_i \in s'} S'_i\right|$ is maximized.

The corresponding decision variant of both problems, has an additional integer, which we will denote by g (for goal) in both cases. The decision variant of *KAMSP* is to find a solution such that $m(s) \geq g$ (see (2)), and the decision variant of *MCP* is to find a solution such that the number of covered elements is greater equal to g. The decision variant of *MCP* is known to be NP-hard [18].

Theorem 1 *The decision variant of KAMSP is NP-Complete.*

Proof We show that *MCP* is a special case of *KAMSP*. Given an instance of *MCP*, we construct an instance of *KAMSP* such that a solution to the latter can be efficiently translated to a solution to the former.

Let e'_j denote the j-th element in $\bigcup_{S'_i \in S'} S'_i$. We construct the following instance of the *KAMSP*:

– $S = S'$, i.e., each subset in the *MCP* is represented by an algorithm in S.
– $I = \bigcup_{S'_i \in S'} S'_i$, i.e., each element in the *MCP* is represented by an instance in I.
– $M(A_i, b_j) = \begin{cases} 1 & \text{if } e'_j \in S'_i \quad \text{for } i \in [1..m],\ j \in \left[1..\left|\bigcup_{S_i \in S} S_i\right|\right]. \\ 0 & \text{otherwise.} \end{cases}$
– $K = K'$, i.e., the bound on the number of algorithms in *KAMSP* is equal to the bound on the number of subsets in the *MCP*.

In addition, g is the same in both problems. A solution to the *KAMSP* is a set s of up to K algorithms, that together have at least g instances with a value of 1. We can translate

this solution to a solution of the corresponding *MCP* in an efficient way: a set S_i' is selected if and only $A_i \in s$. Since each instance b_j in the *KAMSP* corresponds to an element e_j' in the *MCP*, this gives us also a solution to the decision variant of *MCP*. Since the reduction is linear in the size of the *MCP*, we showed that *KAMSP* is NP-hard. Furthermore, since checking that the solution covers at least g elements is also linear, *KAMSP* is in NP, and hence NP-Complete. □

The decision variant of the K-Algorithms Min-Max-Gap problem is also NP-Complete, by a reduction from the set-cover problem (not shown here).

4.3 The *K*-Algorithms Max-Sum problem is approximable

We now prove that this problem is in the APX complexity class, i.e., it is approximable in polynomial time with an approximation ratio bounded by a constant. To that end, we prove that this problem maximizes a function which is *submodular*. Let us recall briefly the definition of such functions.

Definition 6 (**submodular functions**) Let Ω be a finite set. A function f is *submodular* if it is a set function $f : 2^{\Omega} \rightarrow \mathbb{R}$, where 2^{Ω} denotes the power set of Ω, which satisfies the following condition:

For every $X, Y \subseteq \Omega$ with $X \subseteq Y$ and every $x \in \Omega \setminus Y$ we have that

$$f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y) . \tag{10}$$

The importance of proving that our optimization problem maximizes a submodular function, is that all such problems are $1 - [(K-1)/K]^K$-approximable under a cardinality constraint on K [25], i.e., the size of the chosen set is bounded by some constant K. This is exactly the case in the *KAMSP*, because we have a bound on the number of chosen algorithms. In the limit (i.e., when $K = \infty$), the approximation ratio is $1 - 1/e$.

For the following discussion, the finite set Ω is simply our set of algorithms s. Recall that *KAMSP* maximizes a function $F(s)$, where

$$F(s) \doteq \sum_{i \in I} \left(\max_{A \in s} M(A, i) \right) . \tag{11}$$

(note that the right-hand side is a copy of (2)), and we will show that this function is submodular.

As before let I denote the set of instances. For any $i \in I$, let

$$f(s, i) \doteq \max_{A \in s}(M(A, i)) , \tag{12}$$

i.e., the value achieved by the portfolio s on instance i. Hence, we have the equivalence

$$F(s) = \sum_{i \in I} f(s, i) . \tag{13}$$

Theorem 2 *Given two sets of algorithms* s_1, s_2 *such that* $s_1 \subset s_2 \subseteq s$, *and an algorithm* $a \in s \setminus s_2$,

$$F(s_1 \cup a) - F(s_1) \geq F(s_2 \cup a) - F(s_2) . \tag{14}$$

Note that (14) corresponds to (10) in the definition of submodularity (Definition 6).

Proof Falsely assume that the theorem is wrong. Together with (13) this implies that there is at least one instance i such that

$$f(s_1 \cup a, i) - f(s_1, i) < f(s_2 \cup a, i) - f(s_2, i) \ . \tag{15}$$

We now split the proof to two cases, corresponding to whether a is better than s_1's algorithms on i, starting with the case that it is not:

1. $f(s_1 \cup a, i) - f(s_1, i) = 0$. Since $s_1 \subset s_2$, then also $f(s_2 \cup a, i) - f(s_2, i) = 0$ and hence the inequality of (15) is false, which is a contradiction.
2. $f(s_1 \cup a, i) - f(s_1, i) > 0$. Here we again split to two cases.
   (a) $f(s_2 \cup a, i) - f(s_2, i) = 0$, i.e., algorithm a does not contribute to s_2 on instance i. In this case the right-hand side of (15) is equal to 0. But since max is a monotone function, the left-hand side of (15) is non-negative, which is a contradiction.
   (b) $f(s_2 \cup a, i) - f(s_2, i) > 0$, i.e., algorithm a contributes to s_2 on instance i. Since f is a max function (see (12)) and a is better than s_1's algorithms on i, then

$$f(s_1 \cup a, i) = f(a, i) \ , \tag{16}$$

   and since we are now considering the case that a is also better than s_2's algorithms on i, we have

$$f(s_2 \cup a, i) = f(a, i) \ . \tag{17}$$

   Rewriting (15) based on (16) and (17) yields:

$$f(a, i) - f(s_1, i) < f(a, i) - f(s_2, i) \ , \tag{18}$$

   which is equivalent to

$$f(s_1, i) > f(s_2, i) \ . \tag{19}$$

   However, since $s_1 \subset s_2$,

$$f(s_1, i) \leq f(s_2, i) \ , \tag{20}$$

   which contradicts (19).

Hence, in all cases our false assumption (15) leads to a contradiction. □

5 Modeling the *K*-Algorithms cover problem with SMT

The Satisfiability Modulo Theories problem (SMT) [19] is to decide the satisfiability of a first-order formula over some decidable theories. In the last decade it has become a competitive alternative to more traditional optimization frameworks like mixed-integer programming, and generally contains a more general modeling language and is especially useful for analysis in the presence of a rich Boolean structure. Our algorithms-cover problem can be modeled with a theory called *Quantifier-Free Linear Real Arithmetic* (QF_LRA) [7]. This theory supports formulas that are a Boolean combination of linear inequalities over the reals, e.g.,

$$F = (x \geq -2) \vee (y \geq -1 \wedge y \leq 5) \ ,$$

where $x, y \in \mathbb{R}$.

The following encoding is the SMT representation of the K-Algorithms Max-Sum Problem.

– *Real variables:* V_i for $i \in I$, which represent the quality of the portfolio over instance i.

- *Boolean decision variables:* A_i for $i \in [1..n]$, which represent whether algorithm A_i is chosen for the optimal K-portfolio.
- *objective:* Maximize the sum of qualities across the instances:

$$\max \sum_{i=1}^{|I|} V_i \ . \tag{21}$$

- *Constraints:* (22) – (24) presented below are connected by a conjunction.
  The *value choice constraints* define the quality allowed for each instance:

$$\forall i \in I : \bigvee_{j=1..n} V_i = M(A_j, i) \ . \tag{22}$$

  The *implied algorithm constraints* connect between the chosen quality of an instance and the algorithms that return this quality:

$$\forall i \in I, V \in \{M(A_j, i)\}_{j \in \{1..n\}} : (V_i = V) \rightarrow \bigvee_{A \in S : M(A,i)=V} A \ . \tag{23}$$

  The *algorithms cardinality constraints* ensure that the number of chosen algorithms of a portfolio will not exceed K, when we take *true=1* and *false=0*:

$$\sum_{i=1}^{n} A_i \leq K \ . \tag{24}$$

  A constraint such as (24) is not directly allowed in QF_LRA, but it can be reduced to propositional logic, which *is* supported by QF_LRA. We used the encoding suggested in [29] for this purpose.

A similar SMT encoding was used for the modeling of the K-Algorithms Min-Max-Gap.

A modeling tool Based on the above modeling, we implemented[3] an SMT modeling program with the following interface:

- Input:
  - A matrix M, where $M[i, j]$ represents the solution quality of algorithm A_i, for $i \in [1..n]$,
  - A number $K < n$ of algorithms to choose.
- Output:
  - SMT encoding of the K-Algorithms Max-Sum problem, as described above.

6 Empirical results: portfolios

We built three portfolios, where K, the number of cores, is in the range [1..24]:

[3]The code of the SMT modeling tool, with the regular and hard inputs, can be found in https://doi.org/10.5281/zenodo.3841422

1. *Optimal* – Uses the SMT encoding and the tool that we described in the previous section, and solves the resulting model with the SMT solver *Z3* [24]. *Z3* is a high-performance solver developed by Microsoft Research. It is mostly used in software verification applications, e.g., [3, 20, 30].
2. *Greedy* – Chooses the best algorithm to complete a partial portfolio, for K iterations. More specifically, in each round we choose the algorithm that improves the portfolio the most, assuming that all the previous choices of algorithms stay in the portfolio. In the end of step i, $i < K$, we remove the results of the algorithms that we chose in this step. In step $i + 1$ we then check which algorithm that was not yet chosen, improves by the largest amount the results of the portfolio we had in step i. We recall that according to our proof in Section 4.3 and a result in [25], this greedy algorithm has an approximation ratio of $1 - [(K - 1)/K]^K$.
3. *K-Best* – Sorts the algorithms by non-ascending quality (according to the average score of each algorithm over all instances), and takes the first K algorithms.

The K-Best portfolio does not use the information about the quality of each algorithm per-instance, thus we expect it to be inferior to the two other portfolios. Obviously any method of portfolio construction eventually reaches the optimum if $K = n$. The results of the three portfolios and the Max-Sum objective are shown in Fig. 4 for the 'normal' batch, and in Fig. 5 for the 'hard' instance set (recall that the difference between these two sets is that in the latter set of instances there are many more drones to coordinate). Similar results for the Min-Max-Gap objective are shown in Fig. 6 (normal) and Fig. 7 (hard). It turns out that the optimal portfolio in most cases is only marginally better than what can be achieved with the greedy method described above. This means that in practice the results of the greedy approach is better than the worst-case discussed in Section 4.3. This does not nullify the value of choosing an optimal portfolio in general, because the process of finding the optimal portfolio, although computationally more expensive than the greedy method, is executed once and produces a portfolio that should at least in some cases be better, whereas the

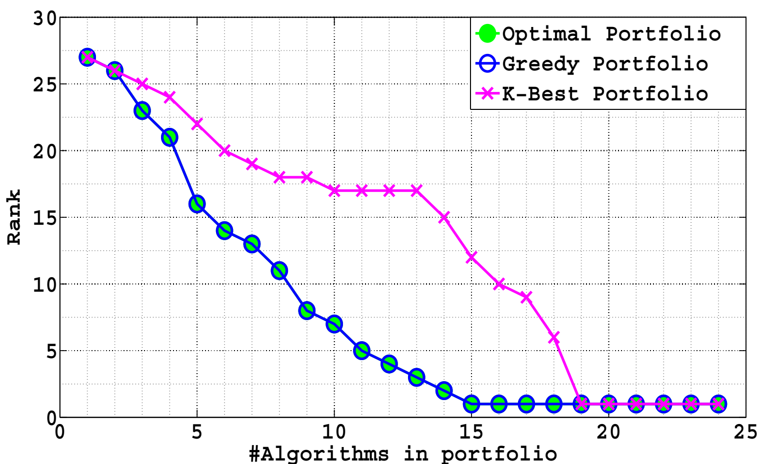

Fig. 4 Ranking of max-sum portfolios for the normal instances. The optimal and greedy portfolio coincide with a better ranking than the K-best portfolio between portfolio sizes of 3 to 19. The optimal and greedy portfolio reach the maximum quality with 15 portfolio members. The K-best portfolio reach the same quality with 19 portfolio members

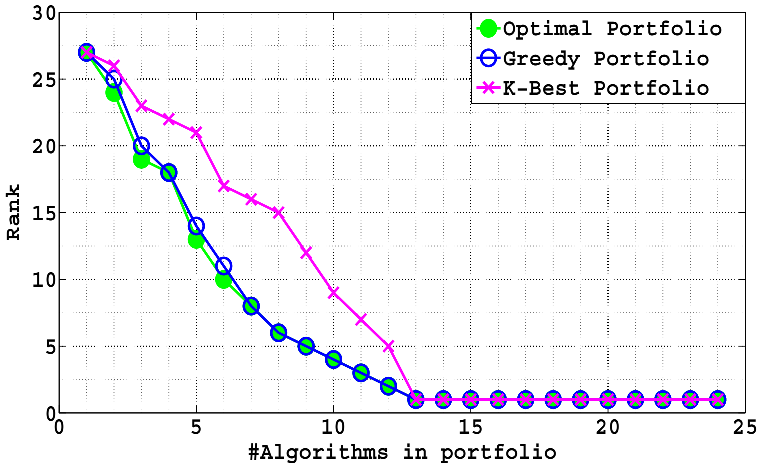

Fig. 5 Ranking of max-sum portfolios for the hard instances set. The optimal and greedy portfolios have the same ranking, except in portfolio sizes of 2,3,5,6, where the optimal is better than the greedy. The K-best portfolio is inferior to the others but reaches the maximum quality with 13 portfolio members, together with the optimal and greedy portfolios

portfolio itself is used many times. In each figure, the Y axis describes the global ranking of each portfolio. More details appear in the caption of the figures.

7 Conclusions

We studied various aspects of attempting to solve computationally hard optimization problems in real-time. As a baseline, we presented an empirical evaluation of the success of

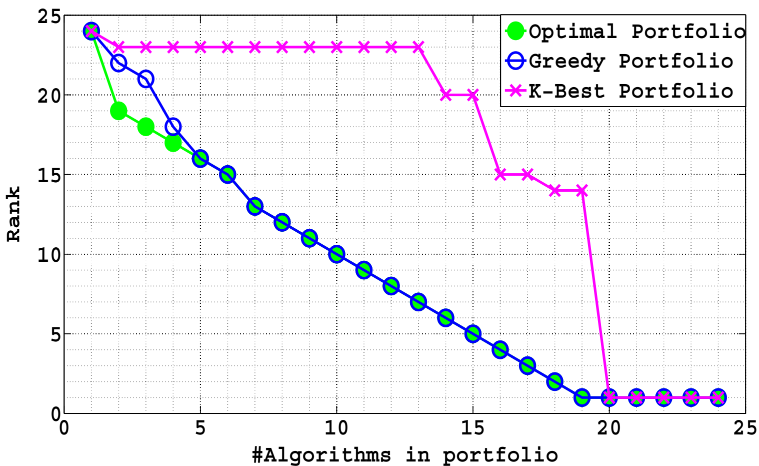

Fig. 6 Ranking of min-max-gap portfolios for the normal instances. The optimal portfolio is superior up to 4 members, then it is identical to the greedy portfolio. There is a long stagnation in the negative ranking of the K-best portfolio, and it reaches a zero gap at the size of 20, just after the optimal and greedy portfolios reach the maximum

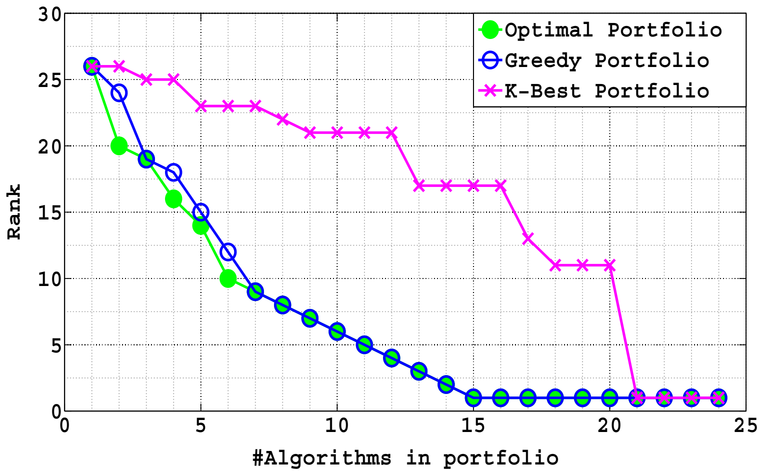

Fig. 7 Ranking of min-max-gap portfolios for the hard instances set. The optimal portfolio is better than the greedy portfolio at sizes 2,4,5,6, and they both reach the a zero gap at size 15. The K-best is the worst portfolio and reaches zero gap at 21 members

various known algorithms in solving such problems, after tuning them automatically to this timeout. But the main contribution of this article is a solution to the problem of choosing the optimal parallel portfolio of algorithms, given a number K of available computing cores, and also the related problem of finding the minimum number of cores that is necessary for achieving the same result. We proved the complexity of this problem and showed that it is approximable.

We believe that future research should focus on testing the performance of other algorithms with such short time-outs, and moreover on finding ways to tune them for this specialized setting, which, to the best of our knowledge, was not studied in the literature until now.

The long version of this work appears in a thesis [26]. It additionally includes the following results:

- A formal definition of the checked algorithms;
- A formal definition of the various constraint optimization problems that we used for evaluation;
- A proof that the decision variant of the optimization problem that was discussed and experimented with is NP-complete;
- Detailed results before and after automatic tuning, for two levels of problem hardness;
- Detailed results of using the above best portfolio.

References

1. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: International Conference on Principles and Practice of Constraint Programming, pp. 142–157. Springer, Berlin (2009)
2. Brooks, S.H.: A discussion of random methods for seeking maxima. Oper. Res. **6**, 244–251 (1958)

3. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a Practical System for Verifying Concurrent C, pp. 23–42. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-03359-9_2
4. Crescenzi, P., Kann, V.: A compendium of NP optimization problems. In: WWW Spring 1994 (1994)
5. Dechter, R.: Constraints Processing. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Burlington (2003)
6. Di Gaspero, L., Schaerf, A.: Easylocal++: an object-oriented framework for flexible design of local search algorithms. Software — Practice & Experience **33**(8), 733–765 (2003)
7. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4144, pp. 81–94. Springer (2006). https://doi.org/10.1007/11817963_11
8. Forrest, S., Mitchell, M.: Relative Building-Block fitness and the Building-Block hypothesis. Foundations of Genetic Algorithms **2**, 109–126 (1993)
9. Glover, F.: Future paths for integer programming and links to artificial intelligence. Comput. Oper. Res. **13**(5), 533–549 (1986)
10. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1-2), 43–62 (2001)
11. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge (2005)
12. Hoos, H., Leyton-Brown, K., Schaub, T., Schneider, M.: Algorithm configuration for portfolio-based parallel sat-solving. In: Workshop on Combining Constraint Solving with Mining and Learning (2012)
13. Hoos, H.H., Stutzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann Massachusetts, Burlington (2004)
14. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296), 51–54 (1997)
15. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, pp. 507–523. Springer (2011)
16. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009)
17. Kirkpatrick, S., Gelatt, C.D. Jr.., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
18. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations. Springer, US (1972)
19. Kroening, D., Strichman, O.: Decision Procedures: an Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2010). http://www.decision-procedures.org
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning, pp. 348–370. Springer (2010)
21. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)
22. Malitsky, Y., Sellmann, M.: Instance-specific algorithm configuration as a method for non-model-based portfolio generation. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimzation Problems.**7298** 244–259
23. Michel, L., Van Hentenryck, P.: Localizer a modeling language for local search. In: Smolka, G. (ed.) Principles and Practice of Constraint Programming-CP97, pp. 237–251. Springer, Berlin (1997)
24. de Moura, L., Bjørner, N.: Z3: an Efficient SMT Solver, pp. 337–340. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Nemhauser, G.L., Wolsey, L.A., Fisher, M.L.: An analysis of approximations for maximizing submodular set functions - I. Math. Program. **14**(1), 265–294 (1978)
26. Nof, Y.: Real time solving of discrete optimization problems. Master's thesis, Technion, Israel Institute of Technology. Available online in https://ie.technion.ac.il/~ofers/publications/theses/yair_nof.pdf (2017)
27. Petrik, M., Zilberstein, S.: Learning parallel portfolios of algorithms. Ann. Math. Artif. Intell. **48**(1), 85–106 (2006)
28. Rubinstein, R., Kroese, D.: The Cross-Entropy Method: a Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning. Springer, New York (2004)
29. Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints. CP **3709**, 827–831 (2005)

30. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, pp. 61–72. ACM (2010)
31. Williamson, D.P., Shmoys, D.B.: The Design of Approximation Algorithms. Cambridge University Press, Cambridge (2011)
32. Yang, X.S. Nature-Inspired Metaheuristic Algorithms, 2nd edn., pp. 12–13. Luniver Press, United Kingdom (2010)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.