A Generic Bet-and-run Strategy for Speeding Up Stochastic Local Search*

Tobias Friedrich[1], **Timo Kötzing**[1], **Markus Wagner**[2]
[1]Hasso Plattner Institute, Potsdam, Germany
[2]Optimisation and Logistics, The University of Adelaide, Adelaide, Australia

Abstract

A common 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. However, while specific restart strategies have been developed for specific problems (and specific algorithms), restarts are typically not regarded as a general tool to speed up an optimization algorithm. In fact, many optimization algorithms do not employ restarts at all.

Recently, *bet-and-run* was introduced in the context of mixed-integer programming, where first a number of short runs with randomized initial conditions is made, and then the most promising run of these is continued. In this article, we consider two classical NP-complete combinatorial optimization problems, traveling salesperson and minimum vertex cover, and study the effectiveness of different bet-and-run strategies. In particular, our restart strategies do not take any problem knowledge into account, nor are tailored to the optimization algorithm. Therefore, they can be used off-the-shelf. We observe that state-of-the-art solvers for these problems can benefit significantly from restarts on standard benchmark instances.

1 Introduction

When a desktop PC is not working properly, the default answer of an experienced system administrator is restarting it. The same holds for stochastic algorithms and randomized search heuristics: If we are not satisfied with the result, we might just try restarting the algorithm again and again. While this is well-known [17, 19], very few algorithms directly incorporate such restart strategies. We assume that this is due to the added complexity of designing an appropriate restart strategy that is advantageous for the considered algorithm.

Hence, it would be beneficial to have a generic framework for restart strategies which is not overly dependent on the exact algorithm used or the problem under consideration. In this paper we want to show that there are restart strategies which are of benefit in a variety of settings.

There are some theories on how to choose optimal restart strategies, independently of the setting. For example, Luby, Sinclair, and Zuckerman [18] 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 an 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. These results have the appeal that they can be used for every problem setting; however, they only apply to Las Vegas algorithms.

For the case of optimization, the situation is much less clear, with plenty of different approaches presented by the stochastic optimization community. A gentle introduction to practical approaches for such restart strategies is given by Marti [19] and Loureno et al. [17], and a recent theoretical result is presented by Schoenauer, Teytaud, and Teytaud [24]. Particularly for the satisfiability problem (SAT), there are several studies that make an empirical comparison of a number of restart policies [5, 15]. These show the substantial impact of the restart policy on the efficiency of SAT solvers. In the context of satisfiability problems this might be unsurprising as state-of-the-art SAT and CSP solvers often speed up their search by learning "no-goods" during backtracking [10].

Classical optimization algorithms are often deterministic and thus cannot be improved by restarts (neither their run time nor their outcome will alter). This also appears to hold for certain popular modern solvers. For example, one might argue that the underlying algorithm of IBM ILOG CPLEX is not random, however, when memory constraints or parallel computations come into play, the characteristics change. This was the initial idea of Lalla-Ruiz and Voß [16], when they 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. Thus, the initial solution often strongly influences the quality of the outcome. It follows that it is natural to do several runs of the algorithm. 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 and Monaci [13] investigated the use of the following *bet-and-*

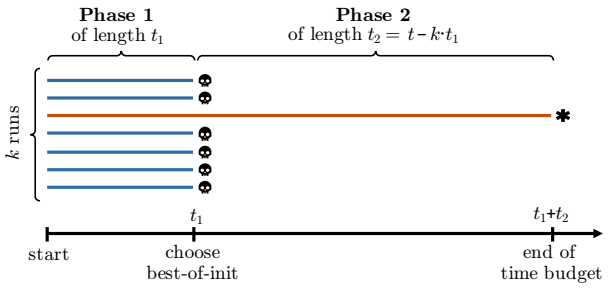

Figure 1: Our *bet-and-run* restart strategy starts with k independent runs and total time budget t. After time t_1 all but the best run are terminated (marked with 💀). The best run (marked with ✱) continues for t_2 time steps until the total time budget runs out.

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.

This strategy is illustrated in Figure 1. 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 and Monaci [13] 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 with $k = 5$, bet-and-run was typically beneficial. de Perthuis de Laillevault, Doerr, and Doerr [12] have recently 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 $\Omega(\sqrt{n})$ [12]. They also rigorously showed that the optimal gain is achieved for some k of order $k = \sqrt{n}$.

In this paper we want to show that there is *no need* to tailor the restart strategy or to access the internal mechanisms of available solvers: in fact there are generic *bet-and-run* restart strategies that consistently outperform single-run and multi-run strategies *across different domains*! We benchmark our strategies on two different problems: traveling sales person (TSP) and minimum vertex cover (MVC). We observe statistically significant improvements of our bet-and-run strategy on standard corpora for state-of-the-art solvers for both optimization problems.

Details for our design choices can be found in Section 2, along with a formal definition of the problems. Since it is a priori not obvious what *bet-and-run* strategies are most promising, we define a generic scheme of restart strategies in Section 3; we compare these strategies in Section 4, where we find 14 parameter settings for the *bet-and-run* strategy (k and t_1) that are representative of the space of possible parameter settings. Finally, in Section 5, we show that there are *bet-and-run* strategies that perform well across a wide range of instances in different domains.

2 Problems and Benchmarks

In the following we briefly introduce the two NP-complete problems we consider, as well as the corresponding solvers and benchmarks used in this paper.

Traveling Salesperson

The Traveling Salesperson problem considers an edge-weighted graph $G = (V, E, w)$, the vertices $V = \{1, \ldots, n\}$ are referred to as *cities*. It asks for a permutation π 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.

Applications of the traveling salesperson problem arise naturally in areas like planning and logistics [20], but they are also encountered in a large number of other domains; the textbook by Applegate, Bixby, Chvatal, and Cook [3] gives an overview of such encounters, listing areas as diverse as genome sequencing, drilling problems, aiming telescopes, and data clustering. TSP is identified as one of the most important (and most studied) optimization problems.

We use the Chained-Lin-Kernighan (CLK) heuristic [2], 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 CLK code is available online [11]. Despite being a few years old, CLK still holds the records for a number of large TSPlib instances.

The TSPlib is a classic repository of TSP instances [21], which are available online [22]. For our first investigations, we pick from TSPlib the nine largest symmetric instances which have between 5,934 and 85,900 cities, and the Mona Lisa TSP Challenge instance [6], which contains 100,000 cities. In summary, the instances are rl5934, pla7397, rl11849, usa13509, brd14051, d15112, d18512, pla33810, pla85900, and mona-lisa100k. For the first seven instances, CLK takes less than 0.3 seconds to initialize. For the remaining three instances, the initialization times are 0.5, 2.5, and 2.5 seconds.

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 $\{u, v\} \in E$,

$$u \in S \text{ or } v \in S.$$

The NP-complete decision problem k-vertex cover decides whether a vertex cover of size k exists. We consider the optimization problem which aims at finding a vertex cover of minimum size.

Applications of the vertex cover problem arise in various areas like network security, scheduling and VLSI design [14]. To give an example, finding a minimum vertex cover in a network corresponds to locating an optimal set of nodes on which to strategically place controllers such that they can monitor the data going through every link in the network. The vertex cover problem is also closely related to the question of finding a maximum clique. This has a range of applications in bioinformatics and biology, such as identifying related protein sequences [1].

Over the past two decades, numerous algorithms have been proposed for solving the vertex cover problem. We choose FASTVC [7] over the popular NUMVC [9] 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. The code of FASTVC is available online [8].

For our initial experimental investigations, we select from the 86 instances used by Cai [7] the 10 instances for which FASTVC outperforms NUMVC the most. With this approach to instance selection (which differs from the one we used for TSP) we attempt to further increase the performance gap between both algorithms. In the order of increasing performance difference, these instances are rec-amazon, large-soc-gowalla, soc-digg, sc-shipsec1, soc-youtube, sc-shipsec5, soc-flickr, soc-youtube-snap, web-it-2004, and ca-coauthors-dblp. On all instances, FASTVC's initialisation takes at most 1.5 seconds. All instances are available online [23].

3 Restart Strategies

A restart strategy describes how the total time budget is distributed over a number of independent runs. We consider two different kinds of restart strategies as follows. On the one hand, we consider the *bet-and-run* strategies where all initial runs have the same length. On the other hand, we are inspired by Luby et al. [18] for defining a kind of restart strategy with different lengths in an attempt to be more robust with respect to choosing the right time s for the initial runs. Luby et al. [18] define a simple universal strategy, defined as an (infinite) sequence indicating how many time units should be used for each run. This sequence is given as

$$S^{\text{univ}} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1, \ldots)$$

and, more formally, by $\forall i \geq 1$,

$$S^{\text{univ}}(i) = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1; \\ S^{\text{univ}}(i - 2^{k-1} + 1), & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

The numbers given in the Luby sequence refer to the number of time units to employ per run. Thus, in order to use the Luby sequence, we have to define the length of this time unit, which we will call *Luby time unit*. We indicate it as $x\%$ of the total time budget. Thus, we get the following definition.

RESTARTS$^{\text{k}}_{\text{x\%}}$ refers to the strategy where k initial runs are performed, and each of the runs has a computational budget of $x\%$ of the total time budget.

RESTARTSLUBY$^{\text{k}}_{\text{x\%}}$ refers to the strategy that uses in its first phase runs whose lengths are defined by the Luby sequence. k refers to the sequence length used in the first phase, and each Luby time unit is $x\%$ of the total time.

4 Choosing Representative Restart Strategies

In the following, we investigate different *bet-and-run* strategies for different instances. For each combination of algorithm, instance, and overall run time budget, we average the outcomes of 100 independent repetitions on a compute cluster with Intel Xeon E5620 CPUs (2.4GHz).[1] The benefit of using 100 repetitions is that the standard error of the mean is only 10% of the sample's standard deviation, which means that the resulting averages are reasonably accurate representatives of the actual average performance, despite the algorithms' randomized nature. This in turn allows us to draw conclusions about the average advantage of our approach across different instances and problem domains.

For each heatmap we use one fixed total time budget, and we systematically vary the number of runs in the first phase and their run times. In each plot we show a diagonal line that indicates the schemes RESTARTS$^{\text{x}}_{\text{1/x}}$, which corresponds to performing x independent runs with each $1/x$-th of the total budget. Every scheme above this line would violate the total time budget, which gives the heatmaps a triangular shape.

Before we come to the discussion of the experimental results, a note regarding the implementation. Often algorithm implementations do not allow us to pause and continue their operation at arbitrary points in time, or to provide initial solutions together with a full internal state of the algorithm. While both options are implementable, the source code is not always available, or it cannot be easily modified. Therefore, we employ a trick that can easily be applied *if* the implementation accepts run time limits and seeds for the random number generation. In our investigations, we first execute all runs of the first phase sequentially, each with the respective allotted time budget. Then we determine the best performing run b that used time t_b and the randomly set seed s_b. In order to complete the restart scheme's second phase, we run b not just with the remaining time budget t_2 (see Figure 1), but we restart it from scratch with $t_b + t_2$ and with the previously used seed s_b. This allows the algorithm to reach its previous state after t_b (which we do not count toward the total time budget) and to continue for t_2.

Figure 2 depicts how the total budget influences the relative performance of the restart strategies on a TSP instance. For a small total time budget we see that 4 to 10 short initial runs are best; with an increase in the budget, more and more strategies with even longer initial run times perform better than the single-run strategy. Also, it should be noted that when the number of initial runs or the time budget for them increases too much, the performance of the scheme deteriorates quickly.

Similar observations also hold for our restart scheme on MVC instances, as shown in Figure 3. Also, we see that the

[1] Our code and results have been made publicly available: `http://bitbucket.org/markuswagner/restarts`

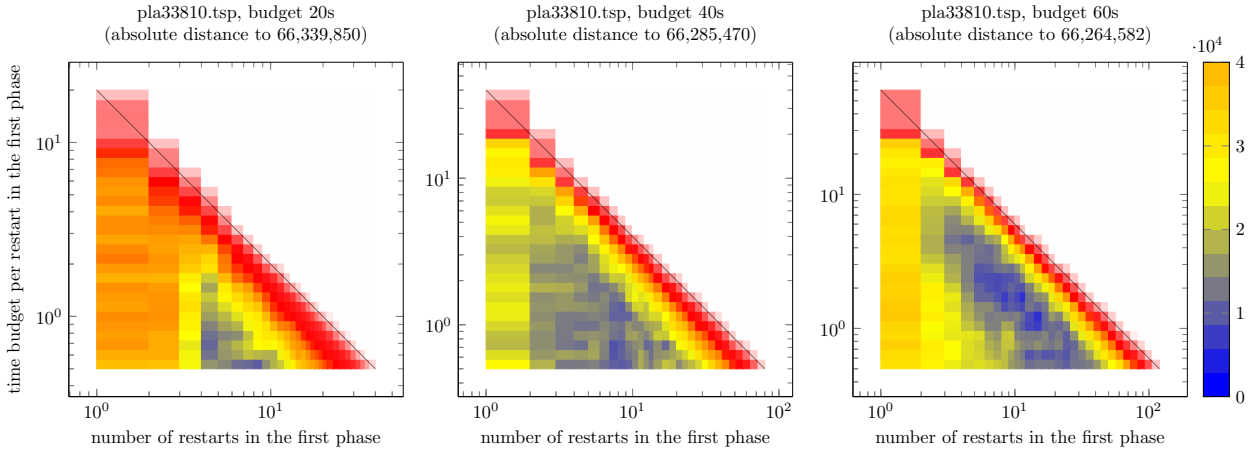

Figure 2: Comparison of different restart strategies $\text{RESTARTS}^{k}_{x\%}$ and total time budgets for the TSP instance pla33810. The plots show the average quality of the solution discovered with restart strategy $\text{RESTARTS}^{k}_{x\%}$ compared to the best average found in all runs (smaller values are better). The best average is shown in the title of each heatmap. For example in the leftmost plot, the x-axis shows the number of runs $k = 1 \ldots 40$ in the first phase, and the y-axis shows the time budget per restart in the first phase in seconds. The black diagonal is the line $k \cdot x = 100\%$ of regular full-restart strategies with no best-of phase. The color is chosen depending on the average distance to the best average of 100 independent repetitions, and the cells are colored based on the average of the corners. In summery, we observe that restart strategies which perform a few short runs perform better on average than e.g. no restarts (=only one run) or full-restarts (=diagonal line).

Luby time unit has an impact on the overall performance of the approach, as does the length of the Luby sequence used.

In summary, no single *bet-and-run* performs best across both problem domains. However, there are always schemes that outperform the naive scheme with just a single run, giving clear evidence for an advantage of our *bet-and-run* approach.

For our general study of restart strategies (across different problems and instances), we use the following diverse set of 14 strategies that vary in the number of runs used in Phase 1 and in the run time allocated to each run:[2]

- Phase 1 takes 100% of the total time
  - $\text{RESTARTS}^{1}_{100\%}$: 1 regular run
  - $\text{RESTARTS}^{4}_{25\%}$: 4 runs, 25% each
- Phase 1 takes 40% of the total time
  - $\text{RESTARTS}^{4}_{10\%}$: 4 runs with 10% each
  - $\text{RESTARTS}^{10}_{4\%}$: 10 runs with 4% each
  - $\text{RESTARTS}^{40}_{1\%}$: 40 runs with 1% each
- Phase 1 takes 10% of the total time
  - $\text{RESTARTS}^{4}_{2.5\%}$: 4 runs with 2.5% each
  - $\text{RESTARTS}^{10}_{1\%}$: 10 runs, each with 1% each
  - $\text{RESTARTS}^{40}_{0.25\%}$: 40 runs, each with 0.25% each
- Phase 1 takes 4% of the total time
  - $\text{RESTARTS}^{4}_{1\%}$: 4 runs with 1% each
  - $\text{RESTARTS}^{10}_{0.4\%}$: 10 runs with 0.4% each
  - $\text{RESTARTS}^{40}_{0.1\%}$: 40 runs with 0.1% each
- Three Luby-based strategies
  - $\text{RESTARTSLUBY}^{4}_{1\%}$: Luby sequence length 4 (5 units in total)
  - $\text{RESTARTSLUBY}^{10}_{1\%}$: Luby sequence length 10 (16 units in total)
  - $\text{RESTARTSLUBY}^{40}_{1\%}$: Luby sequence length 40 (96 units in total)

[2]Note that values like 1, 2.5, 4, and 10 were picked because they are easily human readable.

In all cases, Phase 2 continues with the *bet-and-run* found in Phase 1. In Figure 4 we show for one representative instance the points in the RESTARTS-landscape that we will be investigating subsequently.

5 Cross Problem Study

A crucial decision is the *total time budget* allotted for each instance. If the time limit is too short, no strategy has enough time to finish even its initialization. If the time limit is too long, the differences between the strategies might vanish. To investigate the impact of different total run time budgets, we consider five different budgets. These budgets are all relative to the time t_{init} needed to initialize the algorithm with the given instance. The overall run time budgets that we consider are $100 \cdot t_{\text{init}}$, $400 \cdot t_{\text{init}}$, $1\,000 \cdot t_{\text{init}}$, $4\,000 \cdot t_{\text{init}}$, and $10\,000 \cdot t_{\text{init}}$.

As an example we present Figure 5 to show the results of the impact of the total run time budget, considering the minimum vertex cover instance sc-shipsec5. It is clearly visible that a single run without restarts has the worst performance. This configuration is outperformed by all others, even the one where four independent runs are given just 25% of the total computation budget. These observations hold independently of the chosen total budget. When relatively little time is available (e.g. $100 \cdot t_{\text{init}}$), the performance of the different restart schemes varies significantly. However, the differences between our different schemes seem to disappear with

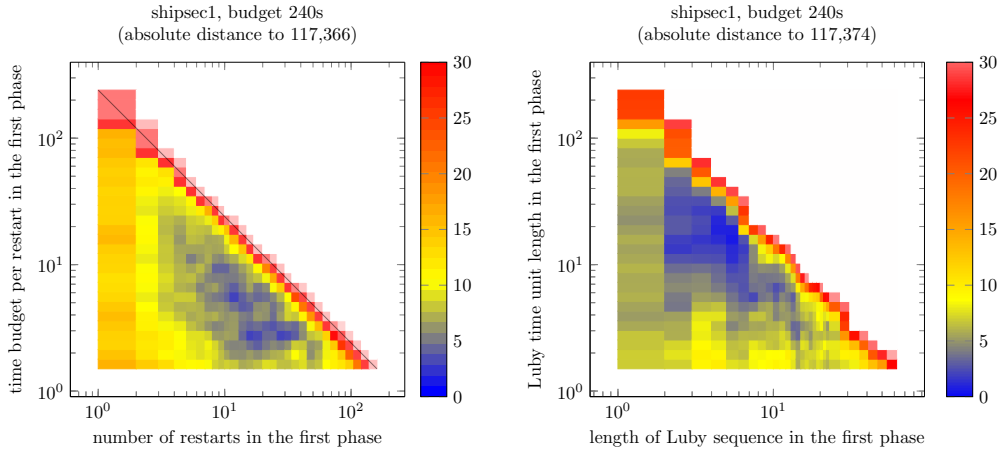

Figure 3: For the MVC instance shipsec1, we show the RESTARTS schemes on the left, and the RESTARTSLUBY scheme on the right. For more details of the presentation, see Figure 2.

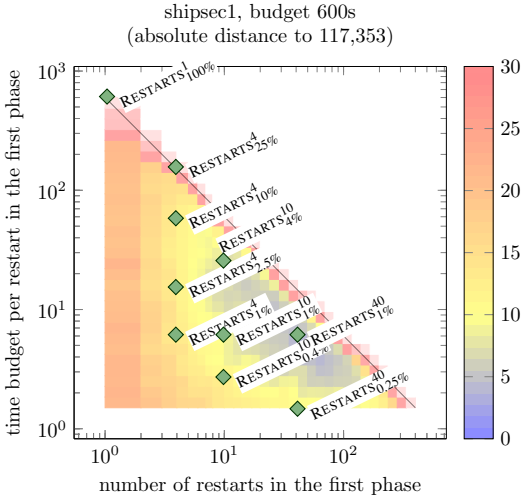

Figure 4: Illustration of the representative restart strategies chosen in Section 4 and compared with each other in Section 5. Each strategy is marked with an ◆ . The background shows the MVC instance shipsec1 ($400 \cdot t_{\text{init}}$). Note that the entire leftmost column represents the scheme $\text{RESTARTS}^{1}_{100\%}$. The strategy $\text{RESTARTS}^{*}_{0.1\%}$ is not defined here due to the overall budget being $400 \cdot t_{\text{init}}$.

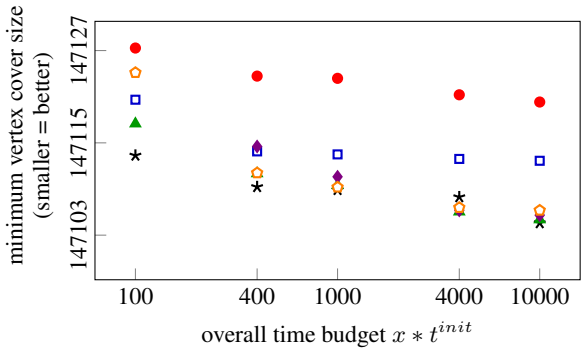

Figure 5: Solution quality achieved for sc-shipsec5 by different schemes for five different total time budgets. Shown are the averages of 100 independent runs of six different strategies: ● one regular run, ▫ four runs with 25% of the time budget each, ⋆ $\text{RESTARTS}^{10}_{4\%}$, ▲ $\text{RESTARTS}^{10}_{1\%}$, ♦ $\text{RESTARTS}^{10}_{0.4\%}$, and ⬠ $\text{RESTARTSLUBY}^{10}_{4\%}$. Note that ♦ $\text{RESTARTS}^{10}_{0.4\%}$ is not defined for $100 \cdot t_{\text{init}}$. It is also curious to observe that the best restart strategy ⋆ with time budget $100 \cdot t_{\text{init}}$ outperforms a single-run strategy ● with time budget $10000 \cdot t_{\text{init}}$.

increasing time budget, and the restart schemes are able to find smaller and smaller vertex covers.

In order to allow all restart strategies introduced in Section 4 to have a fair chance to finish at least the initialization in each run, we have to make sure that each run of Phase 1 gets at least time t_{init}. For example for $\text{RESTARTS}^{40}_{0.1\%}$ this implies that the total time budget has to be at least $1000 \cdot t_{\text{init}}$. As computational resources are the bottleneck for our subsequent studies, we focus on the shortest three total time budgets: $100 \cdot t_{\text{init}}$, $400 \cdot t_{\text{init}}$ and $1000 \cdot t_{\text{init}}$.

In our first *cross problem domain study* we determine the average rank of the 14 restart strategies described in Section 4 for the two optimization problems. For each of the 20 instances listed in Section 2 we perform 100 independent runs. Based on the results, we then determine the relative ranks of the 14 restart strategies for both total time budgets.

Figure 6 shows the average ranks, which are reflecting the trends that we have previously seen in the heatmaps. For the two different problem domains, we observe the following:

- For TSP it is best to use a relatively large fraction (40%) of the total time budget with 4 to 40 runs in the first phase. If less time is used for the initial runs, then the average rank worsens quickly.
- For the MVC instances, the range of effective budgets for the first phase is wider, it covers the range from 4–40%. However, schemes with only a few runs perform the worst.

In all cases, the *bet-and-run* approaches clearly outperform the commonly used single-run strategy, which ends

Budget: $100 \cdot t_{\text{init}}$	TSP	MVC
$\text{RESTARTS}^{1}_{100\%}$	8.1	6.8
$\text{RESTARTS}^{4}_{25\%}$	8.2	4.0
$\text{RESTARTS}^{4}_{10\%}$	1.7	3.2
$\text{RESTARTS}^{10}_{4\%}$	3.0	2.6
$\text{RESTARTS}^{40}_{1\%}$	5.3	2.3
$\text{RESTARTS}^{4}_{2.5\%}$	4.7	2.6
$\text{RESTARTS}^{10}_{1\%}$	5.0	4.4
$\text{RESTARTS}^{4}_{1\%}$	5.6	5.9
$\text{RESTARTSLUBY}^{4}_{1\%}$	7.1	6.6
$\text{RESTARTSLUBY}^{10}_{1\%}$	6.9	4.1
$\text{RESTARTSLUBY}^{40}_{1\%}$	10.4	4.1

Budget: $400 \cdot t_{\text{init}}$	TSP	MVC
$\text{RESTARTS}^{1}_{100\%}$	12.3	9.4
$\text{RESTARTS}^{4}_{25\%}$	3.0	5.4
$\text{RESTARTS}^{4}_{10\%}$	3.7	4.4
$\text{RESTARTS}^{10}_{4\%}$	2.3	1.8
$\text{RESTARTS}^{40}_{1\%}$	3.0	1.3
$\text{RESTARTS}^{4}_{2.5\%}$	6.2	5.2
$\text{RESTARTS}^{10}_{1\%}$	5.6	3.2
$\text{RESTARTS}^{40}_{0.25\%}$	7.7	3.7
$\text{RESTARTS}^{4}_{1\%}$	9.9	6.5
$\text{RESTARTS}^{10}_{0.4\%}$	9.0	4.5
$\text{RESTARTSLUBY}^{4}_{1\%}$	10.8	6.4
$\text{RESTARTSLUBY}^{10}_{1\%}$	10.3	3.5
$\text{RESTARTSLUBY}^{40}_{1\%}$	7.2	2.5

Budget: $1000 \cdot t_{\text{init}}$	TSP	MVC
$\text{RESTARTS}^{1}_{100\%}$	13.7	10.1
$\text{RESTARTS}^{4}_{25\%}$	5.1	6.9
$\text{RESTARTS}^{4}_{10\%}$	5.6	6.6
$\text{RESTARTS}^{10}_{4\%}$	1.9	2.8
$\text{RESTARTS}^{40}_{1\%}$	2.2	1.0
$\text{RESTARTS}^{4}_{2.5\%}$	6.9	5.8
$\text{RESTARTS}^{10}_{1\%}$	6.0	2.4
$\text{RESTARTS}^{40}_{0.25\%}$	7.1	2.5
$\text{RESTARTS}^{4}_{1\%}$	10.0	5.9
$\text{RESTARTS}^{10}_{0.4\%}$	9.9	3.8
$\text{RESTARTS}^{40}_{0.1\%}$	11.0	4.9
$\text{RESTARTSLUBY}^{4}_{1\%}$	12.8	7.0
$\text{RESTARTSLUBY}^{10}_{1\%}$	9.8	3.4
$\text{RESTARTSLUBY}^{40}_{1\%}$	3.0	2.2

Figure 6: Average rank (smaller values are better) of the different restart strategies for the two optimization problems with three total time budgets. Strategies that use the same total time for the first phase are grouped together, as are the ones based on the Luby sequence. The colors correspond to the average rank of a scheme (colder colors are better). The two *bet-and-run* strategies $\text{RESTARTS}^{10}_{4\%}$ and $\text{RESTARTS}^{40}_{1\%}$ have the best average rank. A single-run with no restarts has the worst average rank.

up on one of the worst ranks. When considering the average performance across all total time limits, our schemes $\text{RESTARTS}^{40}_{1\%}$ and $\text{RESTARTS}^{10}_{4\%}$ can almost be considered universal for the given instances and solvers. For the TSP and the MVC, they achieve the best or second best rankings. The universal sequence of Luby et al. [18] turned out inferior compared to restarts of fixed length, which matches the earlier studies on the decision version of SAT/UNSAT problems by Audemard and Simon [4].

Lastly, we investigate the broader applicability of the best performing strategy $\text{RESTARTS}^{40}_{1\%}$ when the total time limit was $1000 \cdot t_{\text{init}}$. We apply it to the 86 MVC instances used in [7], which come from 10 categories of networks, and to the 111 symmetric TSP instances from TSPlib, which cover geographical instances as well as circuit board layouts. As before, we repeat each experiment 100 times independently in order to get reasonable estimates of the performance distribution. The results in Figure 7 show that on almost all instances, the standard run is outperformed by our bet-and-run strategy $\text{RESTARTS}^{40}_{1\%}$.

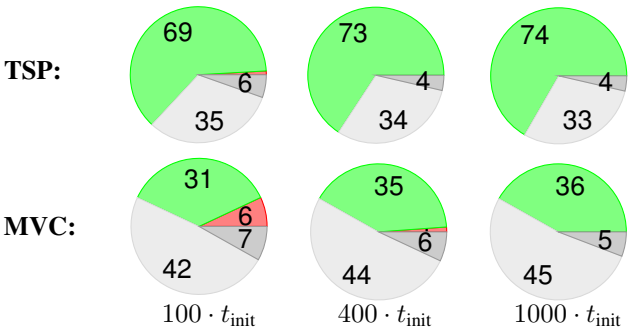

Figure 7: Statistical comparison of $\text{RESTARTS}^{40}_{1\%}$ and $\text{RESTARTS}^{1}_{100\%}$ (no restarts) with Wilcoxon rank-sum test (significance level $p = 0.05$) for both problems and three total time budgets. The colors have the following meaning: Green indicates that $\text{RESTARTS}^{40}_{1\%}$ is statistically better, Red indicates that $\text{RESTARTS}^{40}_{1\%}$ is statistically worse, Light gray indicates that both performed identical, Dark gray indicates that the differences were statistically insignificant. Overall, the solutions for most problem instances were either improved or stayed unchanged by introducing our bet-and-run strategy. Worsenings due to our bet-and-run strategy do not occur for the largest time budget ($1000 \cdot t_{\text{init}}$). Within the medium time budget ($400 \cdot t_{\text{init}}$), 0 out of 111 TSP-instances and 1 out of 86 MVC-instances got worse. Within the smallest time budget ($100 \cdot t_{\text{init}}$), 1 out of 111 TSP-instances and 6 out of 86 MVC-instances got worse.

6 Conclusions and Future Work

We study a generic *bet-and-run* restart strategy, which is easy to implement as an additional speed-up heuristic for solving difficult optimization problems. We demonstrate its efficiency on two classical NP-complete optimization problems with *state-of-the-art solvers*. Our experiments show a significant advantage of *bet-and-run* strategies on all problems. The best strategy overall 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. The universal sequence of Luby et al. [18] turned out inferior.

The gain achieved by our *bet-and-run* strategy differs depending on the studied optimization problem. For both TSP and MVC the gain is significant.

As the two problem domains are structurally different, we expect that *bet-and-run* strategies are generally helpful. Future research should study further classes of optimization problems such as multi-objective problems or continuous domains. While we focus on strategies with two phases only, it is interesting to consider iterated or hierarchical best-of strategies. Another direction are dynamic *bet-and-run* strategies, which restart runs that stop improving.

References

[1] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons. Scalable parallel algorithms for FPT problems. *Algorithmica*, 45:269–284, 2006.

[2] D. L. Applegate, W. J. Cook, and A. Rohe. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15:82–92, 2003.

[3] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2011.

[4] G. Audemard and L. Simon. *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP)*, chapter Refining Restarts Strategies for SAT and UNSAT, pp. 118–126. Springer, 2012.

[5] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 28–33, 2008.

[6] R. Bosch. Mona Lisa TSP Challenge (Website). `http://www.math.uwaterloo.ca/tsp/data/ml/monalisa.html`, 2009. [Online; accessed 11-September-2016].

[7] S. Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 747–753, 2015.

[8] S. Cai. Local Search for Minimum Vertex Cover (Website). `http://lcs.ios.ac.cn/~caisw/MVC.html`, 2015. [Online; accessed 11-September-2016].

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

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

[11] W. Cook. The Traveling Salesperson Problem: Downloads (Website). `http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm`, 2003. [Online; accessed 11-September-2016].

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

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

[14] F. C. Gomes, C. N. Meneses, P. M. Pardalos, and G. V. R. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Compututers and Operations Research*, 33: 3520–3534, 2006.

[15] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI)*, pp. 2318–2323, 2007.

[16] E. Lalla-Ruiz and S. Voß. Improving solver performance through redundancy. *Journal of Systems Science and Systems Engineering*, 25:303–325, 2016.

[17] H. R. Loureno, O. C. Martin, and T. Stützle. Iterated local search: Framework and applications. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, pp. 363–397. Springer US, 2010.

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

[19] R. Marti. Multi-start methods. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pp. 355–368. 2003.

[20] M. Polacek, K. F. Doerner, R. F. Hartl, G. Kiechle, and M. Reimann. Scheduling periodic customer visits for a traveling salesperson. *European Journal of Operational Research*, 179:823–837, 2007.

[21] G. Reinelt. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3:376–384, 1991.

[22] G. Reinelt. Symmetric traveling salesman problem: TSP data (Website). `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`, 2008. [Online; accessed 11-September-2016].

[23] R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization (Website). `http://networkrepository.com`, 2015. [Online; accessed 11-September-2016].

[24] M. Schoenauer, F. Teytaud, and O. Teytaud. A rigorous runtime analysis for quasi-random restarts and decreasing stepsize. In *Artificial Evolution*, pp. 37–48. Springer, 2012.