# Learning a Reactive Restart Strategy to Improve Stochastic Search

Serdar Kadioglu[1], Meinolf Sellmann[2], and Markus Wagner[3]

[1] Department of Computer Science, Brown University, RI, USA
serdark@cs.brown.edu,
[2] Cortlandt Manor, NY, USA
meinolf@gmail.com,
[3] Optimisation and Logistics, The University of Adelaide, SA, Australia
markus.wagner@adelaide.edu.au

**Abstract.** Building on the recent success of bet-and-run approaches for restarted local search solvers, we introduce the idea of learning online adaptive restart strategies. Universal restart strategies deploy a fixed schedule that runs with utter disregard of the characteristics that each individual run exhibits. Whether a run looks promising or abysmal, it gets run exactly until the predetermined limit is reached. Bet-and-run strategies are at least slightly less ignorant as they decide which trial to use for a long run based on the performance achieved so far. We introduce the idea of learning fully adaptive restart strategies for black-box solvers, whereby the learning is performed by a parameter tuner. Numerical results show that adaptive strategies can be learned effectively and that these significantly outperform bet-and-run strategies.

**Keywords:** Restart strategies, adaptive methods, parameter tuning

## 1 Introduction

Restarted search has become an integral part of combinatorial search algorithms. Even before heavy-tailed runtime distributions were found to explain the massive variance in search performance [1], in local search restarts were commonly used as a search diversification technique [2].

Fixed-schedule restart strategies were studied theoretically in [3]. For SAT and constraint programming solvers, practical studies followed. For example, one study found that there is hardly any difference between theoretically optimal schedules and simple geometrically growing limits [4]. SAT solvers used geometrically growing limits for quite some time before the community largely adapted theoretically optimal schedules (whereby the optimality guarantees are based on assumptions that actually do not hold for clause-learning solvers where consecutive restarts are not independent). Audemard and Simon [5] argued that fixed schedules are suboptimal for SAT solvers and designed adaptive restarts strategies for one SAT solver specifically.

In this paper, we describe a general methodology for embedding any black-box optimization solver into an adaptive stochastic restart framework. The framework monitors certain key performance metrics that are based on the evolution of the objective function values of the solutions found. Based on these observations, the method then adaptively computes scores that affect the likelihood whether we continue the current run beyond the original limit, whether we start a new run, or whether we continue the best run so far. We employ an automatic parameter tuner to learn how to adapt these probabilities dependent on the observed performance metrics.

In the following, we recap the idea of bet-and-run strategies. We continue with reviewing the recently introduced idea of hyper-parameterizing local search solvers to achieve superior online adaptive behavior. We then introduce the idea of using automatic hyper-reactive search tuning for learning adaptive restart strategies. Finally, we present experimental results that clearly show that adaptive search significantly outperforms bet-and-run strategies.

## 2 Restart Strategies

Nowadays, stochastic search algorithms and randomized search heuristics are frequently restarted: If a run does not conclude within a pre-determined limit, we restart the algorithm. This was shown to to help avoid heavy-tailed runtime distributions [1]. Due to the added complexity of designing an appropriate restart strategy for a given target algorithm, the two most common techniques used are to either restarts with a certain probability at the end of each iteration, or to employ a fixed schedule of restarts.

Some theoretical results exist on how to construct optimal restart strategies. For example, Luby et al. [3] showed that, for Las Vegas algorithms with known run time distribution, there is an optimal stopping time in order to minimize the expected running time. They also showed that, if the distribution is unknown, there is a universal sequence of running times which is the optimal restarting strategy up to constant factors.

Fewer results are known for the optimization case. Marti [6] and Lourenco et al. [7] present practical approaches, and a recent theoretical result is presented by Schoenauer et al. [8]. Particularly for the satisfiability problem, several studies make an empirical comparison of a number of restart policies [9, 10].

Quite often, classical optimization algorithms are deterministic and thus cannot be improved by restarts. This also appears to hold for certain popular modern solvers, such as IBM ILOG CPLEX. However, characteristics can change when memory constraints or parallel computations are encountered. This was the initial idea of Lalla-Ruiz and Voß [11], who investigated different mathematical programming formulations to provide different starting points for the solver.

Many other modern optimization algorithms, while also working mostly deterministically, have some randomized component, for example by choosing a random starting point. Two very typical uses for an algorithm with time budget $t$ are to (a) use all of time $t$ for a single run of the algorithm (single-run strategy), or (b) to make a number of $k$ runs of the algorithm, each with running time $t/k$ (multi-run strategy).

Extending these two classical strategies, Fischetti et al. [12] investigated the use of the following BET-AND-RUN strategy with a total time limit $t$:

**Phase 1** performs $k$ runs of the algorithm for some (short) time limit $t_1$ with $t_1 \leq t/k$.

**Phase 2** uses remaining time $t_2 = t - k \cdot t_1$ to continue *only the best run* from the first phase until timeout.

Note that the multi-run strategy of restarting from scratch $k$ times is a special case by choosing $t_1 = t/k$ and $t_2 = 0$ and the single-run strategy corresponds to $k = 1$; thus, it suffices to consider different parameter settings of the *bet-and-run* strategy to also cover these two strategies.

Fischetti et al. [12] experimentally studied such a BET-AND-RUN strategy for mixed-integer programming. They explicitly introduce diversity in the starting conditions of the used MIP solver (IBM ILOG CPLEX) by directly accessing internal mechanisms. In their experiments, $k = 5$ performed best.

Recently, Friedrich et al. [13] investigated a comprehensive range of BET-AND-RUN strategies on the traveling salesperson problem and the minimum vertex cover problem. Their best strategy was $\text{RESTARTS}^{40}_{1\%}$, which in the first phase does 40 short runs with a time limit that is 1% of the total time budget and then uses the remaining 60% of the total time budget to continue the best run of the first phase. They investigated the use of the universal sequence of Luby et al. [3] as well, using various choices of $t_1$, however, it turned out inferior.

The theoretical analysis is provided by Lissovoi et al. [14], who investigated BET-AND-RUN for a family of pseudo-Boolean functions, consisting of a plateau and a slope, as an abstraction of real fitness landscapes with promising and deceptive regions. The authors showed that BET-AND-RUN with non-trivial $k$ and $t_1$ are necessary to find the global optimum efficiently. Also, they showed that the choice of $t_1$ is linked to properties of the function, and they provided a fixed budget analysis to guide selection of the bet-and-run parameters to maximise expected fitness after $t = k \cdot t_1 + t_2$ fitness evaluations.

The goal of our present research is to address the two challenges encountered in previous works: the need to set $k$ and $t_1$ in case of BET-AND-RUN, and the general issue of inflexibility in previous approaches. Our framework can decide online whether (i) the current run should be continued, (ii) the best run so far should be continued, or (iii) a completely new run should be started.

## 3 Learning Dynamic Parameter Updates

Our objective is to provide a generic framework for making restart strategies adaptive for any optimization solver. To this end we build on the idea to use parameter tuners for training adaptive search strategies [15, 16] and a recently proposed approach for constructing a hyper-reactive dialectic search solver [17].

In [17], an existing local search meta-heuristic called dialectic search [18] was modified in such a way that the search decisions (when and how much to diversify, how strongly to intensify, when to restart, etc) were taken with regard to the way how the optimization was observed to progress. In essence, the solver tracked features of the optimization process itself and then tied these to decisions (such as: which percentage of variables to modify to generate a new start point) via logistic regression functions. The weights of these functions, one for each meta-heuristic search decision, then became the hyper-parameters of the solver.

Key to making this work in practice is an effective method for learning the weights in the logistic regression functions. Since the only immediately meaningful observation is the overall performance of the solver, parameter tuner GGA [19] was used to "learn" which weights result in good performance.

## 4 A Hyper-Parameterized Restart Strategy

We now combine the two core ideas presented above. Namely, the idea of considering a batch of runs with the option to continue some of them, and the idea to automatically learn which run to continue or whether to start a new run based on the observed performance characteristics of past runs.

The first ingredient we need are features that somehow give us an idea of the big picture of what is going on when tackling the instance at hand.

### 4.1 Features

Whenever a restart decision has to be made, we have three options. We can either continue the current run, we can continue the best run so far, or we can start a completely new run. For each of these options we essentially track two values: The first tells us how good each run looked initially, the second what the trajectory looks like for making further progress.

For the current run and the best run so far, we record their best objective function found after the initial limit. For the new run option, we track how well any new run did after the initial limit and compute the running average.

For the trajectory of the current and the best run, we extrapolate the performance improvement achieved between the best solution found in the initial run and the best performance achieved so far. The extrapolation point is the end of the remaining time we have for the optimization.

For the new runs, to get an estimate how well we might do if (from now until the overall time limit is reached) all we did was run new runs, we consider the standard deviation in objective function performance. Then, we estimate the trajectory as the average minus the standard deviation times the square root of two times the logarithm of the number of new runs we can still afford to conduct. While not exact, this is a lower bound for the minimum of repeated stochastic experiments for which all we know are the mean and the standard deviation [20].

We thus compute six data points that we can use to decide whether to continue the current run, give the current best run more time, or start a completely new run. One complication arises. Namely, for different instances, the objective function values observed will generally operate on vastly different scales. However, to learn strategies offline, we need to compute weights, and these need to work with all kinds of instances. Consequently, rather than taking average and projected objective function values at face value, we first normalize them.

In particular, we consider the three initial values (best found in initial time interval for current and best, and running average of best found for all new runs) and normalize them between 0 and 1. That is, we shift and scale these values in such a way that their smallest will be 0, the largest will be 1, and the last will be somewhere between 0 and 1. Analogously, we normalize the trajectory values.

On top of the six features thus computed we also use the percentage of overall time that has already elapsed, the percentage of overall time afforded in the beginning where all we do is restart a new run every time, and the time a new run will be given as percentage of total time left. In total we thus arrive at nine features.

### 4.2 Turning Features Into Scores

Now, to compute the score for each of the three possibilities (continue current run, continue best run so far, and start a new run) we compute the following function

$$p^k(f) \leftarrow \frac{1}{1 + \exp(w_0^k + \sum_i f_i w_i^k)},$$

whereby $k \in \{1, 2, 3\}$ marks whether the function marks the score for continuing the current run, continuing the best run, or starting a new run, and $f \in R^9$ is the feature vector that characterizes our search experience so far. Note that $p^k(f) \in (0, 1)$, whereby the function approaches 0 when the weighted sum in the denominators exponential function goes to infinity, and how the function approaches 1 when the same sum approaches minus infinity. Finally, note that we require a total of 30 weights to define the three functions. These weights will be learned later by a parameter tuner to achieve superior runtime behavior.

### 4.3 The Reactive Restart Framework

Given the weights $w_i^k$ with $k \in \{1, 2, 3\}$ and $i \in \{0, \ldots, 9\}$, we can now define the framework within which we can embed any black-box optimization solver.[4]

We present a stylized version of our framework in Algorithm 1. Given are a randomized optimization algorithm $S$, an input x, a timeout, a fraction $k \in [0, 1]$, a factor $r \geq 1$, and weights $w$. The framework first runs $S$ on x until a first solution is computed. It records the time to find this solution and sets the incremental time interval each run is given to $r$ times this input-dependent value. Next, $S$' run on x is continued until this incremental time interval is reached. The best solution seen so far is recorded.

[4] We say black-box because we do not need to know anything about the inner workings of the solver. However, we make two assumptions. First, that we can set a time limit to the solver where it stops, and that we can add more time and continue the interrupted computation later. Second, that whenever the solver stops it returns information when it found the first solution, when it found the best solution so far, and what the quality of the best solution found so far is.

**Algorithm 1** Reactive Restart Framework Algorithm

```
1:  function ReactiveRestarts ($S$,x,timeout,$k$,$r$,$w_{i\in\{0,\ldots,9\}}^{k\in\{1,2,3\}}$)
2:      (initTime, best) $\leftarrow S(\text{x}, \textit{newRun}, \textit{stopAtFirstSolution})$
3:      interval $\leftarrow r \times$ initTime
4:      $b \leftarrow S(\text{x}, \textit{continueLastRun}, \text{interval} - \text{initTime})$
5:      elapsedTime $\leftarrow$ interval
6:      Update(best, $b$)
7:      while elapsedTime $\leq k \times$ timeout do
8:          $(a, b) \leftarrow S(\text{x}, \text{interval})$
9:          elapsedTime $\leftarrow$ elapsedTime + interval
10:         Update(initTime, best, $a$, $b$)
11:         interval $\leftarrow r \times$ initTime
12:     Init($F$)
13:     while elapsedTime $\leq$ timeout do
14:         $p^k \leftarrow \frac{1}{1+\exp(w_0^k + \sum_i w_i^k F_i)} \quad \forall k \in \{1, 2, 3\}$
15:         $p^k \leftarrow \frac{p^k}{p^1+p^2+p^3} \quad \forall k \in \{1, 2, 3\}$
16:         pick random $x \in [0, 1]$
17:         if $x \leq p^1$ then
18:             $b \leftarrow S(\text{x}, \textit{continueBestRun}, \text{interval})$
19:             elapsedTime $\leftarrow$ elapsedTime + interval
20:             Update(best, $b$)
21:         else if $x \leq p^1 + p^2$ then
22:             $b \leftarrow S(\text{x}, \textit{continueLastRun}, \text{interval})$
23:             elapsedTime $\leftarrow$ elapsedTime + interval
24:             Update(best, $b$)
25:         else
26:             $(a, b) \leftarrow S(\text{x}, \textit{newRun}, \text{interval})$
27:             elapsedTime $\leftarrow$ elapsedTime + interval
28:             Update(initTime, best, $a$, $b$)
29:             interval $\leftarrow r \times$ initTime
30:         Update($F$)
31:     return best
```

Now, the first phase begins, which lasts for the fraction of the total time allowed as specified by $k$. In this first phase, we start a new run on x every single time, whereby we update the best solution seen so far and the time it takes each time to find a first solution. The function Update is assumed to record the best solution quality found so far as well as to maintain the running average of the time it took to compute a first solution for each new run.

After the first phase ends, we initialize the features based on the search experience so far. Then, we enter the main phase. Based on the given weights and the current features we compute scores for the three options how we can continue the computation at each step. We then choose randomly and proportionally to these scores whether we continue the best run so far, the last run, or whether we begin a new run.

No matter which choice we always keep the best solution found so far up to date. When we choose to start a new run, we also update the running average of the times it takes to find a first solution as well as the incremental time interval that results from this running average times the factor $r$. Finally, we update the features and continue until the time has run out.

The last ingredient needed to apply this framework in practice is a method for learning the weights $w$. Based on a training set of instances, we compute weights that result in superior performance using the gender-based genetic algorithm tuner GGA [19], following the same general approach for tuning hyper-parameterized search methods as introduced in [17].

## 5 Experimental Analysis

We now present our numerical analysis. First, we briefly introduce the combinatorial optimization problems, the solvers, and the instances used in our experiments. Second, we describe our comprehensive data collection, which allows us to conduct our investigations completely offline, that is, without the need of running any additional experiments. Third, we present the results of our investigations which show the effectiveness of our online method.

### 5.1 Problems and Benchmarks

First, we briefly introduce the two considered NP-complete problems, as well as the corresponding solvers and benchmarks used in our investigations.

**Traveling Salesperson.** The Traveling Salesperson Problem (TSP) considers an edge-weighted graph $G = (V, E, w)$, the vertices $V = \{1, \dots, n\}$ are referred to as *cities*. It asks for a permutation $\pi$ of $V$ such that $\left(\sum_{i=1}^{n-1} w(\pi(i), \pi(i+1))\right) + w(\pi(n), \pi(1))$ (the cost of visiting the cities in the order of the permutation and then returning to the origin $\pi(1)$) is minimized.

Natural applications of the TSP are in areas like planning and logistics [21], but they are also encountered in a large number of other domains, such as genome sequencing, drilling problems, aiming telescopes, and data clustering [22]. TSP is one of the most important (and most studied) optimization problems.

We use the Chained-Lin-Kernighan (LINKERN) heuristic [23, 24], a state-of-the-art incomplete solver for the Traveling Salesperson problem. Its stochastic behavior comes from random components during the creation of the initial tour.

The TSPlib is a classic repository of TSP instances [25]. For our investigations, we pick all 112 instances from TSPlib, and as additional challenging instances ch71009, mona-lisa100k, and usa115475.

**Minimum Vertex Cover.** Finding a minimum vertex cover of a graph is a classical NP-hard problem. Given an unweighted, undirected graph $G = (V, E)$, a vertex cover is defined as a subset of the vertices $S \subseteq V$, such that every edge of $G$ has an endpoint in $S$, i.e. for all edges $\{u, v\} \in E$, $u \in S$ or $v \in S$. The decision problem $k$-vertex cover decides whether a vertex cover of size $k$ exists. We consider the optimization variant to find a vertex cover of minimum size.

Applications arise in numerous areas such as network security, scheduling and VLSI design [26]. The vertex cover problem is also closely related to the problem of finding a maximum clique. This has a range of applications in bioinformatics and biology, such as identifying related protein sequences [27].

Numerous algorithms have been proposed for solving the vertex cover problem. We choose FASTVC [28] over the popular NUMVC [29] as a solver for the minimum vertex cover problem as it works better for massive graphs. FASTVC is based on two low-complexity heuristics, one for initial construction of a vertex cover, and one to choose the vertex to be removed in each exchanging step, which involves random draws from a set of candidates.

For our experimental investigations, we select all 86 instances used in [28]. Among these, the number of vertices ranges from about 1000 to over 4 million, and the number of edges ranges from about 2000 to over 56 million.

### 5.2 Data Collection

We recorded 10,000 independent, regular runs of the original solvers on each of the 115 TSP instances and on each of the 86 MVC instances. For TSP, the time limit per instance was 1 hour. For MVC, we allowed 100 seconds. The runs were conducted on a compute cluster with Intel Xeon E5620 CPUs (2.4GHz).

For each run, we make a record whenever a solver finds a better solution, together with the solution quality. Altogether, the records of our 20,000 runs take up over 8 GB when GZ-compressed with default settings. We plan to make these files publicly available (upon finding a suitable webserver) as a resource for studying the behaviour of these algorithms.

### 5.3 Training of Hyper

For each of the two benchmarks, we used two thirds of the respectively available instances for training. That is, we handed the parameterized framework to a recently improved version of GGA that uses surrogate models to predict where improved parameterizations may be found [30] We ran GGA for 70 generations with a population size of 100 individuals. The random replacement rate was set to 5%, the mutation rate was set to 5% as well.

### 5.4 Results

Following the training of Hyper on two thirds of the instances (per problem domain), we are left with 38 of the 115 TSP instances and 28 of the 86 MVC instances. We use these to compare the performance of the following investigated approaches:

1. Single: the solver is run once with a random seed, allowing it to run for the total time given;
2. Restarts: the solver is restarted from scratch whenever a preset time limit is reached, and this loop is repeated until time is up;
3. Luby: restarts based on the fixed Luby sequence [3], where one Luby time unit is based on five times the time the first run needs to produce the first solution;
4. Bet-and-Run: the previously described bet-and-run strategy by Friedrich et al. [13];
5. Hyper: our trained hyper-parameterized bet-and-run restart strategy, as described above.

We will analyze the outcomes using several criteria. First, we compare the performance gaps achieved with respect to the optimal solution possible within the time budget.[5] Second, we consider the number of times an approach is able to find the best possible solution. Third, we compare the amount of time needed in order to compute the final results.

To start off, Tables 1 and 2 show the results of the individual solvers across the sets of 38 and 28 instances. Note that we are using the problem domain names TSP and MVC instead of the respective solvers to facilitate reading.

We observe that the number of times the best possible solution is found increases with increasing time budget. Note that this is not natural as the best possible solution is the best possible solution for the respective time limit! The fact that the relative gap decreases anyhow is therefore a reflection of the fact that the best restart can actually find the best solution rather quickly. With increasing time limits, the restarted approaches thus have more buffer to find this best quality solution as well.

Next, we find that Single and Restarts are clearly outperformed by the other three approaches across both problem domains and across all total time budgets. On TSP, Hyper achieves less than half the performance gap of Bet-and-Run when the total time budget is only 100 seconds. This advantage for Hyper becomes more and pronounced as the budget increases to 5,000 seconds.

[5] This best possible solution is the best solution provided within the given time limit by any of the 10,000 runs we conducted.

| time budget | solutions | no solutions | best found | average performance | average time |
|---|---|---|---|---|---|
| SINGLE | | | | | |
| 100 | 380 | 0 | 234 | 0.1415 | 12 |
| 200 | 378 | 2 | 239 | 0.1368 | 21 |
| 500 | 380 | 0 | 266 | 0.0885 | 95 |
| 1000 | 380 | 0 | 266 | 0.0877 | 105 |
| 2000 | 380 | 0 | 266 | 0.0762 | 165 |
| 5000 | 380 | 0 | 266 | 0.0596 | 290 |
| RESTARTS | | | | | |
| time budget | solutions | no solutions | best found | average performance | average time |
| 100 | 380 | 0 | 252 | 0.0689 | 21 |
| 200 | 380 | 0 | 255 | 0.0618 | 35 |
| 500 | 380 | 0 | 259 | 0.0519 | 61 |
| 1000 | 380 | 0 | 261 | 0.0474 | 98 |
| 2000 | 380 | 0 | 261 | 0.0457 | 154 |
| 5000 | 380 | 0 | 258 | 0.0435 | 268 |
| LUBY | | | | | |
| time budget | solutions | no solutions | best found | average performance | average time |
| 100 | 380 | 0 | 296 | 0.0274 | 19 |
| 200 | 380 | 0 | 299 | 0.0189 | 32 |
| 500 | 380 | 0 | 309 | 0.0135 | 75 |
| 1000 | 380 | 0 | 317 | 0.0108 | 127 |
| 2000 | 380 | 0 | 318 | 0.0090 | 229 |
| 5000 | 380 | 0 | 322 | 0.0070 | 476 |
| BET-AND-RUN | | | | | |
| time budget | solutions | no solutions | best found | average performance | average time |
| 100 | 380 | 0 | 244 | 0.0487 | 5 |
| 200 | 380 | 0 | 245 | 0.0473 | 6 |
| 500 | 380 | 0 | 246 | 0.0444 | 8 |
| 1000 | 380 | 0 | 248 | 0.0436 | 13 |
| 2000 | 380 | 0 | 251 | 0.0429 | 22 |
| 5000 | 380 | 0 | 256 | 0.0419 | 49 |
| HYPER | | | | | |
| time budget | solutions | no solutions | best found | average performance | average time |
| 100 | 380 | 0 | 295 | 0.0216 | 15 |
| 200 | 380 | 0 | 302 | 0.0142 | 26 |
| 500 | 380 | 0 | 307 | 0.0132 | 57 |
| 1000 | 380 | 0 | 307 | 0.0090 | 87 |
| 2000 | 380 | 0 | 319 | 0.0077 | 178 |
| 5000 | 380 | 0 | 321 | 0.0066 | 322 |

Table 1: TSP results. Shown are time in seconds, and performance gap from the best possible solution within the respective time limit. ”solutions” and ”no solutions” refer to the number of times the approach has produced any solution at all. ”best found” lists the number of times the best possible solution was found given 380 runs (38 instances * 10 independent runs). Highlighted in dark blue and light blue are the best and second best average approaches.

| time budget | solutions | no solutions | best found | average performance | average time |
|---|---|---|---|---|---|
| **SINGLE** | | | | | |
| 5 | 211 | 69 | 74 | 0.1097 | 3 |
| 10 | 223 | 57 | 76 | 0.4558 | 5 |
| 20 | 254 | 26 | 80 | 0.6181 | 9 |
| 50 | 264 | 16 | 98 | 0.2273 | 19 |
| **RESTARTS** | | | | | |
| 5 | 228 | 52 | 76 | 0.1111 | 3 |
| 10 | 252 | 28 | 82 | 0.4140 | 6 |
| 20 | 268 | 12 | 88 | 0.6128 | 12 |
| 50 | 278 | 2 | 101 | 0.1802 | 25 |
| **LUBY** | | | | | |
| 5 | 228 | 52 | 80 | 0.1064 | 3 |
| 10 | 252 | 28 | 91 | 0.3907 | 6 |
| 20 | 268 | 12 | 91 | 0.5767 | 11 |
| 50 | 278 | 2 | 114 | 0.1032 | 23 |
| **BET-AND-RUN** | | | | | |
| 5 | 228 | 52 | 65 | 0.0800 | 3 |
| 10 | 252 | 28 | 79 | 0.3328 | 5 |
| 20 | 268 | 12 | 90 | 0.4721 | 9 |
| 50 | 278 | 2 | 105 | 0.0390 | 18 |
| **HYPER** | | | | | |
| 5 | 228 | 52 | 75 | 0.0781 | 3 |
| 10 | 252 | 28 | 87 | 0.3309 | 5 |
| 20 | 268 | 12 | 104 | 0.4710 | 9 |
| 50 | 278 | 2 | 119 | 0.0385 | 19 |

Table 2: MVC results. Shown are time in seconds, and performance gap from the best possible solution within the respective time limit. ”solutions” and ”no solutions” refer to the number of times the approach has produced any solution at all. ”best found” lists the number of times the respective best possible solution has been found given 280 runs (28 instances * 10 independent runs). Highlighted in dark blue and light blue are the best and second best average approaches.

For this time limit, HYPER has a six-times lower average gap than BET-AND-RUN, which is marked improvement. At the same time, BET-AND-RUN can find the best solutions in only 67% of the runs, whereas HYPER's success rate is 84%.

MVC can be seen as a little bit more challenging in our setting, as the computation time budgets were rather short and FASTVC encountered significant initialization times on some of the large instances. As a consequence, the number of times where no solution has been produced by the various approaches is higher than for TSP, however, this number decreases with increasing time budget.

On MVC, HYPER and BET-AND-RUN are really close in terms of average performance gap, however, there is an advantage for HYPER in number of times the best possible solutions are found. In practice this is still a substantial improvement.

Interestingly, our results differ from [13], where Luby-based restarts performed not as well as RESTARTS, whereas in our study BET-AND-RUN is outperformed by LubyStat on TSP. This might be due to a different approach of setting $t_{\text{init}}$ and because we use a larger instance set for TSP. Independent of this HYPER outperforms both.

Figure 1 adds to the results by showing the results of performing single-sided Wilcoxon rank-sum tests on the outcomes of 10 independent runs. For the two different problem domains, we observe the following. For TSP, HYPER dominates the field and is beaten five times by LUBY (as assessed by the statistical tests) in terms of quality gap to the optimum. For MVC, HYPER typically outperforms SINGLE, RESTARTS, and LUBY. In contrast to this, HYPER and BET-AND-RUN perform essentially comparably on MVC, and the differences are rarely significant.

Lastly, we summarize the investigations by testing whether the performance differences between HYPER and and the other approaches across all instances and time budgets are statistically significant. Again, we apply a single-sided Wilcoxon test to test the null hypothesis that two given distributions are identical. We compare the approaches based on the performance gap achieved, and across all time budgets and instances. In particular, we take the median of the 10 independent runs per instance, and then collect for each restart approach the medians across all instances and time limits. As a consequence, each approach has 38*6=228 medians for TSP and 28*4=112 medians for MVC.

Table 3 shows the results of these two tests. In summary, we can deduce from the outcome that HYPER performs no worse than existing approaches, and typically better. A closer inspection of the raw results of HYPER and BET-AND-RUN on MVC reveals that their performance is near-identical, despite the fact that the averages of HYPER are consistently lower than those of BET-AND-RUN (as seen in Table 2). In stark contrast to this, the performance comparisons on TSP are mostly highly significant and in an favor of HYPER.

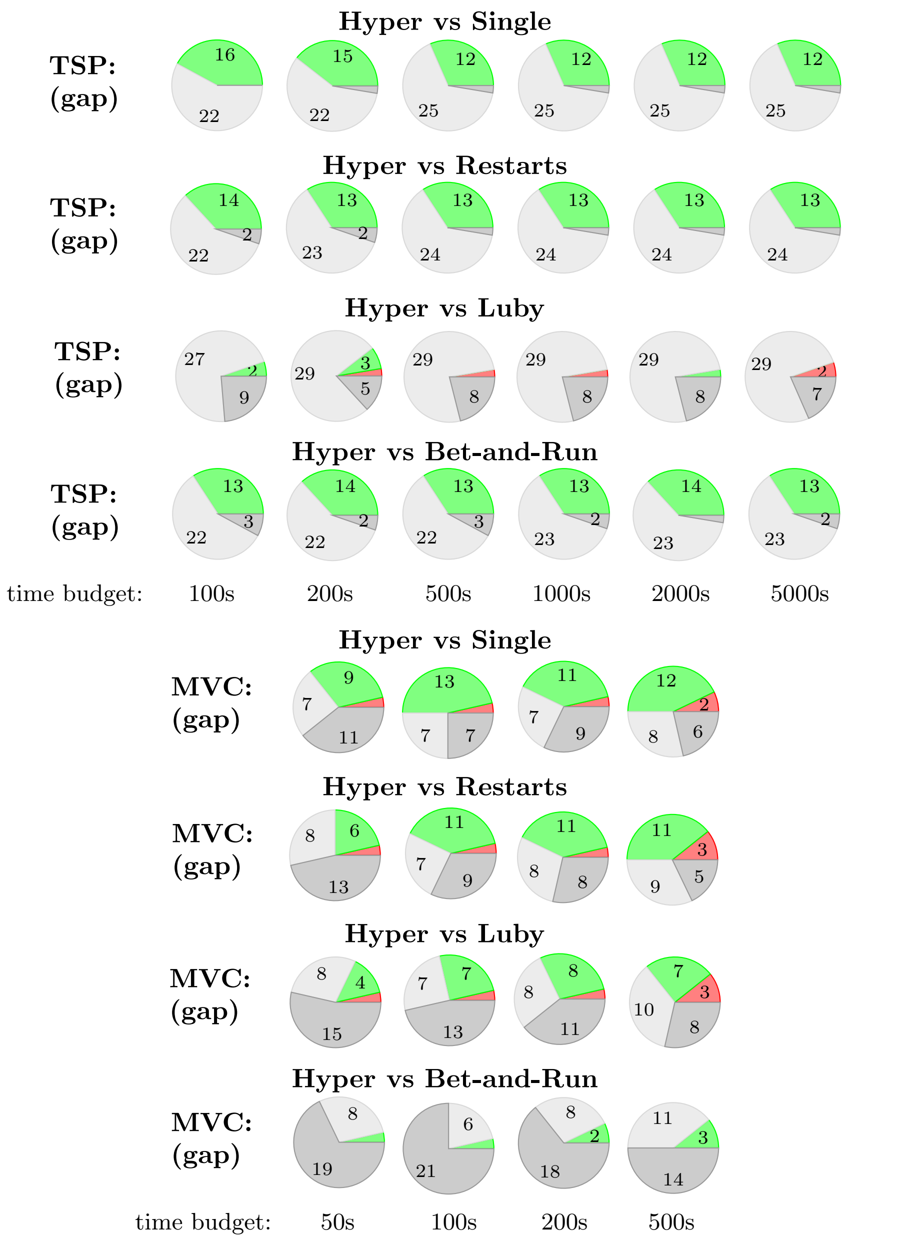

Fig. 1: Statistical comparison of Hyper with the other approaches using the Wilcoxon rank-sum test (significance level $p = 0.05$). The approaches are compared based on the quality gap to the best possible solution (smaller is better). The colors have the following meaning: Green indicates that Hyper is statistically better, Red indicates that Hyper is statistically worse, Light gray indicates that both performed identical, Dark gray indicates that the differences were statistically insignificant. We have chosen pie charts on purpose because they allow for a quick qualitative comparison of results.

| | Single | Restarts | Luby | Bet-and-Run |
|---|---|---|---|---|
| TSP | 0.000003 | 0.000015 | 0.478300 | 0.000002 |
| MVC | 0.060172 | 0.248689 | 0.354935 | 0.236808 |

Table 3: One-sided Wilcoxon rank-sum test to test whether the quality gap distribution of Hyper is shifted to the left of that of the other approaches. Shown are the p-values.

## 6 Conclusion

We introduced the idea of learning reactive restart strategies for combinatorial search algorithms. We compared this new approach (Hyper) with other approaches, among them a very recent Bet-and-Run approach that had been assessed comprehensively on TSP and MVC instances. Across both domains, Hyper resulted in markedly better average solution qualities, and it exhibited significantly increased rates of hitting the best possible solution.

As the investigated problem domains are structurally very different, we expect our approach to generalize to other problem domains as well, such as continuous and multi-objective optimization problems.

Future work will focus on the development of other runtime features as a basis for making restart decisions.

## Bibliography

[1] Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1) (2000) 67–100

[2] Hoos, H.H.: Stochastic local search - methods, models, applications. PhD thesis, TU Darmstadt (1998)

[3] Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters **47**(4) (1993) 173–180

[4] Wu, H., van Beek, P.: On universal restart strategies for backtracking search. In: Principles and Practice of Constraint Programming (CP), Springer (2007) 681–695

[5] Audemard, G., Simon, L.: Refining Restarts Strategies for SAT and UNSAT. In: Principles and Practice of Constraint Programming (CP). Springer (2012) 118–126

[6] Marti, R.: Multi-start methods. In Glover, F., Kochenberger, G.A., eds.: Handbook of Metaheuristics. (2003) 355–368

[7] Loureno, H.R., Martin, O.C., Stützle, T.: Iterated local search: Framework and applications. In: Handbook of Metaheuristics. Springer (2010) 363–397

[8] Schoenauer, M., Teytaud, F., Teytaud, O.: A rigorous runtime analysis for quasi-random restarts and decreasing stepsize. In: Artificial Evol., Springer (2012) 37–48

[9] Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Theory and Applications of Satisfiability Testing (SAT). (2008) 28–33

[10] Huang, J.: The effect of restarts on the efficiency of clause learning. In: Int. Joint Conference on Artifical Intelligence (IJCAI). (2007) 2318–2323

[11] Lalla-Ruiz, E., Voß, S.: Improving solver performance through redundancy. Systems Science and Systems Engineering **25**(3) (2016) 303–325

[12] Fischetti, M., Monaci, M.: Exploiting erraticism in search. Operations Research **62**(1) (2014) 114–122
[13] Friedrich, T., Kötzing, T., Wagner, M.: A generic bet-and-run strategy for speeding up stochastic local search. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. (2017) 801–807
[14] Andrei Lissovoi, Dirk Sudholt, M.W.C.Z.: Theoretical results on bet-and-run as an initialisation strategy. In: Genetic and Evolutionary Computation Conference (GECCO). (2017) Accepted for publication.
[15] Stützle, T., López-Ibáñez, M.: Automatic (offline) configuration of algorithms. In: Genetic and Evolutionary Computation Conference (GECCO). (2016) 795–818
[16] Bezerra, L.C.T., López-Ibáñez, M., Stützle, T.: Automatic component-wise design of multiobjective evolutionary algorithms. IEEE Transactions on Evolutionary Computation **20**(3) (2016) 403–417
[17] Ansótegui, C., Pon, J., Tierney, K., Sellmann., M.: Reactive dialectic search portfolios for MaxSAT. In: AAAI Conference on Artificial Intelligence. (2017) Accepted for publication.
[18] Kadioglu, S., Sellmann, M.: Dialectic search. In: Principles and Practice of Constraint Programming (CP) 2009, Springer (2009) 486–500
[19] Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: Principles and Practice of Constraint Programming (CP). (2009) 142–157
[20] Hartigan, J.A.: Bounding the maximum of dependent random variables. Electronic Journal of Statistics **8**(2) (2014) 3126–3140
[21] Polacek, M., Doerner, K.F., Hartl, R.F., Kiechle, G., Reimann, M.: Scheduling periodic customer visits for a traveling salesperson. European Journal of Operational Research **179** (2007) 823–837
[22] Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press (2011)
[23] Applegate, D.L., Cook, W.J., Rohe, A.: Chained Lin-Kernighan for large traveling salesman problems. INFORMS Journal on Computing **15**(1) (2003) 82–92
[24] Cook, W.: The Traveling Salesperson Problem: Downloads (Website). `http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm` (2003) [Online; accessed 21-Dec-2016].
[25] Reinelt, G.: TSPLIB – A Traveling Salesman Problem Library. ORSA Journal on Computing **3**(4) (1991) 376–384 Instances: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`. [Online; accessed 21-Dec-2016].
[26] Gomes, F.C., Meneses, C.N., Pardalos, P.M., Viana, G.V.R.: Experimental analysis of approximation algorithms for the vertex cover and set covering problems. Computers and Operations Research **33**(12) (2006) 3520–3534
[27] Abu-Khzam, F.N., Langston, M.A., Shanbhag, P., Symons, C.T.: Scalable parallel algorithms for FPT problems. Algorithmica **45**(3) (2006) 269–284
[28] Cai, S.: Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In: International Joint Conference on Artificial Intelligence (IJCAI). (2015) 747–753 Code: `http://lcs.ios.ac.cn/~caisw/MVC.html`. [Online; accessed 21-Dec-2016]",.
[29] Cai, S., Su, K., Luo, C., Sattar, A.: Numvc: An efficient local search algorithm for minimum vertex cover. Journal of Artificial Intelligence Research **46**(1) (2013) 687–716
[30] Ansótegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K.: Model-based genetic algorithms for algorithm configuration. In: International Joint Conference on Artificial Intelligence (IJCAI). (2015) 733–739