An Improved Generic BET-AND-RUN Strategy with Performance Prediction for Stochastic Local Search

Thomas Weise[1], **Zijun Wu**[1], **Markus Wagner**[2]
[1]Institute of Applied Optimization, Faculty of Computer Science and Technology, Hefei University
[2]Optimisation and Logistics, The University of Adelaide, Adelaide, Australia

Abstract

A commonly used strategy for improving optimization algorithms is to restart the algorithm when it is believed to be trapped in an inferior part of the search space. Building on the recent success of BET-AND-RUN approaches for restarted local search solvers, we introduce a more generic version that makes use of performance prediction. It is our goal to obtain the best possible results within a given time budget t using a given black-box optimization algorithm. If no prior knowledge about problem features and algorithm behavior is available, the question about how to use the time budget most efficiently arises. We first start $k \geq 1$ independent runs of the algorithm during an initialization budget $t_1 < t$, pause these runs, then apply a decision maker D to choose $1 \leq m < k$ runs from them (consuming $t_2 \geq 0$ time units in doing so), and then continue these runs for the remaining $t_3 = t - t_1 - t_2$ time units. In previous BET-AND-RUN strategies, the decision maker $D =$ currentBest would simply select the run with the best-so-far results at negligible time. We propose using more advanced methods to discriminate between "good" and "bad" sample runs with the goal of increasing the correlation of the chosen run with the a-posteriori best one. In over 78 million experiments, we test different approaches to predict which run may yield the best results if granted the remaining budget. We show (1) that the currentBest method is indeed a very reliable and robust baseline approach, and (2) that our approach can yield better results than the previous methods.

Introduction

Optimization algorithms are widely used in a variety of domains, such as production scheduling and planning or vehicle routing. In many such practical applications, the total time budget t available for optimization is limited to at most a few minutes. The goal is to find a solution which is as-good-as-possible within this budget. One method to do so is to develop better optimization algorithms. Another method is to make the best use of an existing solver.

There are two straightforward methods when approaching an optimization problem with one algorithm and a total time budget t: One can either assign the whole budget t to a single run of the algorithm or execute k independent restarts of the algorithm (Martí 2003; Lourenço, Martin, and Stützle 2010) and therefore divide the budget t into k equally-sized chunks. It can be expected that the former strategy is the best for small budgets while the latter one is the better choice for large budgets. Budgets t of a few minutes, however, fall in neither category for many problem types, which, of course, depends on the instance and the solver.

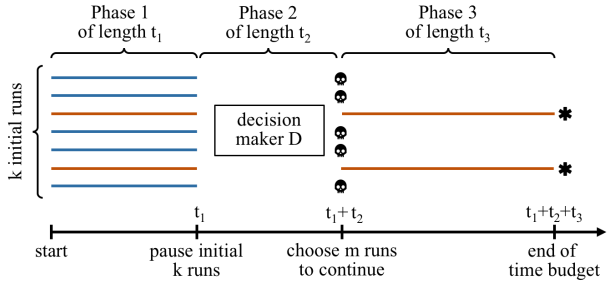


Figure 1: Our generic BET-AND-RUN strategy receives a total time budget t. It starts k independent runs and pauses them after time t_1. A decision maker D then takes time t_2 to decide which of them to continue. All but the m chosen runs are terminated (marked with 💀). The m chosen runs (marked with ✱) continue for a total of t_3.

Here, BET-AND-RUN strategies (Fischetti and Monaci 2014) pose a compromise by using an initialization time budget $0 \leq t_1 \leq t$ which is divided evenly amongst k independent runs. The run with the best-so-far solution is then continued for the remaining $t - t_1$ time units. (Friedrich, Kötzing, and Wagner 2017) showed that this simple approach can routinely outperform the two budgeting approaches above. Yet, it makes only use of a single unit of information per run for the "decision" which of the k runs to resume, namely the solution quality they reached at the end of their respective initialization budgets.

In order to investigate the question *Can we do better than that?*, we generalize the BET-AND-RUN concept as illustrated in Figure 1: The budget t is divided into three pieces, i.e., $t = t_1 + t_2 + t_3$ and used as follows.

Phase 1 The initialization budget t_1 is divided among a set of k initial, independent runs according to a budgeting strategy. All of these runs are paused after t_1 has been consumed.

Phase 2 Then, a decision maker D is applied which may access the history of each run in form of $(\mathit{time}, \mathit{quality})$

tuples. D will choose $1 \leq m \leq k$ of the k runs for continuation and consume an *a priori* unknown time t_2 while doing so.

Phase 3 The remainder t_3 of the total budget t is then divided evenly among the m chosen runs, which thus each receive $(t - t_1 - t_2)/m$ additional time units.

In this article, we show (1) that the currentBest method is indeed a very reliable and robust baseline approach, and (2) that our approach with performance prediction can yield better results than the previous methods.

Background

In practice, stochastic search algorithms and randomized search heuristics are frequently restarted: If a run does not conclude within a pre-determined solution quality limit, we restart the algorithm (Martí 2003; Lourenço, Martin, and Stützle 2010). One of the advantages of this simple approach is that it helps to avoid heavy-tailed runtime distributions (Gomes et al. 2000). However, 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 restart 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, Sinclair, and Zuckerman 1993) 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. Even if the distribution is unknown, there is a universal sequence of running times given by (1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,...), which is the optimal restarting strategy up to constant factors. While these results can be used for every problem setting, they only apply to Las Vegas algorithms.

Fewer results are known for the optimization case. A range of practical approaches for such restart strategies is known (Martí 2003; Lourenço, Martin, and Stützle 2010). A relatively recent theoretical result is presented by (Schoenauer, Teytaud, and Teytaud 2012). Several studies show the substantial impact of the restart policy on the efficiency of solvers for satisfiability problems (Biere and Fröhlich 2015; Huang 2007). In this context, restarts have also been used to learn "no-goods" during backtracking (Ciré, Kadioglu, and Sellmann 2014).

Quite often, classical optimization algorithms are deterministic and thus cannot be improved by restarts, as run time and outcome will not change. However, their characteristics can be subject to change. For example, (Lalla-Ruiz and Voß 2016) exploited this by using different mathematical programming formulations so as to provide different starting points for the solver. While many other modern optimization algorithms also work mostly deterministically, they often have some randomized component, for example by choosing a random starting point. These initial solutions often strongly influence the quality of the outcome and the speed of reaching it. In our opinion, it follows quite naturally that algorithms should be run several times.

(Fischetti and Monaci 2014) extended the classical restart strategies to the so-called BET-AND-RUN strategy:

Phase 1 perform k runs of the algorithm for some (short) time limit $t_1 \leq t$, assigning t_1/k time units to each run.

Phase 2 use remaining time $t_3 = t - t_1$ to continue *only the best run* from the first phase until timeout.

They experimentally studied this 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. For them, $k = 5$ performed best.

(de Perthuis de Laillevault, Doerr, and Doerr 2015) have shown that a BET-AND-RUN strategy can also benefit asymptotically from larger k. For the pseudo-boolean test function ONEMAX it was proven that choosing $k > 1$ decreases the $O(n \log n)$ expected run time of the (1+1) evolutionary algorithm by an additive term of $\Omega(\sqrt{n})$.

(Lissovoi et al. 2017) 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. They proved that non-trivial k and t_1 are necessary to find the global optimum efficiently, and that the choice of t_1 is linked to features of the problem. They also provided a fixed budget analysis to guide selection of the BET-AND-RUN parameters to maximize the solution quality.

(Friedrich, Kötzing, and Wagner 2017) investigated a range of BET-AND-RUN strategies on the traveling salesperson problem and the minimum vertex cover problem. Their best strategy performed 40 short runs in the initial phase with a time limit that is 1% of the total time budget each, and then it used 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, Sinclair, and Zuckerman 1993) as well, using various choices of t_1, however, it turned out inferior.

Building on the success of BET-AND-RUN approaches for restarted local search solvers, (Kadioglu, Sellmann, and Wagner 2017) introduced the idea of adaptive restart strategies. Inside their approach, a learned black-box decision procedure dynamically decides whether to continue the current run, to continue a previous run, or to start a new run. While their approach performed favorably, the internal mechanisms were black-box and it remained unclear which algorithmic components and which decisions contributed to the success.

Note that the stream of BET-AND-RUN-related research is related to the very mature field of multi-armed bandits. To the best of our knowledge, however, there are no existing works there that make use of the core ideas of BET-AND-RUN to solve a single instance, i.e., to have an overall limited budget as well as the idea to exclusively stick to one arm after some first exploratory phase. For example, (Gagliolo and Schmidhuber 2011) propose a method for allocating computation time to algorithm portfolios for solving instance sets (thus working on a much higher/coarser granularity), however, their approach does not carry over to our fine-grained scenario of using partial runs for optimizing a single instance.

Benchmarks

Here we shortly introduce the two benchmark problems and the optimization algorithms used to solve them.

Minimum Vertex Cover Problem

Solving the minimum vertex cover problem (MVC) means finding the smallest set of vertexes of a graph which contains at least one vertex from every edge. The MVC is one of the classical $\mathcal{NP}$-hard problems with many applications (Gomes et al. 2006). It also is closely related to the problem of finding a maximum clique (Abu-Khzam et al. 2006). The state-of-the-art algorithms for solving the MVC comprise FASTVC (Cai 2015), NuMVC (Cai et al. 2013), TwMVC (Cai, Lin, and Su 2015), and FastWVC (Li, Cai, and Hou 2017).

(Kadioglu, Sellmann, and Wagner 2017) applied FASTVC (Cai 2015) in their experiments, one of the best algorithms for large MVC instances. FASTVC is based on two low-complexity heuristics. The first one constructs an initial vertex cover and the second one chooses the vertex to be removed in each exchanging step, which involves random draws from a set of candidates. We use the data that (Kadioglu, Sellmann, and Wagner 2017) gathered in 10,000 independent runs on the 86 instances used by (Cai 2015). These real-world instances are of rather large scale and most of them are sparse, which is challenging for solvers. The number of vertices in the instances ranges from about 1000 to over 4 million and the number of edges from about 2000 to over 56 million.

Traveling Salesperson Problem

The traveling salesperson problem (TSP) (Applegate et al. 2007; Lawler et al. 1985) is one of the most well-known combinatorial optimization tasks. A TSP instance is defined as a fully-connected graph. Each edge in the graph has a weight, representing the distance between the two nodes it connects. A candidate solution is a cycle that visits each node in the graph exactly once and returns back to its starting node. The objective function, subject to minimization, is the sum of the weights of all edges in the tour, i.e., the total tour length. This optimization version of the TSP is $\mathcal{NP}$-hard (Gary and Johnson 1979; Gutin and Punnen 2002). The state-of-the-art algorithms for the TSP include EAX (Nagata and Kobayashi 2013), the Chained-Lin-Kernighan heuristic (Applegate, Cook, and Rohe 2003; Cook 2005), Partition Crossover (Whitley 2016), as well as hybrid metaheuristics (Liu et al. 2015).

We use the data from (Kadioglu, Sellmann, and Wagner 2017), who used the Chained-Lin-Kernighan heuristic (Applegate, Cook, and Rohe 2003; Cook 2005) as TSP solver. They applied it 10,000 times to each of the 111 symmetric instances from TSPLib (Reinelt 1991) and additionally to the large instances `ch71009`, `mona-lisa100k`, and `usa115475`. We omit instance `linhp318`, as no data was available on it.

In the next sections, for the sake of readability, we will refer to the combination of solver and problem by just using the problem domain, i.e., TSP and MVC.

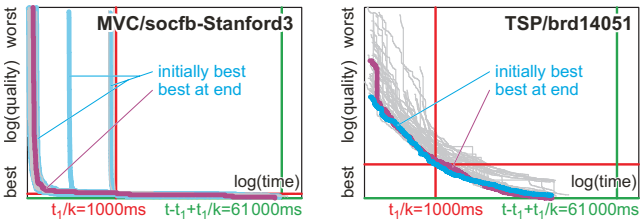


Figure 2: $k = 40$ selected runs from the datasets `brd14051` (TSP) and `socfb-Stanford3` (MVC) for a total budget of $t = 100'000ms$ and an initialization budget of $t_1 = 40'000ms$, illustrating that the runs which are best after the initialization budget ($t_1 = 1000ms$) are not necessarily the best ones after the full budget (colored violet, $t_1 + t_3 = 61000ms$).

BET-AND-RUN with Better Decision Makers

Our generalized BET-AND-RUN strategy can simulate a range of existing approaches:

- The simple multi-run strategy of restarting from scratch k times is a special case by choosing $t_1 = t/k$ and $t_2 = t_3 = 0$.
- The single-run strategy corresponds to a multi-run method with $k = 1$.
- The strategies from (Fischetti and Monaci 2014; Friedrich, Kötzing, and Wagner 2017) and (Lissovoi et al. 2017) are special cases by choosing $m = 1$ and having $t_2 \approx 0$.

Our experiments detailed in the next section therefore also cover these approaches.

Let us consider the illustrations in Figure 2. In all related work, $D =$ currentBest is applied, which picks one of the runs with the current best result after initialization (cyan) and resumes it where it was paused. Intuitively, this is a robust strategy for which $t_2 \approx 0$ holds, but it does not necessarily pick the run which yields the best result (violet) after all budget has been exhausted. In such scenarios, paying some time cost t_2 for a more sophisticated choice can yield a better overall result.

In our experiments, we need to use clock time as time measure and cannot apply any other measure common in optimization (Weise et al. 2014) such as function evaluations (FEs). This is because the decision makers do not evaluate the objective function or generate candidate solutions by themselves, but only process the data already collected, i.e., the aforementioned $(\textit{time}, \textit{quality})$ tuples. Using clock time to measure the computational effort of both the optimization algorithm and the decision maker has the further advantage that we can also consider otherwise "hidden" costs, such as for the initialization of data structures and bookkeeping.

In order to get an initial estimate on how likely such scenarios are, we randomly draw 1'000'000 samples of k runs from our each of our 86 MVC and 113 TSP datasets. In Table 1, we count how often the runs chosen by currentBest, which were best after time t_1/k, are outperformed by another run after time $t - t_1 + t_1/k$.

As can be seen, at least for $t = 100s$ and $t_1 = 40s$, currentBest cannot be beaten in the majority of samples. Still, in

Table 1: Baseline performance of currentBest. Shown is the number (and percentage) of instances from the MVC and TSP experiments where another decision maker could potentially outperform currentBest and the estimated overall probability averaged over all datasets. $t = 100s$, $t_1 = 40s$ and $t_2 = t_3 = 0$

experiment	k	instances	mean probability
MVC	4	67 (78%)	0.11
MVC	10	52 (60%)	0.17
MVC	40	32 (37%)	0.23
TSP	4	30 (27%)	0.03
TSP	10	34 (30%)	0.06
TSP	40	44 (39%)	0.12

78% of the MVC benchmark instances, there were at least some samples of $k = 4$ runs where currentBest made the wrong choice. For $k = 40$, the chance to theoretically being able to outperform currentBest on a random instance of the MVC problem is 23%. For the TSP, these chances tend to be lower, but there is still a potential to improve the overall performance. However, these are mean probabilities, and the actual values can deviate significantly. For example, for the two instances shown in Figure 2, the observed probabilities of outperforming currentBest when varying $k \in \{4, 10, 40\}$ range from 0.25 to 0.93 (`brd14051`) and 0.01 to 0.10 (`socfb-Stanford3`).

A decision maker *better* than currentBest would need to, e.g., outperform it in at least some of these scenarios while not performing worse in others. Although we have confirmed that there exist sufficiently many scenarios where this is potentially possible, there is another requirement which may decrease these chances: The performance data collected during the initialization budget $t_{1,i}$ of a run i must permit making a sufficiently accurate prediction regarding the future progress of that run. If this is true, then sophisticated decision makers have a chance to yield better results. (Qi, Weise, and Li 2017; 2018), for instance, showed that perceptrons have good prediction accuracy in their experiments on the Maximum Satisfiability Problem and the TSP with simple solvers. This prediction capability should make them suitable for determining which solution quality a run would yield if continued for a certain amount of time. However, other techniques, such as linear predictors, might yield improvements as well.

Experimental Study

Experimental Setup

To investigate the performance of our approach, and in particular to investigate the benefits of our more general BET-AND-RUN setup, we first perform a wide scan of many different setups and then investigate fewer setups in more detail. All datasets have been made publicly available (Weise and Wagner 2018).

Initial Large-Scale Experiment. In the first set of experiments, we limit ourselves to 20 random samples for each benchmark dataset and setup. We cover the total budgets $t \in \{1s, 4s, 10s, 40s, 100s, 400s\}$. For $k \in \{4, 10, 40\}$ and $m \in \{1, 2\}$, we test 25 different values of t_1. These are automatically chosen according to a heuristic based on the data from (Kadioglu, Sellmann, and Wagner 2017) for each instance before the experiment in order to maximize the number of different, meaningful outputs, e.g., the smallest values of t_1 are chosen that there is at least one data point. We briefly investigated more choices for m, but found that the preliminary results did not look very promising while the overall experimental time required would have more than doubled.

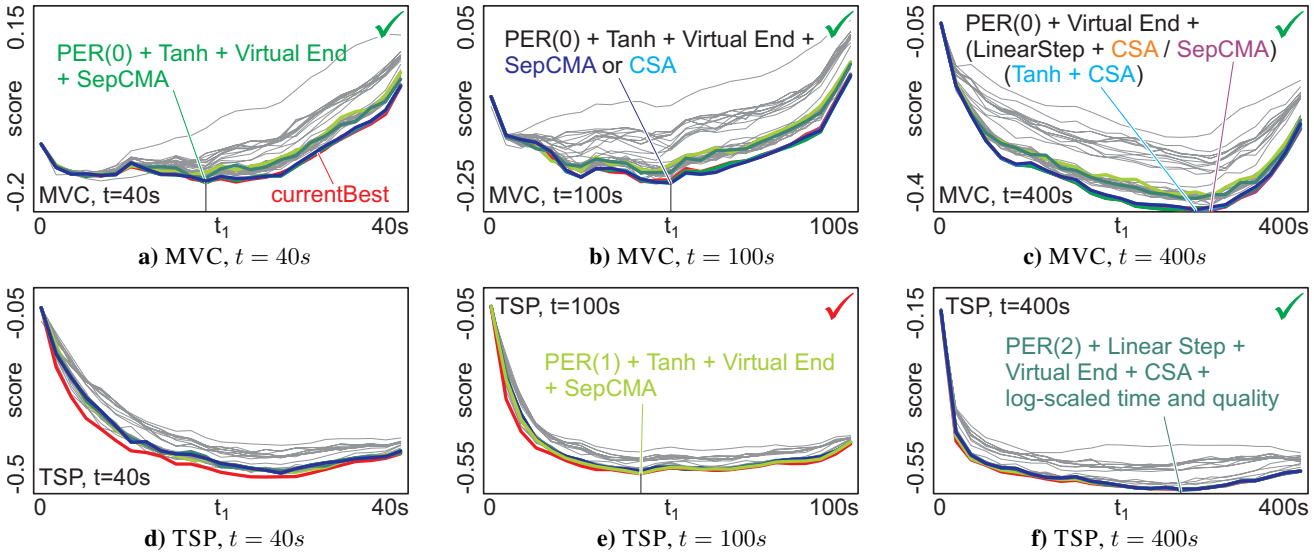
For the distribution of t_1 among the k initial runs, two strategies are tested. EVEN assigns $t_{1,i} = t_1/k$ for each $i \in 1..k$. LUBY instead follows the Luby sequence (Luby, Sinclair, and Zuckerman 1993) and sets $t_{1,i} = l(i)$, which equals to 2^{z-1} if $i = 2^z - 1$ and to $i - 2^{z-1} + 1$ otherwise, with $2^{z-1} \leq i < 2^z - 1$. The values of t_1 are chosen to be multiples of $\sum_{i=1}^{k} l(i)$ for the LUBY experiments and multiples of k for those using EVEN.

Our decision makers have access to the measured data points collected until t_1 is exhausted in the form of tuples of $(\textit{time}, \textit{quality})$. The set of basic decision makers includes currentBest, which picks the runs with the best *quality* value in their last measured data point, random, which randomly picks runs, and currentWorst, which picks the worst runs. The latter two performed worse and were included as sanity tests only. mostImprovements simply choses the run(s) i with the highest number of improvements (measure points) divided by the logarithm of their consumed time $t_{1,i}$ in the hope that they may be likely to attain further improvements. logTimeSum chooses the runs for which the sum of the logarithms of all time stamps at which improvements were made are the highest.

We also propose model-based decision makers that try to construct, for each run, a functional relationship between the time stamps and the achieved quality. These relationships are used to predict the quality that a run would reach if it was selected and pick the runs with the best predicted results.

As *model types* we test linear, quadratic, and cubic polynomials as well as perceptrons. The latter is suggested by (Qi, Weise, and Li 2017; 2018) for modeling optimization algorithm behavior. We apply perceptrons PER(n) with $n \in \{1, 2, 3\}$ nodes on a single hidden layer and such just with input/output layer ($n = 0$). We use either $\tanh$ or the linear step as activation function. The parameters of the polynomials can either be computed directly based on two, three, or four data points or fitted using the Levenberg-Marquardt algorithm (Levenberg 1944; Marquardt 1963) algorithm based on last ten measured points. The parameters of the perceptrons are obtained by either applying SepCMA-ES (Ros and Hansen 2008) or CSA (Arnold and Beyer 2008) for at most 400 function evaluations, on the last 10 points collected in the run. We chose 10 points only in order to limit the runtime t_2 consumed for training the perceptrons, which grows linearly with the number of points in our setup.

For the modeling, time and quality may be either used directly or logarithmically scaled. Furthermore, if the time value of the last measured tuple $(\textit{time}, \textit{quality})$ is less than $t_{1,i}$, we may add a "virtual end point" $(\textit{time}, t_{1,i})$ to the dataset of run i. This makes sense because an optimization



a) MVC, $t = 40s$ **b)** MVC, $t = 100s$ **c)** MVC, $t = 400s$

d) TSP, $t = 40s$ **e)** TSP, $t = 100s$ **f)** TSP, $t = 400s$

Figure 3: Performance of the decision makers compared with a single run executed over the whole budget t, for strategy EVEN, with $k = 40$, $m = 1$, and different values of t (diagrams) and t_1 (x-axes). We display the average score over all benchmark instances of the MVC/TSP datasets at the y-axes. For each time a setup yields a better result than the single run would have yielded, it receives a score of -1, for each time it returns a worse solution, it yields 1 (0 for the same solution). Only the relevant of the 44 decision makers are highlighted. If the best-performing setup was not currentBest, the diagram is marked with ✓, if another setup scored equally good with currentBest, we mark the diagram with ✓.

process may first quickly trace down a local optimum and then not improve anymore at all. In that case, no further measure point would appear in its initial budget and simply extrapolating its initial progress while ignoring this fact may yield wrong predictions. Finally, we test a linear model extrapolating from the very first measured point and the "virtual end point" of a run into the future.

This results in 44 decision maker setups, yielding a total of $(113 + 86) * 20 * 25 * 6 * 3 * 2 * 44 = 157'608'000$ experiments simulated on the data from (Kadioglu, Sellmann, and Wagner 2017).

Targeted Smaller Experiment. In a second experiment we investigate fewer, selected values of t_1, which also allows us to test additional configurations.

We investigate one additional decision maker, diminishing, which is based on the idea of *diminishing returns* (Samuelson and Nordhaus 2001). We set $\Delta_q = \min\{0.95, \Delta_{q,1}/\Delta_{q,2}\}$, where $\Delta_{q,1}$ be the last improvement in terms of quality a run has made and $\Delta_{q,2}$ the previous one. We further set $\Delta_t = \max\{1.05, \Delta_{t,1}/\Delta_{t,2}\}$ where $\Delta_{t,1}$ and $\Delta_{t,2}$ are the corresponding required runtime. The decision maker assumes that it will take longer by factor Δ_t to achieve each further improvement for the run, which, in turn, will be smaller by factor Δ_q. Improvements and times are always discretized.

In this experiment, we set $m = 1$. We choose $t \in \{2s, 10s, 20s, 50s, 100s, 200s, 500s, 1000s, 2000s, 5000s\}$ in correspondence to (Kadioglu, Sellmann, and Wagner 2017), who used the range 50s to 500s for MVC and 100s to 5000s for TSP. We take 1000 samples for each setup.

Results

Initial Large-Scale Experiment. An experiment of the scale and with as many parameters as described before cannot be discussed in full here. Our findings confirm that currentBest is a very robust basic strategy that performs the best in many situations. We know from Table 1 that good decision makers should perform very similar to it and only sometimes can yield better results. Averaged over all benchmark instances, it should be possible to gain an advantage of a few percent. From Table 1 we can predict that this advantage should be bigger on the MVC than on the TSP.

Indeed, in Figure 3, we can observe exactly this.[1] We find that perceptron-based decision makers work generally well and are (slightly) more likely to most-often outperform a single run with the full budget than currentBest on MVC for all $t_1 \in \{40s, 100s, 400s\}$ while this only holds for $t_1 = 400s$ on the TSP.

Larger total budgets t seem to be beneficial when the goal is to outperform single runs or currentBest. Note that this is a parameter which cannot be controlled by the user as it results from application requirements.

The time t_2 needed by the decision makers is generally the highest for perceptron-based methods (influenced by the presence and size of the hidden layer) and in the 100ms range. If we do not consider t_2 in our simulated experiments, i.e., artificially set $t_2 = 0$, the outcome of the experiments stays almost the same. t_2 is deducted from t_3 to be used for continuing the selected runs. It would be conceivable that

[1] Over all values of k, m, and strategies EVEN and LUBY, we can observe both scenarios where currentBest is outperformed and such where it is not. We selected figures without bias.

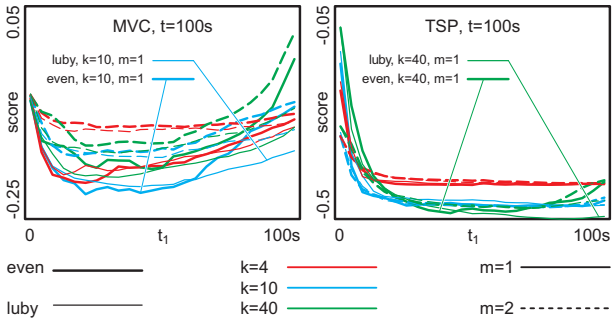


Figure 4: The average scores of the different budgeting strategies for $t = 100s$ over all decision makers and MVC and TSP instances. (see Figure 3 for definition of "score")

using much time to make a decision could decrease t_3 too much so that the gain from better prediction is destroyed by the loss of budget for actually attaining the gain. However, the experiment indicates that using more complex decision makers requiring more time t_2 may be viable, e.g., using more than the last 10 points to train our perceptrons would have been possible.

We now analyze the impact of the budgeting strategy, i.e., the choices of k, m, and whether to apply the EVEN or LUBY time distributions. Good choices of these parameters should obviously depend on the available total budget t and the runtime behavior of the solvers. In Figure 4 we plot the performance of the different configurations for $t = 100s$, averaged over *all decision makers* and for different values of t_1.

Using EVEN with $k = 10$, $m = 1$ is the best choice for the MVC $t \in \{40s, 100s\}$ and the TSP for $t \in \{10s, 40s\}$. For smaller budgets of the MVC and $t = 100s$ on the TSP, it makes sense to just perform $k = 40$ independent restarts distributing the time according to the LUBY strategy. This may result from the fact that the set of decision makers over which we average also contains worse performing methods such as currentWorst and random. Continuing two runs ($m = 2$) only is a good choice for the large budgets $t = 400s$ on both the MVC and TSP. This cost of continuing a second run is only then outweighed by the benefits of exploiting the variance of runtime performance.

Alternatives to currentBest. As shown in Table 1 and Figure 3, it is theoretically and practically possible to outperform the predictor currentBest. Next, we show to which extent and under which conditions we are able to do so given the predictors described above.

We compare our results with the best approach from (Friedrich, Kötzing, and Wagner 2017) (named F17 here and used as a benchmark by us), which uses currentBest to pick $m = 1$ run from $k = 40$ initial runs, each of which received 1% of the total time budget t, i.e., $t_1 = 0.4t$. The purpose of this comparison is to see whether improvements are possible, and also whether they are statistically significant.

In Figure 5, we show a qualitatively representative subset of our results. We have chosen two extreme total time budgets (a very small one of 2s and a very large one of 2'000s) and selected a diverse set of predictors. Note that we have chosen pie charts on purpose as they allow for a quick qualitative comparison of results.

It turns out that the benchmark approach dominates or is dominated depending on the problem domain, the instances, and the total time budget. For example, it is no surprise that the benchmark approach can beat the single run (lots of red) when the total time budget is large, as performance variance can be exploited. Also, we can see that the last phase of BETAND-RUN, i.e. when a run is continued, is generally helpful when the total runtime is short, as both EVEN and LUBY are beaten significantly and often in both the MVC and TSP case (lots of red).

For MVC and small budgets, many predictors can beat the benchmark approach. This advantage vanishes as the total time budget increases, which is due to the algorithm's convergence within the used time t_1/k for the individual initial runs. Consequently, performances are typically not distinguishable anymore from F17 (lots of gray), while the differences remain statistically significant for the TSP.

When it comes to the different problem domains, it also turns out that for MVC many predictors perform better than the benchmark approach. For example, the diminishing approach is significantly better on 43 instances while worse only on 24 instances; similar ratios hold for the other predictors. For the TSP and the long total time budget, however, there are a few deviations from the "usual" pie chart in this category. Noteworthy deviations are the perceptrons PER(0) without hidden layer and diminishing. In both cases, the benchmark is better on only three instances (as visible by the little red section), while being beaten on 19 instances (shown in green).

Lastly, we briefly compare the performance of different predictors when only 4 instead of 40 initial runs are performed. The results in Figure 6 show that predictors more elaborate than currentBest are again significantly more successful in picking the best run for both small and large total time budgets (lots of green and gray).

Summary and Conclusions

Over a wide variety of scenarios, we found that predicting the future performance of the initial runs in order to select those to continue is feasible. This means that it is possible to discriminate between "good" and "bad" sample runs, and to increasing the correlation of the chosen run with the a-posteriori best one. In particular, the crude and very fast concept of diminishing returns has led to surprisingly good results. Another good approach was to fit the parameters of a perceptron to the observed data, using numerical black-box optimizers.

The challenge here arises from the fact that the very same algorithm on the very same instance can show significantly different behavior with intersecting performance profiles (see Figure 2). These then cause difficulties in choosing the right run to continue: depending on the total time budget, this causes a switch of the best decision maker from "currentBest" even to "currentWorst" in some cases. Theoretical results are needed to characterize this further in the context of stochastic algorithms, and this will be the subject of our future work.

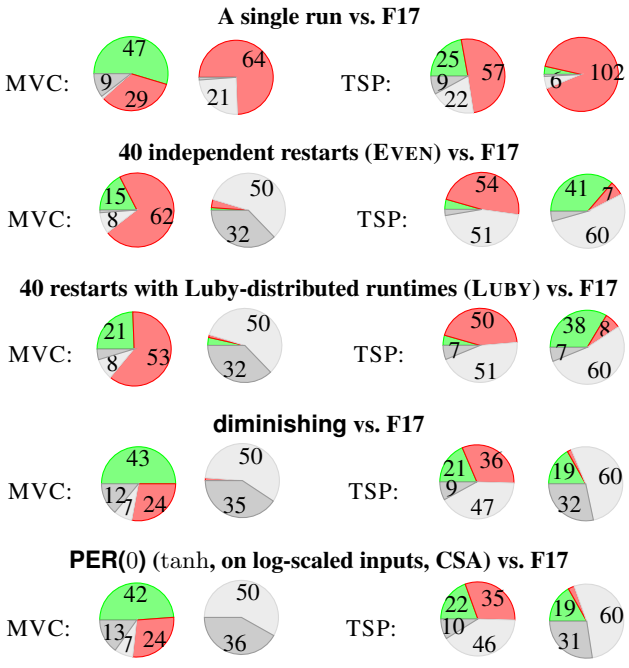


Figure 5: Statistical comparison of the best BET-AND-RUN configuration from (Friedrich, Kötzing, and Wagner 2017) (here named F17) with a subset of our approaches using the Wilcoxon rank-sum test (significance level $p = 0.05$) on 1'000 independent samples per setup. All decision makers are applied for $k = 40$, $m = 1$, $t_1 = 0.4t$, except for single run, EVEN and LUBY, which have $t_1 = t$. The approaches are compared based on the final quality gap to the best possible solution. Each pair of pie charts shows the outcomes for two extreme total time budgets: $t = 2s$ (left) and $t = 2'000s$ (right).
In short, the more green we see, the better the alternative is compared to F17. In detail, the colors have the following meaning: green indicates that the alternative is statistically better than F17, red indicates that the alternative is worse, light gray indicates that both performed identically, dark gray indicates that the differences were statistically insignificant.

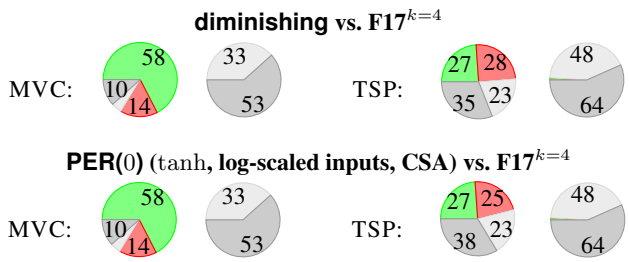


Figure 6: Comparison of predictors for $k = 4$, $m = 1$, $t_1 = 0.4t$. The style is identical to that of Figure 5. As a new reference, F17$^{k=4}$ corresponds to the previous F17 with $k = 4$.

To look a bit more into the future, we conjecture that per-instance configuration of BET-AND-RUN is possible. Our preliminary work in this direction indicates that this is indeed feasible, however, the very uneven heterogeneity of the instance features in combination with the small number of instances is currently posing a major challenge.

In further future work, we plan to relax two current restrictions: the approaches to date make their decisions purely based on solution quality and also consider just a single algorithm. Both aspects can be extended by giving the decision maker access to features of the solutions, and also by allowing for a diverse set of solvers (or configurations thereof) to participate in the overall optimization – with the overall goal to better exploit performance variance of solvers.

Acknowledgments

T. Weise acknowledges support by the National Natural Science Foundation of China under Grants 61673359, 61150110488, and 71520107002. We used Alexandre Devert's great implementation of SLPs, MLPs, SepCMA, and CSA (see http://github.com/marmakoide/jpack). M. Wagner acknowledges support by the ARC Discovery Early Career Researcher Award DE160100850 and by the Australia-China Young Scientist Exchange Program 2017.

References

Abu-Khzam, F.; Langston, M.; Shanbhag, P.; and Symons, C. 2006. Scalable parallel algorithms for FPT problems. *Algorithmica* 45:269–284.

Applegate, D.; Bixby, R.; Chvátal, V.; and Cook, W. 2007. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Applegate, D.; Cook, W.; and Rohe, A. 2003. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing* 15.

Arnold, D., and Beyer, H. 2008. Evolution strategies with cumulative step length adaptation on the noisy parabolic ridge. *Natural Computing* 7:555–587.

Biere, A., and Fröhlich, A. 2015. Evaluating CDCL restart schemes. In *International Workshop on Pragmatics of SAT (POS)*, 16. Preliminary version.

Cai, S.; Su, K.; Luo, C.; and Sattar, A. 2013. Numvc: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research* 46:687–716.

Cai, S.; Lin, J.; and Su, K. 2015. Two weighting local search for minimum vertex cover. In *29th AAAI Conference on Artificial Intelligence (AAAI'15)*, 1107–1113. AAAI Press.

Cai, S. 2015. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, 747–753. AAAI Press. Code: http://lcs.ios.ac.cn/~caisw/MVC.html, accessed 2018-06-18.

Ciré, A.; Kadioglu, S.; and Sellmann, M. 2014. Parallel restarted search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 842–848.

Cook, W. 2005. The traveling salesperson problem: Downloads (website). http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm, accessed 2018-06-18.

de Perthuis de Laillevault, A.; Doerr, B.; and Doerr, C. 2015. Money for nothing: Speeding up evolutionary algorithms through better initialization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'15)*, 815–822.

Fischetti, M., and Monaci, M. 2014. Exploiting erraticism in search. *Operations Research* 62:114–122.

Friedrich, T.; Kötzing, T.; and Wagner, M. 2017. A generic bet-and-run strategy for speeding up stochastic local search. In *31st AAAI Conference on Artificial Intelligence*, 801–807. AAAI Press.

Gagliolo, M., and Schmidhuber, J. 2011. Algorithm portfolio selection as a bandit problem with unbounded losses. *Annals of Mathematics and Artificial Intelligence* 61(2):49–86.

Gary, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company.

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

Gomes, F.; de Meneses, C.; Pardalos, P.; and Viana, G. 2006. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers & OR* 33:3520–3534.

Gutin, G., and Punnen, A., eds. 2002. *The Traveling Salesman Problem and its Variations*, volume 12 of *Combinatorial Optimization*. Kluwer.

Huang, J. 2007. The effect of restarts on the efficiency of clause learning. In *International Joint Conference on Artifical Intelligence (IJCAI'07)*, 2318–2323.

Kadioglu, S.; Sellmann, M.; and Wagner, M. 2017. Learning a reactive restart strategy to improve stochastic search. In *Learning and Intelligent Optimization – 11th International Conference (LION 11), Revised Selected Papers*, 109–123.

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

Lawler, E.; Lenstra, J.; Rinnooy Kan, A.; and Shmoys, D. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley.

Levenberg, K. 1944. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics* 2:164–168.

Li, Y.; Cai, S.; and Hou, W. 2017. An efficient local search algorithm for minimum weighted vertex cover on massive graphs. In *Proceedings of the 11th International Conference on Simulated Evolution and Learning (SEAL'17)*, 145–157.

Lissovoi, A.; Sudholt, D.; Wagner, M.; and Zarges, C. 2017. Theoretical results on bet-and-run as an initialisation strategy. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17)*, 857–864. ACM.

Liu, W.; Weise, T.; Wu, Y.; and Chiong, R. 2015. Hybrid ejection chain methods for the traveling salesman problem. In *Proceedings of the 10th International Conference on Bio-Inspired Computing – Theories and Applications (BIC-TA'15)*, volume 562 of *Communications in Computer and Information Science*, 268–282. Springer.

Lourenço, H.; Martin, O.; and Stützle, T. 2010. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science (ISOR)*. Springer. 363–397.

Luby, M.; Sinclair, A.; and Zuckerman, S. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters* 47:173–180.

Marquardt, D. 1963. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics* 11(2):431–441.

Martí, R. 2003. Multi-start methods. In *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science (ISOR)*. Springer. chapter 12, 355–368.

Nagata, Y., and Kobayashi, S. 2013. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing* 25:346–363.

Qi, Q.; Weise, T.; and Li, B. 2017. Modeling optimization algorithm runtime behavior and its applications. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17) Companion*, 115–116. ACM.

Qi, Q.; Weise, T.; and Li, B. 2018. Optimization algorithm behavior modeling: A study on the traveling salesman problem. In *Proceedings of the Tenth International Conference on Advanced Computational Intelligence (ICACI'18)*, 845–850. IEEE.

Reinelt, G. 1991. TSPLIB – a traveling salesman problem library. *ORSA Journal on Computing* 3(4):376–384. Instances: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/, accessed 2018-06-18.

Ros, R., and Hansen, N. 2008. A simple modification in CMA-ES achieving linear time and space complexity. In *10th International Conference on Parallel Problem Solving from Nature (PPSN X)*, volume 5199 of *Lecture Notes in Computer Science*, 296–305. Springer.

Samuelson, P., and Nordhaus, W. 2001. *Microeconomics*. McGraw-Hill.

Schoenauer, M.; Teytaud, F.; and Teytaud, O. 2012. A rigorous runtime analysis for quasi-random restarts and decreasing stepsize. In *10th International Conference Evolution Artificielle (EA'11), Revised Selected Papers*, 37–48. Springer.

Weise, T., and Wagner, M. 2018. Results of Bet-and-Run Strategies with Different Decision Makers on the Traveling Salesman Problem and the Minimum Vertex Cover Problem.

Weise, T.; Chiong, R.; Tang, K.; Lässig, J.; Tsutsui, S.; Chen, W.; Michalewicz, Z.; and Yao, X. 2014. Benchmarking optimization algorithms: An open source framework for the

traveling salesman problem. *IEEE Computational Intelligence Magazine* 9(3):40–52.

Whitley, D. 2016. Blind no more: Deterministic partition crossover and deterministic improving moves. In *Companion Material Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'16)*, 515–532. ACM.