Optimal algorithm portfolios for computationally hard real-time problems

Yair Nof, Ofer Strichman
Information Systems Engineering, IE, Technion, Haifa, Israel.
yair.nof@gmail.com, ofers@ie.technion.ac.il

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 benchmarks).

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 benchmarks. As benchmarks 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.

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 and hence finding the best move under such hard constraints is an NP-hard problem. Another example is the Weapon-Target Assignment Problem [2], where the best assignment of weapons has to be found, so as to maximize the probability of destroying the targets. It is rather obvious that there are scenarios in which such decisions should be made in real time.

A third 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 (not to collide) and to soft objectives, such as minimizing the time to complete the mission. Here again the decision problem is NP-hard.

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; the targets in case of the weapon assignment; and 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) [27], which can guarantee a $1 - \epsilon$ proximity to the optimal solution, such solutions are generally irrelevant because they can exceed the hard time bound $T$. Moreover, the run-time increases when the precision parameter $\epsilon$ 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); 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), 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.

\(^1\)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.
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:

**Definition 1.1 (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 $\text{soft}(P)$ 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 $\text{FT}(P)$, is the problem of finding a solution within time $T$ to $\text{soft}(P)$.

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. When the candidate solutions are sampled not necessarily following the neighboring relation, the search is not considered an optimal solution is found or the time bound is reached. The

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 problem 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 benchmarks, 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 §3 we describe empirical evidence of the relative performance of the various algorithms.

In §4 and §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 §6 we describe empirical results that compare parallel portfolios that were constructed in different ways, and we conclude in §7. All the experiments in this article are based on benchmarks 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 benchmarks, in two batches: ‘normal’, and ‘hard’, the latter having the property of having a much larger number of UAVs to coordinate.

## 2 The algorithms that we checked

A detailed description of the algorithms mentioned below appear in the first author’s thesis [1]. 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 decision and 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 for implementing them, e.g., LOCALIZER [22], EASYLOCAL++ [7], 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)** [28], **Stochastic Hill-Climbing (SHC)** [8], **Tabu Search (TS)** [9] and **Simulated Annealing (SA)** [17] are such local search 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)** [4], and **Cross Entropy Method (CE)** [24] are such non-local search algorithms that we experimented with, although they are still stochastic. We also experimented with a **Greedy** algorithm and with combining it with the other algorithms (this turned out to be a winning strategy as we will see).
Our 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.

### 2.3 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. The Value function is tailored for the specific COP. 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 Def. 1.1).

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

```plaintext
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
```

### Figure 1: Quality of various algorithms, after tuning for $T = 0.1$ sec, over 500 benchmarks (the ‘normal’ benchmarks)

#### 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 §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 [3], [20], [15], 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 benchmarks. 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 benchmarks. 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 problems the hybrid algorithms still lead the chart, although, as expected, there is some differences in the relative competitiveness of the algorithms – see Fig. 2.
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 [23] 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 that can adapt per input [21]. 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 was 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, start with an empty portfolio and tune one algorithm instance at a time, to find the best parameters we can, in order 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 were already tuned. We assume that the algorithms do not communicate during their run. The problem that we solve is to choose the algorithms, given that we do not have enough cores to run all of them. The choice is based on a training set, and the hope is that optimizing against this set will prove itself effective also with respect to future benchmarks. We proceed with formal definitions of two variants of this problem.

4.1 The $K$-Algorithms Cover Problems

Definition 4.1 (The $K$-Algorithms Cover Problem) An instance of the $K$-algorithms cover problem is a 5-tuple $(S, I, M, m, K)$, where:

- $S$ is a set of $n$ algorithms,
- $I$ is a set of inputs for $S$,
- $M : S \times I \mapsto \mathbb{R}$ is the quality of solution that each algorithm in $S$ returns with each input in $I$,
- $m_1 : \mathcal{P}(S) \mapsto \mathbb{R}$ is a portfolio measure over $I$, of a parallel portfolio $s \subseteq S$, which satisfies the property $m_1(s) = \max_{i \in I} \{M(A, i)\}$, for $i \in I$, $A \in S$, and finally
- $K < n$ is the number of algorithms to choose from $S$.

A solution to the instance $(S, I, M, m, K)$ is a parallel portfolio of $K$ algorithms from $S$ with the best performance, according to $m_1$:

$$s^* = \arg\max_{s \subseteq S : |s| = K} (m_1(s))$$  \hspace{1cm} (1)

Definition 4.2 (K-Algorithms Max-Sum Problem) An instance of the $K$-algorithms max-sum problem is an instance of the $K$-algorithms cover problem with the following portfolio measure, where $s \subseteq S$:

$$m_1(s) = \sum_{i \in I} \left( \max_{A \in s} M(A, i) \right)$$ \hspace{1cm} (2)

In words, the objective of a $K$-algorithms cover problem with this measure is to maximize the portfolio’s sum of qualities across benchmarks.
Example 4.3 Let \((S, I, M, k)\) 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:

<table>
<thead>
<tr>
<th></th>
<th>(i_1)</th>
<th>(i_2)</th>
<th>(i_3)</th>
<th>(i_4)</th>
<th>(i_5)</th>
<th>(i_6)</th>
<th>(i_7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1)</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>(A_2)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>(A_3)</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

\(m_I(s)\) is given in (2).

If \(K = 1\) we have:

\[
m_I(\{A_1\}) = \sum_{i \in I} \left( \max_{A \in \{A_1\}} M(A, i) \right) = \sum_{i \in I} M(A_1, i) = 22
\]

(3)

\[
m_I(\{A_2\}) = \sum_{i \in I} \left( \max_{A \in \{A_2\}} M(A, i) \right) = \sum_{i \in I} M(A_2, i) = 22
\]

(4)

\[
m_I(\{A_3\}) = \sum_{i \in I} \left( \max_{A \in \{A_3\}} M(A, i) \right) = \sum_{i \in I} M(A_3, i) = 21
\]

(5)

Thus \(\{A_1\}\) or \(\{A_2\}\) is the best 1-portfolio in this case.

If \(K = 2\), we have:

\[
m_I(\{A_1, A_2\}) = \sum_{i \in I} \left( \max_{A \in \{A_1, A_2\}} M(A, i) \right) = 1 + 4 + 4 + 4 + 4 + 4 + 4 = 25
\]

(6)

\[
m_I(\{A_1, A_3\}) = \sum_{i \in I} \left( \max_{A \in \{A_1, A_3\}} M(A, i) \right) = 3 + 4 + 3 + 4 + 3 + 4 + 3 = 21
\]

(7)

\[
m_I(\{A_2, A_3\}) = \sum_{i \in I} \left( \max_{A \in \{A_2, A_3\}} M(A, i) \right) = 3 + 3 + 4 + 3 + 4 + 4 + 4 = 24
\]

(8)

Thus \(\{A_1, A_2\}\) is the best 2-portfolio in this case.

5 Modeling the \(K\)-Algorithms Cover Problem with SMT

The Satisfiability Modulo Theories problem (SMT) [18] 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 better 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). This theory supports formulas that are a Boolean combination of linear inequalities over the reals, e.g.,

\[
F = (x \geq -2) \lor (y \geq -1 \land y \leq 5),
\]

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 benchmark \(i\).
- **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 benchmarks:

\[
\max \sum_{i=1}^{|I|} V_i.
\]

(9)

- **Constraints:** (10)-(12) below are connected by a conjunction.

The value choice constraints defines the allowed quality for each benchmark:

\[
\forall i \in I : (V_i = M(A_1, i)) \lor \ldots \lor (V_i = M(A_n, i)).
\]

(10)

The implied algorithm constraints connect between the chosen quality of a benchmark and the algorithms that return this quality:

\[
\forall i \in I, V \in \{M(A_j, i)\}_{j \in \{1..n\}} : (V_i = V) \to \bigvee_{A \in S : M(A, i) = V} A.
\]

(11)

The algorithms cardinality constraints ensure that the number of chosen algorithms of a portfolio will be \(K\), when we take \(true = I\) and \(false = 0\):

\[
\sum_{i=1}^n A_i \leq k.
\]

(12)

A constraint such as (12) is not allowed in QF_LRA, but it can be reduced to propositional logic, which is allowed in QF_LRA. We used the encoding suggested in [25] for this purpose.

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

- **Input:**
  - A matrix \(M\), where \(M[i, j]\) represents the quality of the solution that 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]:
1. **Optimal** – Uses the SMT encoding and tool that we described in the previous section, and solves the resulting model with the SMT solver Z3 [6]. Z3 is a high-performance solver developed by Microsoft Research. It is mostly used in software verification applications, e.g., [5], [19], [26].

2. **Greedy** – Chooses the best algorithm to complete a partial portfolio, for $K$ iterations

3. **K-Best** – Sorts the algorithms by quality, 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 are shown in Fig. 4 for the ‘normal’ batch, and in Fig. 5 for the ‘hard’ benchmark set. It turns out that the optimal portfolio in both cases is only marginally better than what can be achieved with the greedy method described above. 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 portfolio itself is used many times.

In both figures the sum, which has to be maximized, is divided by the best single algorithm’s quality, and this ratio is what annotates the $Y$ axis. More details appear in the caption of the figures.

---

**Figure 4**: Max-Sum portfolios. The optimal portfolio reaches its maximum improvement of about 2% at a portfolio size of 17, with a decreasing rate of improvement. This does not guarantee convergence (convergence is guaranteed only when a partial portfolio reaches the quality of the full portfolio). Since the greedy portfolio’s quality is almost identical to the optimal portfolio, we show them together. The $K$-Best portfolio’s quality is lower, as expected, and reaches the optimum quality at a portfolio size of 19.

**Figure 5**: Max-Sum portfolios for hard problems. The optimal portfolio reaches its maximum improvement of 11% at a portfolio size of 15. This is not apparent from the graph but can be seen in the numerical results. The 11% improvement is higher than the improvement we saw in the portfolio over general problems. Again, the greedy portfolio almost coincides with the optimal portfolio, and the $K$-best portfolio is inferior.

---

**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 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.

The long version of this work appears in a thesis [1]. 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;
- Modeling of the best-portfolio problem with a min-max criterion, i.e., choose a portfolio whose maximum gap from the optimum is minimal;
- Detailed results of using the above best portfolio.

The benchmark generator is publicly available, and we hope that future research will use it to examine the capability of other algorithms in solving this hard and important problem.
References


