Chapter 18
Computational Comparison of Metaheuristics

John Silberholz, Bruce Golden, Swati Gupta, and Xingyin Wang

Abstract Metaheuristics are truly diverse in nature—under the overarching theme of performing operations to escape local optima, algorithms as different as ant colony optimization, tabu search, harmony search, and genetic algorithms have emerged. Due to the unique functionality of each type of metaheuristic, the computational comparison of metaheuristics is in many ways more difficult than other algorithmic comparisons. In this chapter, we discuss techniques for the meaningful computational comparison of metaheuristics. We discuss how to create and classify instances in a new testbed and how to make sure other researchers have access to these test instances for future metaheuristic comparisons. In addition, we discuss the disadvantages of large parameter sets and how to measure complicated parameter interactions in a metaheuristic's parameter space. Finally, we explain how to compare metaheuristics in terms of both solution quality and runtime and how to compare parallel metaheuristics.

J. Silberholz
Ross School of Business, University of Michigan, Ann Arbor, MI, USA
e-mail: josilber@umich.edu

B. Golden (✉)
R. H. Smith School of Business, University of Maryland, College Park, MD, USA
e-mail: bgolden@rhsmith.umd.edu

S. Gupta
Simons Institute for the Theory of Computing, UC Berkeley, CA, USA
e-mail: swatig@alum.mit.edu

X. Wang
Engineering Systems and Design, Singapore University of Technology and Design, Singapore, Singapore
e-mail: xingyin_wang@sutd.edu.sg

M. Gendreau, J.-Y. Potvin (eds.), *Handbook of Metaheuristics*,
International Series in Operations Research & Management Science 272,
https://doi.org/10.1007/978-3-319-91086-4_18

18.1 Introduction

Metaheuristics are truly diverse in nature—under the overarching theme of performing operations to escape local optima (we assume minima in this chapter without loss of generality), algorithms as different as ant colony optimization, tabu search, harmony search, and genetic algorithms have emerged. Due to the unique functionality of each type of metaheuristic, the computational comparison of metaheuristics is in many ways more difficult than other algorithmic comparisons. For example, if we compare two exact solution procedures, we can focus solely on runtime. With metaheuristics, we must compare with respect to both solution quality and runtime; these measures are influenced by the selected parameter values. It is also the case that, unlike simple heuristics, metaheuristics may be difficult to replicate by another researcher.

In this chapter, we discuss techniques for the meaningful computational comparison of metaheuristics. In Sect. 18.2, we discuss how to create and classify instances (e.g., based on source (real-world vs. artificial), size (large vs. small), difficulty (hard vs. easy), and specific instance features (such as the distribution of item weights in bin packing) in a new testbed and how to make sure other researchers have access to these test instances for future metaheuristic comparisons. In Sect. 18.3, we discuss the disadvantages of large parameter sets and how to measure complicated parameter interactions in a metaheuristic's parameter space. In Sects. 18.4 and 18.5, we discuss how to compare metaheuristics in terms of both solution quality and runtime. Finally, in Sect. 18.6, we discuss how to compare parallel metaheuristics.

We point out that we do not discuss multi-objective metaheuristics (MOMHs) in this chapter, although many of the ideas presented here are applicable to MOMHs. We refer the interested reader to articles [11, 29, 32, 61].

18.2 The Testbed

One of the most important components of a meaningful comparison among metaheuristics is the set of test instances or the testbed. The heterogeneity of test instances is key to identifying instance spaces where one metaheuristic might outperform the other. In this section, we will highlight the nuances of using existing testbeds, augmenting them with diverse new test instances, and using instance characteristics to systematically compare metaheuristics.

18.2.1 Using Existing Testbeds

When comparing a new metaheuristic to existing ones, it is advantageous to test on the problem instances already tested by previous papers. Then, results will be comparable on a by-instance basis, allowing relative gap calculations between the two

heuristics. Additionally, the trends in the performance of the new metaheuristic on existing testbeds can help in providing insights to the behavior of the metaheuristic.

18.2.2 Developing New Testbeds

While ideally testing on an existing testbed should be sufficient, there are many cases when this is either not possible or not sufficient. For instance, when writing a metaheuristic for a new problem, there will be no testbed for that problem, so a new one will need to be developed. In addition, even on existing problems where heuristic solutions were tested on non-published, often randomly generated problem instances, such as those presented in [23] and [44], a different testbed will need to be used. Last, if the existing testbed is insufficient (often due to containing instances that are too simple or too homogeneous) to effectively test a heuristic, a new one will need to be developed. Given the increases in available computing power observed through time, it is often the case that a difficult instance from 10 years ago may be simple today, necessitating the development of more challenging instances.

18.2.2.1 Goals in Creating the Testbed

The goals of a problem suite include mimicking real-world problem instances while providing test cases that are of various types and difficulty levels. Further, if one metaheuristic outperforms all others on the testbed then it is important to add new test instances, as one would expect that no metaheuristic can be best on all instances by the no free lunch theorems [67]. As an example of the value of a broad testbed, the authors of [19] show that for the NP-hard Max-Cut and Quadratic Unconstrained Binary Optimization problems, 23 heuristics out of the 37 heuristics they tested were not the best heuristic for any instance in the standard testbed but outperformed all the other heuristics on at least one instance when the standard testbed was expanded to include a more heterogeneous set of instances. Thus, the testbed used for evaluating and comparing heuristics or more sophisticated metaheuristics must be heterogeneous so that the performance over the testbed reflects the weaknesses and strengths of metaheuristics.

In order to generate heterogenous test instances, it is common to define a set of instance features and to cover the feasible feature space. For graph-related problems, common practice in the literature includes using various random graph generators like the machine-independent generator Rudy [51], the Python NetworkX library [25], and the Culberson random graph generators [15]. These random graph generators can be sampled appropriately such that the constructed instances have a desired range of various feature values, like average degree, connectivity, etc. To estimate which types of instances should be included in a testbed, one can either visualize the instance space projected down to a two-dimensional plane across the most predictive features [55] to check for instance types that are underrepresented or estimate the

coverage of normalized features (in [0,1]) as the fraction of the interval covered by the testbed[19]. The missing feature values can then be included in the testbed using appropriately parameterized random graph generation. In a recent line of work, genetic algorithms have also been used to evolve random instances until they have features in the desired range [55].

Another key requirement of the testbed that is especially important in the testing of metaheuristics is that large problem instances must be tested. For small instances, optimal solution techniques often run in reasonable runtimes and they generate a guaranteed optimal solution. It is, therefore, critical that metaheuristic testing occurs on the large problems for which optimal solutions cannot be calculated in reasonable runtimes using known techniques. As discussed in [28], it is not enough to test on small problem instances and extrapolate the results for larger instances; algorithms can perform differently in both runtime and solution quality on large problem instances.

While it is desirable that the new testbed be based on problem instances found in industrial applications of the problem being tested (like the TSPLib [50]), it is typically time intensive to do this sort of data collection. Often real-world data is proprietary and, therefore, difficult to obtain and potentially not publishable [45]. Still, capturing real aspects of a problem is important in developing a new testbed. For example, in the problem instances found in [21], the clustering algorithm placed nodes in close proximity to each other in the same cluster, capturing real-life characteristics of this problem.

It is more common to create a testbed based on existing well-known problem instances than it is to create one from scratch. For example, many testbeds have been successfully made using instances from the TSPLib [50]. Recent examples include testbeds both for variants of the Traveling Salesman Problem (TSP) like the Prize-Collecting TSP with a Budget Constraint [46] or the TSP with Time-Dependent Service Windows [62] as well as a wide variety of other problems like the Hamiltonian p-median problem [20] and the graph search problem [36]. It is also beneficial to use well-studied reductions of NP-hard problems [33] to combine test instances of various interesting problems. For example, the SATLIB benchmark library for the Satisfiability Problem contains SAT-encoded benchmark instances for the Graph Coloring Problem [31], which is also NP-hard.

18.2.2.2 Accessibility of New Test Instances

When creating a new testbed, the focus should be on providing others access to the problem instances. This will allow other researchers to more easily make comparisons, ensuring the problem instances are widely used. One way to ensure this would be to create a simple generating function for the problem instances. For example, the clustering algorithm proposed in [21] that converted TSPLib instances into clustered instances for the Generalized Traveling Salesman Problem was simple, making it easy for others to create identical problem instances. Additionally,

publishing problem instances in the paper [40] or on the Internet [19, 45] are other common ways to make problem instances accessible.

18.2.2.3 Problem Instances with Known Optimal Solutions

One problem in the analysis of metaheuristics, as discussed in more detail in Sect. 18.4, is finding optimality gaps for the procedures. Even when using advanced techniques, it is typically difficult to determine optimal solutions for large problem instances; indeed, this motivates the use of metaheuristics. A way to minimize the difficulty in this step is to construct instances where optimal or near-optimal solutions are known, often via geometric construction techniques or reduction from another optimization problem. This removes the burden on a metaheuristics designer to also implement an exact approach, relaxation results, or a tight lower bound. Instead, the designer can use the specially designed problem instances and provide a good estimate of the error of each metaheuristic tested.

A number of papers in the literature have used this approach. For instance, in [8], problem instances for the split delivery vehicle routing problem were generated with customers in concentric circles around the depot, making estimation of optimal solutions possible visually. Other examples of this approach are found in [7, 18, 37–39].

18.2.3 Problem Instance Classification

Apart from identifying instance types that are underrepresented in the testbed, problem instance classification is critical to the proper analysis of metaheuristics. It is first important to identify features or metrics that might correlate well with the algorithmic performance, and then extensively test and report performance over instances that have a wide spread across these metrics (see Sect. 18.2.2.1 for expanding the testbed). The choice of the features depends on the problem domain; for instance, for graph problems one can consider the number of nodes, density of edges, spectral analysis of the adjacency matrix [10], eigenvalues of the Laplacian, or planarity [56]. Another technique is to use predictive features in the study of phase transitions to identify hard instances for various NP-hard problems. For example, the k-colorability problem has been shown to undergo a phase-transition on regular random graphs with finite connectivity dependent on the average degree of the vertices [35]. Typically, heuristics are known to take a long time on instances that are closer to the phase transitions (thus providing a proxy for hard instances). Recently, the solutions from fast heuristics for a related NP-hard problem have been used to predict heuristic performance on a different problem [19].

A thorough comparison of performance of heuristics over a broad heterogenous testbed opens up many possibilities for further analysis. Machine learning techniques like classification and regression trees can be used to interpret heuristic per-

formance for different instance types [19]. Especially in testbeds based on real-world data, this classification of problem instances and subsequent analysis could help algorithm writers in industry with a certain type of dataset determine which method will work the best for them.

18.3 Parameters

One way to compare two heuristics is to compare their complexity; if two algorithms produce similar results but one is significantly simpler than the other, then the simpler of the two could be considered superior. Algorithms with a low degree of complexity have a number of advantages, including being simple to implement in an industrial setting, being simple to reimplement by researchers, and being simpler to explain.

A number of measures of simplicity exist. Reasonable metrics include the number of steps of pseudocode needed to describe the algorithm or the number of lines of code needed to implement the algorithm. However, these metrics are not particularly useful, since they vary based on programming language and style in the case of the lines of code metric and pseudocode level of detail in the case of the pseudocode length metric. A more meaningful metric for metaheuristic complexity is the number of parameters used in the metaheuristic, as a larger number of parameters makes it harder to analyze the method.

Parameters are the configurable components of an algorithm that can be changed to alter the performance of that algorithm. Parameters can either be set statically (for example, creating a genetic algorithm with a population size of 50) or based on the problem instance (for example, creating a genetic algorithm with a population size of $5\sqrt{n}$, where n is the number of nodes in the problem instance). In either of these cases, the constant value of the parameter or the function of problem instance attributes used to generate the parameter must be set to run the procedure.

Different classes of metaheuristics have a number of parameters that must be set before algorithm execution. Consider Table 18.1, which lists basic parameters required for major types of metaheuristics. Though these are guidelines for the minimum number of parameters typical in different types of algorithms, in practice, most metaheuristics have more parameters. For instance, a basic tabu search procedure can have just one parameter, the tabu list length. However, some procedures have many more than that one parameter. The tabu search for the vehicle routing problem presented in [69] uses 32 parameters. Likewise, algorithms can have fewer than the "minimum" number of parameters by combining parameters with the same value. For instance, the genetic algorithm for the minimum label spanning tree problem in [68] uses just one parameter, which functions to both control the population size and to serve as a termination criterion.

Table 18.1 Popular metaheuristics and their standard parameters

Name	Parameters
Ant colony optimization	Pheromone evaporation parameter Pheromone weighting parameter
Genetic algorithm	Crossover probability Mutation probability Population size
Harmony search	Distance bandwidth Memory size Pitch adjustment rate Rate of choosing from memory
Simulated annealing	Annealing rate Initial temperature
Tabu search	Tabu list length
Variable neighborhood search	Maximum neighborhood size

All metaheuristics also must include a termination criterion

18.3.1 Parameter Space Visualization and Tuning

The effort needed to tune or understand a metaheuristic's parameters increases as the number of parameters increases. A brute-force technique for parameter tuning involves testing m parameter values for each of the n parameters, a procedure that should test m^n configurations over a subset of the problem instances. Assuming we choose to test just 3 values for each parameter, we must test 9 configurations for an algorithm with 2 parameters and 2187 values for an algorithm with 7 parameters. While this number of configurations is likely quite reasonable, the number needed for a 32-parameter algorithm, 1,853,020,188,851,841, is clearly not reasonable. The size of the parameter space for an algorithm with a large number of parameters expands in an exponential manner, making the search for a good set of parameters much more difficult as the number of parameters increases. While, of course, there are far better ways to search for good parameter combinations than brute-force search, such as automatic parameter tuning packages like `irace` [41], the size of the search space still increases exponentially with the number of parameters, meaning a large number of parameters makes this search much more difficult.

A large number of parameters also makes the parameter space much harder to visualize or understand. As a motivating example, consider the relative ease with which the parameter space of an algorithm with two parameters can be analyzed. For example, we applied the two-parameter metaheuristic in [54] for solving the Generalized Orienteering Problem on a few random problems from the TSPLib-based large-scale Orienteering Problem dataset considered in that paper. To analyze this

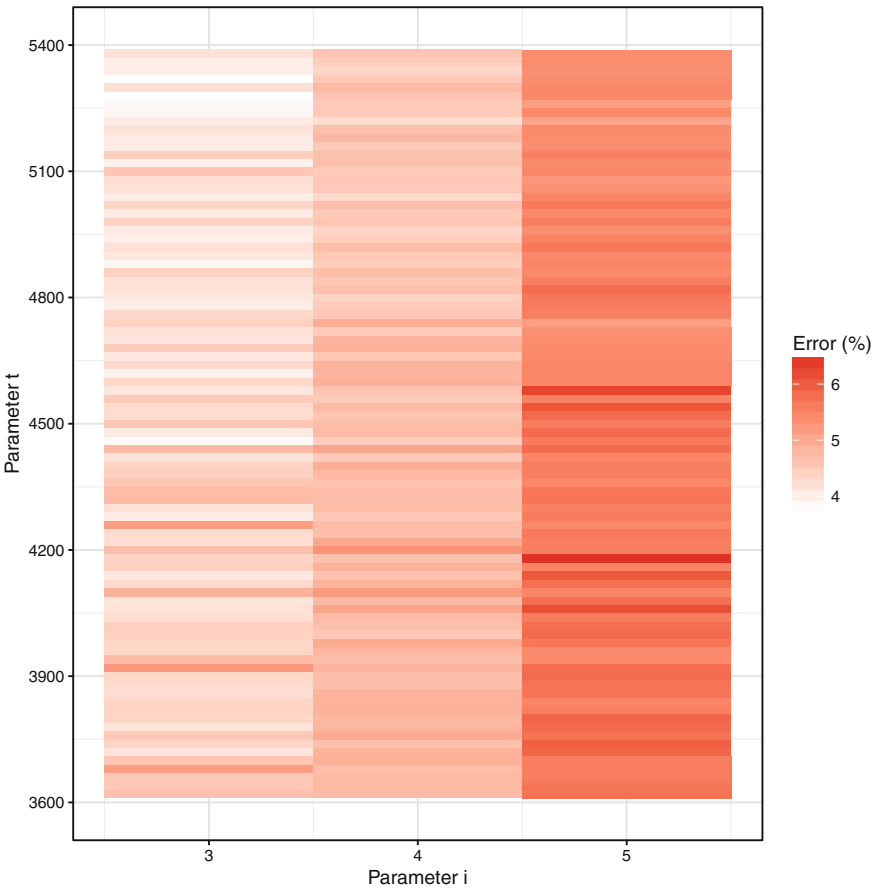

Fig. 18.1 Depiction of solution quality of a metaheuristic for the Generalized Orienteering Problem over its two-dimensional parameter space. The *x*-axis is the parameter *i* at three separate values and the *y*-axis is the parameter *t* over a large range of values. The colors in the figure represent the optimality gap of the metaheuristic at the indicated parameter setting

algorithm, we chose a number of parameter configurations in which each parameter value was close to the parameter values used in that paper. For each parameter set, the algorithm was run 20 times on each of five randomly selected problem instances with known optima from all the TSPLib-based instances used.

The resulting image, shown in Fig. 18.1, is a testament to the simplicity of analysis of an algorithm with just two parameters. In this figure, different values of the parameter *i* are shown on the *x*-axis, while different values of the parameter *t* are shown on the *y*-axis. Parameter *i* is an integral parameter with small values, so results are plotted in three columns representing the three values tested for that parameter: 3, 4, and 5. For each parameter set (a pair of *i* and *t*), a rectangle is plotted with a color indicating the average optimality gap of the algorithm over the 20 runs

for each of the five problem instances tested. It is immediately clear that the two lower values tested for i, 3 and 4, are superior to the higher value of 5 on the problem instances tested. Further, it appears that higher values of t are preferred over lower ones for all of the values of i tested. This sort of simplistic visual analysis becomes more difficult as the dimensionality of the parameter space increases.

18.3.2 Parameter Interactions

Parameter space visualization and tuning are not the only downside of metaheuristics with a large number of parameters. It is also difficult to analyze parameter interactions in metaheuristics with a large set of parameters. These complex interactions might lead to, for instance, multiple locally optimal solutions in the parameter space in terms of solution quality. In a more practical optimization sense, this concept of parameter interaction implies that optimizing parameters individually or in small groups will become increasingly ineffective as the total number of parameters increases.

Parameter interaction is a topic that has been documented in a variety of works. For instance, in [16] the authors observe non-trivial parameter interactions in genetic algorithms with just three parameters. These authors note that the effectiveness of a given parameter mix is often highly based on the set of problem instances considered and the function being optimized, further noting the interdependent nature of the parameters. To a certain extent, it is often very difficult to avoid parameter interactions such as these. In the case of genetic algorithms, for instance, a population size parameter, crossover probability parameter, and mutation probability parameter are typically used, meaning these algorithms will often have at least the three parameters considered in [16]. However, there have been genetic algorithms developed that operate using only one parameter [68] or none [52, 53], eliminating the possibility of parameter interactions.

Given three or more parameters, an effective and efficient design of experiments method is the Plackett-Burman method [48], which tests a number of configurations that is linear in the number of parameters considered. Though this method is limited in that it can only show second-order parameter interactions (the interactions between pairs of parameters), this is not an enormous concern as most parameter interactions are of the second-order variety [43].

18.3.3 Fair Testing Involving Parameters

Though the effect of parameters on algorithmic simplicity is an important consideration, it is not the only area of interest in parameters while comparing metaheuristics. The other major concern is one of fairness in parameter tuning—if one algorithm is tuned very carefully to the particular set of problem instances on which it is tested,

this can make comparisons on those instances unfair. Instead of tuning parameters on all the problem instances used for testing, a fairer methodology for parameter setting involves choosing a representative subset of the problem instances to train parameters on, to avoid overtraining the data. A full description of one such methodology can be found in [14]. To ensure reproducibility of results, the resultant parameters, which are used to solve the test instances, must also be published along with the running time and the quality of solutions obtained.

18.4 Solution Quality Comparisons

While it is important to gather a meaningful testbed and to compare the metaheuristics in terms of simplicity by considering their number of parameters, one of the most important comparisons involves solution quality. Metaheuristics are designed to give solutions of good quality in runtimes better than those of exact approaches. To be meaningful, a metaheuristic must give acceptable solutions, for some definition of acceptable.

Depending on the application, the amount of permissible deviation from the optimal solution varies. For instance, in many long-term planning applications or applications critical to a company's business plan, the amount of permissible error is much lower than in optimization problems used for short-term planning or for which the solution is tangential to a company's business plan. Even for the same problem, the amount of permissible error can differ dramatically. For instance, a parcel company planning its daily routes to be used for the next year using the capacitated vehicle routing problem would likely have much less error tolerance than a planning committee using the capacitated vehicle routing problem to plan the distribution of voting materials in the week leading up to Election Day.

As a result, determining a target solution quality for a combinatorial optimization problem is often difficult or impossible. Therefore, when comparing metaheuristics it is not sufficient to determine if each heuristic meets a required solution quality threshold; comparison among the heuristics is necessary.

18.4.1 Solution Quality Metrics

To compare two algorithms in terms of solution quality, a metric to represent the solution quality is needed. In this discussion of the potential metrics to be selected, we assume that solution quality comparisons are being made over the same problem instances. Comparisons over different instances are generally weaker, as the instances being compared often have different structures and almost certainly have different optimal values and difficulties.

Of course, the best metric to use in solution quality comparison is the deviation of the solutions returned by the algorithms from optimality. Finding the average percentage error over all problems is common practice, because this strategy gives

equal weight to each problem instance (instead of, for instance, giving preference to problem instances with larger optimal solution values).

However, using this metric requires knowledge of the optimal solution for every problem instance tested. This is an assumption that likely cannot always be made except in the case of instances constructed with known optima, as described in Sect. 18.2.2.3. If exact algorithms can compute optimal solutions for every problem instance tested in reasonable runtimes, then the problem instances being considered are likely not large enough.

This introduces the need for new metrics that can provide meaningful information without access to the optimal solution for all (or potentially any) problem instances. Two popular metrics that fit this description are deviation from best-known solutions for a problem and deviation between the algorithms being compared.

Deviation from best-known solution or tightest lower bound can be used on problems for which an optimal solution was sought using an exact approach, but optimal solutions were not obtained for some problem instances within a predetermined time limit. In these cases, deviation from best-known solution or tightest relaxation is meaningful because for most problem instances the best-known solution or tightest relaxation will be a near-optimal solution. An example of the successful application of this approach can be found in [22]. In this paper, the authors implement three approaches for solving the multilevel capacitated minimum spanning tree problem. One of these approaches is a metaheuristic, another uses mixed integer programming, and the third is a linear programming relaxation. Though the optimal solution was not provably computed for the largest problem instances due to the excessive runtime required, the low average deviation of the metaheuristic from the optimal solution on smaller problem instances (0.3%) and the moderate average deviations from the relaxed solutions over all problem instances (6.1%) conveyed a notion of the solution quality achieved by the metaheuristic.

The deviation from best-known solution could be used for problems for which no optimal solution has been published, though the resulting deviations are less meaningful. It is unclear in this case how well the metaheuristic performs without an understanding of how close the best-known solutions are to optimal solutions. One way to construct such a bound is to consider a relaxation P_R of the original problem P. Typically P_R is much easier to solve than P, and the optimal solution of P_R provides a lower (upper) bound to the minimization (maximization) problem P. The gap from optimality of any solution to P can then be bounded using the gap from the optimal solutions to the relaxed problem P_R. We refer the reader to [64] for an introduction to such techniques.

Though the metric of deviation between the metaheuristics being compared also addresses the issue of not having access to optimal solutions, it operates differently—any evaluation of solution quality is done in relation to the other metaheuristic(s) being considered. This method has the advantage of making the comparison between the metaheuristics very explicit—all evaluations, in fact, compare the two or more procedures. However, these comparisons lack any sense of the actual error of solutions. Regardless of how a metaheuristic fares against another metaheuristic, its actual error as compared to the optimal solution is unavailable using this metric. Therefore, using a metric of deviation from another algorithm loses

much of its meaningfulness unless accompanied by additional information, such as optimal solutions for some of the problem instances, relaxation results for the problem instances, or deviation from tight lower bounds (to give a sense of the global optimality of the algorithms).

When comparing two stochastic metaheuristics, whenever possible one should generate a number of replicates (say 10) for each instance. For each metaheuristic, one should record the average, worst, and best solutions, as well as the standard deviation. Furthermore, one should try to compare the distribution of solutions obtained from each metaheuristic. Statistical tests might be applied to compare the average or minimum solution values. See [8] for an interesting comparison based on the binomial distribution and [49] for more on statistical analysis.

18.4.2 Comparative Performance on Different Types of Problem Instances

When comparing the performance of a portfolio of heuristics, it is often useful to identify instance types where one heuristic outperforms all the others. As noted earlier, such a comparison does not require the knowledge of the optimal solutions for hard problems and the comparison can be made with respect to the best solution attained by any heuristic in the portfolio. A comparative analysis highlighting the weaknesses and strengths of heuristics in the instance space is known as an *algorithmic footprint* [13]. Instances are represented as points in a high dimensional feature space, and these points can be colored on a continuous scale (e.g., a gradation from red to blue where red depicts worst performance and blue depicts best performance) in order to reveal heuristic performance. It helps to visualize the instance space on a 2-dimensional plane, by performing a principal component analysis [58] or self-organizing maps [57]. Important insights into the effectiveness of various algorithmic techniques can be gained by analyzing the footprints of classes of heuristics, for instance, evolutionary, tabu search, simulated annealing approaches, etc.

Interpretable machine learning models like regression trees can also be used to identify problem instances where a given heuristic performs better or worse, using instance features or metrics that are the most predictive of performance (see Sect. 18.2.3 for details). Such heuristic-specific models make it harder to directly compare the footprints of different heuristics, since the most significant features used in each model might be different across heuristics. However, the results of such an analysis remain interpretable without losing much information due to dimension reduction [19].

18.5 Runtime Comparisons

One can find examples in the metaheuristics literature where a metaheuristic A outperforms another metaheuristic B in terms of solution quality using the metrics described in Sect. 18.4 but was also run for a substantially longer time before termina-

tion. This makes it challenging to interpret the comparison of A and B because most metaheuristics will continue making progress toward optimality if they are allowed to run for longer; the reader cannot determine if the solution quality difference is due to the additional computational resources given to A or due to superior algorithmic performance.

To address this concern, researchers must allocate the same amount of computational resources when comparing heuristics. One way to do this is to limit the heuristics to the same fixed runtime, an approach that we describe in Sects. 18.5.1 and 18.5.2. Runtime growth rate is discussed in Sect. 18.5.3. Alternatives to runtime-based limits are described in Sect. 18.5.4.

18.5.1 Runtime Limits Using the Same Hardware

When making a comparison between metaheuristics A and B using a fixed runtime limit for each problem instance, the best approach is to get the source code for each algorithm, compile them with the same compilation flags, and run both algorithms on the same computer. Since the hardware and software environments are the same for both metaheuristics, one can argue that the runtime limit gives each the same computational resources, enabling us to focus on solution quality differences when comparing A and B. However, this technique for imposing runtime limits is often not possible.

One case in which it is not possible is if the algorithms were programmed in different languages. This implies that their runtimes are not necessarily directly comparable. Though attempts have been made to benchmark programming languages in terms of speed (see, for instance, [6]), these benchmarks are susceptible to the types of programs being run, again rendering any precise comparison difficult. Further invariants in these comparisons include compiler optimizations. The popular C compiler `gcc` has over 100 optimization flags that can be set to fine-tune the performance of a C program. While the technique of obtaining a scalar multiplier between programming languages will allow comparisons to be more accurate within an order of magnitude between algorithms coded in different programming languages, these methods cannot provide precise comparisons.

It is sometimes not possible to obtain the source code for the algorithm being compared to. The source code may have been lost (especially in the case of older projects) or the authors may be unwilling to share their source code. Another way to proceed when comparing heuristics with a runtime limit is to reimplement the other code in the same language as your code and run it on the same computer on the same problem instances. However, this approach suffers from two major weaknesses. First, the exposition of some algorithms is not clear on certain details of the approach, making an exact reimplementation difficult. Second, there is no guarantee that the approach used to reimplement another researcher's code is really similar to their original code. For instance, the other researcher may have used a clever data structure or algorithm to optimize a critical part of the code, yielding better runtime efficiency. As there is little incentive for a researcher to perform the hard work of

optimizing the code to compare against, but much incentive to optimize one's own code, at times reimplementations tend to overstate the runtime performance of a new algorithm over an existing one (see [5] for a humorous view of issues such as these). One way to address these concerns is to make the implementations of previously published heuristics open source, so that an active research community can optimize implementations as required.

18.5.2 Runtime Limits Using Different Hardware

As indicated previously, it can be challenging to compare two heuristics using a runtime-based termination criterion without access to source code or reimplementation. One approach is to compare the performance of a metaheuristic A to the published results of another metaheuristic B on the publicly available problem instances reported in B's publication. While the instances being tested are the same and the algorithms being compared are the algorithms as implemented by their developers, the computer used to test these instances is different, and the compiler and compiler flags used are likely also not the same. This approach has the advantage of ease and simplicity for the researcher—no reimplementation of other algorithms is needed. Further, the implementations of each algorithm are the implementations of their authors, meaning there are no questions about implementation as there were in the reimplementation approach.

However, the problem then remains to provide a meaningful comparison between the two runtimes. Researchers typically solve this issue by using computer runtime comparison tables such as the one found in [17] to derive approximate runtime multipliers between the two computers. These comparison tables are built by running a benchmarking algorithm (in the case of [17], this algorithm is a system of linear equations solved using LINPACK) and comparing the time to completion for the algorithm. However, it is well known that these sorts of comparisons are imprecise and highly dependent on the program being benchmarked, and the very first paragraph of the benchmarking paper mentions the limitations of this sort of benchmarking: "The timing information presented here should in no way be used to judge the overall performance of a computer system. The results only reflect one problem area: solving dense systems of equations." In fact, [30] argues that new and more relevant benchmark codes in the field of combinatorial optimization (perhaps based on metaheuristics for the Traveling Salesman Problem) would be quite useful. Beyond limitations of the code being benchmarked, these scaling factors do not take into account RAM, operating system, compiler and its optimization level, and other factors known to impact the runtime of metaheuristics. Hence, the multipliers gathered in this way can only provide a rough idea of runtime performance, clearly a downside of the approach. In situations where the systems used for testing seem roughly comparable, there may be no benefit to performing runtime scaling in this way, and indeed the scaling may only introduce noise to the comparison.

18.5.3 Runtime Growth Rate

Regardless of the comparison method used to compare algorithms' runtimes, the runtime growth rate can be used as a universal language for the comparison of runtime behaviors of two algorithms. While upper bounds on runtime growth play an important role in the discussion of heuristic runtimes, metaheuristic analysis often does not benefit from these sorts of metrics. Consider, for instance, a genetic algorithm that terminates after a fixed number of iterations without improvement in the solution quality of the best solution to date. No meaningful worst-case analysis can be performed, as there could be many intermediate best solutions encountered during the metaheuristic's execution. (For example, the nearest neighbor heuristic for the TSP is a simple heuristic with an unambiguous stopping point. It has a running time that grows with n^2 in the worst case, where n is the number of nodes. For metaheuristics such as tabu search, simulated annealing, genetic algorithms, etc., there is no unambiguous stopping point.)

An alternative approach to asymptotic runtime analysis for metaheuristics is fitting a curve to the runtimes measured for each of the algorithms across a range of problem instance sizes. These results help indicate how an algorithm might perform as the problem size increases. Though there is no guarantee that trends will continue past the endpoint of the sampling (motivating testing on large problem instances) or on problem instances with different structural properties than the ones used for the analysis, runtime trends are key to runtime analyses. Even if one algorithm runs slower than another on small- or medium-sized problem instances, a favorable runtime trend suggests the algorithm may well perform better on large-sized problem instances, where metaheuristics are most helpful. Curve-fitting for runtime analysis has been recommended or used in a number of metaheuristics articles [9, 12, 66].

18.5.4 Alternatives to Runtime Limits

The focus thus far has been on using runtime limits to control the computational resources allocated to each metaheuristic. This sounds like a fair comparison, but, as [30] points out, the results are not reproducible. Another researcher, using a slightly different computing environment, might obtain distinctly different results. Instead of controlling computational resources with runtime limits, codes might be designed to count easy-to-measure basic combinatorial operations, such as the number of neighborhoods searched or the number of branching steps. Then, solution quality and running time can be reported after a stopping rule of at most k basic combinatorial operations, as recommended in [1, 30]. A number of studies compare heuristic runtimes using representative operation counts or give all heuristics the same budget of these operations [2, 3, 47].

Beyond improved reproducibility, there are several clear advantages to this approach over runtime comparisons. As described in [1], it removes the invariants of compiler choice, programmer skill, and power of computation platform. However, this approach suffers from the fact that it is often difficult to identify good oper-

ations that each algorithm being compared will implement. Also, some operations may take longer than others. The only function sure to be implemented by every procedure is the evaluation of the function being optimized. As a result, comparisons of this type often only compare on the optimization function, losing information about other operations, which could potentially be more expensive or more frequently used. As a result, in the context of metaheuristic comparison, this technique is best if used along with more traditional runtime comparisons.

A related approach is to predetermine a percentage or several percentages above a well-established tight lower bound (e.g., the Held and Karp TSP lower bound) and compare metaheuristics based on how long each one takes to reach these targets (see [26, 27] for details). A maximum runtime is typically specified. Of course, these tight lower bounds are often difficult to obtain. We point out that this notion of setting a level of solution quality and comparing runtimes is used in the definition of speedup for parallel algorithms (e.g., see Fig. 18.2 in Sect. 18.6.1). It has also been used with MOMHs [29].

18.6 Parallel Algorithms

Until 15 or so years ago, the use of parallel computers was largely restricted to computer scientists at major research universities or national laboratories; they were the only ones with access to these resources. More recently, parallel computing has become a very practical tool in the computational sciences (and in industry). In this section, we devote our attention to the use of parallel computing in metaheuristic optimization and, in particular, to the theme of assessing the relative performance of metaheuristic algorithms. For example, suppose we have a serial (or sequential) metaheuristic (called A), a parallel metaheuristic (called B) designed to run on m processors, and another parallel metaheuristic (called C) designed to run on n processors. How should we compare the performance of these three metaheuristics? What are the right metrics to look at and report? Accuracy, runtime, and cost of computation are measures that come to mind, but some of these issues are more subtle than they may seem at first glance.

Furthermore, there are at least two key scenarios to consider. In the first, we have access to the three codes (A, B, and C) and we can fully control the computational experiments. That is, we can select the benchmark instances and the experimental environment (computer, network protocol, operating system, etc.). In the second, we have access to the literature but *not* the codes themselves. The three codes were run on three different machines and in different experimental environments at different times. How should we perform a computational comparison that is fair, revealing, and informative?

This topic will be the focus of this section. Although we will offer some detailed suggestions, we point out that our recommendations are tentative. This is a topic of discussion that has not been widely covered in the literature. Two recent exceptions deal specifically with parallel genetic algorithms [4, 42].

18.6.1 Evaluating Parallel Metaheuristics

Although parallel algorithms have been proposed and analyzed with respect to genetic algorithms, the operations research community has been slow to take advantage of this readily available technology. This is somewhat surprising, since most modern desktop computers already have CPUs with at least four cores, and the number of cores will surely increase over the next few years. In addition, new research from M.I.T. by [59] seeks to make it easier to write parallel computer programs.

The motivation behind parallel computing is to reduce the elapsed or wall-clock time needed to solve a particular problem or problem instance. In other words, we want to solve larger problems in minutes or hours, rather than weeks or months. For some applications, parallel computing may be the only way to solve a problem. For example, nearly 400 computers were used to create the 2017 NFL schedule [34]; the elapsed time to solve the problem on a single machine would have been prohibitively long.

Given the importance of elapsed time, speedup is a key metric in evaluating parallel algorithms. A standard speedup metric is given by

$$S_n = \frac{\mathbb{E}(T_1)}{\mathbb{E}(T_n)}$$

where $\mathbb{E}(T_n)$ is the mean parallel execution time of a given task using n processors and $\mathbb{E}(T_1)$ is the mean serial execution time of the same task. Numerous other measures of speedup are discussed in Chapter 2 of [42] and in [60].

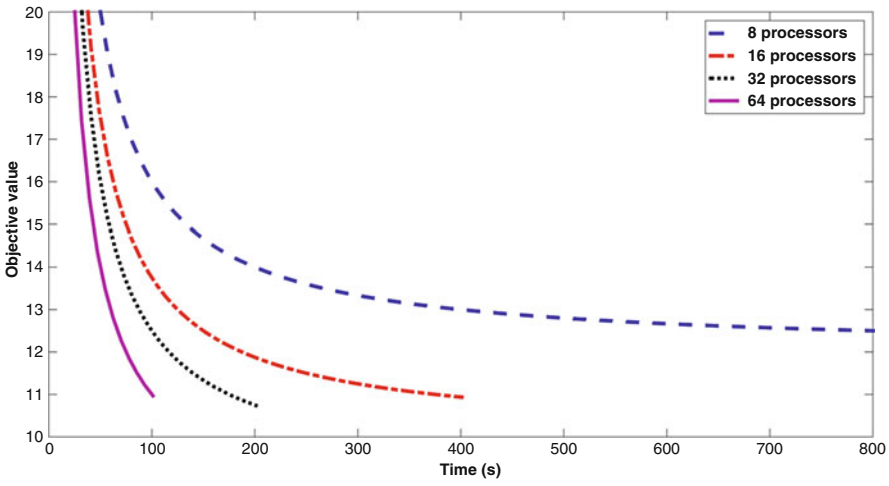

Fig. 18.2 Average solution trajectories

We will illustrate the notion of speedup in Fig. 18.2 where four trajectories are shown for an algorithm running with four different numbers of processors (8, 16, 32, and 64). Each trajectory (solution value improvement over time) represents an

average over 20 runs of a stochastic metaheuristic. This figure is an adaptation of several figures in [24]. For example, we observe that the trajectories for 64 processors and 16 processors can obtain the same low objective value of about 11.0. The latter requires 400 s vs. 100 s for the former. This reduction in elapsed or wall-clock time is the key motivation behind parallel computing.

18.6.2 Comparison When Competing Approaches Can Be Run

We first consider the computational comparison of a parallel metaheuristic against other metaheuristics (either serial or parallel) when we are able to run all the relevant procedures. As in Sect. 18.5, this is the ideal scenario for computational comparison, because we can test all the procedures on a wide range of test instances and using the same computing environment. We can see how well each code performs with respect to quality of solution as we increase a predetermined limit on elapsed runtime. In addition, we are better able to specify most of the key characteristics of each metaheuristic. However, in practice, such a comparison might not be possible due to unavailability of the source code for competing approaches or due to difficulties in getting that code to work in one's own computing environment.

In comparing two parallel metaheuristics, ideally each metaheuristic would be run with the same number of processors and the same limit on elapsed runtime. Then, solution qualities could be directly compared. However, parallel algorithms are often designed for a specific computer environment with a pre-determined number of processors, so it may not be possible to run them with the exact same number of processors. In some parallel algorithms, all the processors essentially perform the same task. In more heterogeneous parallel algorithms, different processors play different roles. For example, in [24] some processors perform local search, while others solve set-covering problems. In addition, there is a master processor responsible for controlling the flow and timing of communication. It also coordinates the search for the best set of vehicle routes and tries to ensure that bottlenecks are avoided or minimized. The relative numbers of the three types of processors are determined in the algorithm design process; the goal is to maximize the performance of the parallel algorithm. An algorithm designed to run on 129 processors may not make sense on 29 processors. Even if it does run, it is likely to be a substantially different algorithm. In such scenarios, it may only be possible to run the two parallel metaheuristics with a similar number of processors.

Now suppose we want to compare a serial (stochastic) metaheuristic and a parallel metaheuristic that runs on n processors. While we could simply run each procedure for the same elapsed time, this provides an unfair advantage to the parallel metaheuristic, which uses more processors. To perform a more evenhanded comparison, we could instead build a simple parallel algorithm on n processors for the serial metaheuristic, simply running the procedure with a different random seed on each of the n processors and then taking the best of the n solutions produced. Furthermore, we could give each of the two (now) parallel algorithms t units of elapsed time

and compare the resulting solution values. This approach can be implemented using software such as the SNOW package in R (see [63] for details). Equivalently (from a conceptual point of view), we could run the serial code n times in succession, ideally allowing t units of time per run. Next, we would record the best of the n solution values. Of course, if the serial metaheuristic is deterministic, this will not work.

18.6.3 Comparison When Competing Approaches Cannot Be Run

Given that source code is often unavailable for heuristics published in the literature, it is often the case that a new parallel metaheuristic must be compared against procedures that cannot be further tested. In this case, the comparison must be based on published information about the competing metaheuristics.

First, consider the comparison of a new parallel metaheuristic (A) against a parallel metaheuristic (B) published in the literature. Assume B was tested using n processors and that information was published about average elapsed runtimes and solution qualities on a testbed of problem instances. Following our approach from Sect. 18.6.2, we would ideally like to test A on the same instances for the same elapsed runtime using n processors and the same hardware configuration. However, it is very unlikely that we have access to the same hardware that was used in the published study. Instead, we might scale the elapsed runtimes of the procedures based on the hardware used, as described in Sect. 18.5.2. Such scaling should be done with a good deal of caution—in addition to the limitations described in Sect. 18.5.2, the scaling also does not control for details of the communication network connecting the processors, which can also have a significant impact on the performance of a parallel algorithm. As discussed in Sect. 18.6.2, additional complications may arise if metaheuristic A cannot be run on exactly n processors; in such cases, it may only be possible to test on approximately the same number of processors.

Next, consider the comparison of a new stochastic serial metaheuristic (A) against a parallel metaheuristic (B) published in the literature. Again, assume that B was tested using n processors and that information was published about average elapsed runtimes and solution qualities on a testbed of problem instances. Following our approach from Sect. 18.6.2, we "parallelize" A by running it with different random seeds on n different processors and returning the best result from the n independent runs. Ideally, we would perform our comparison by running each of the n copies of A for the elapsed time that was reported by B, using the same hardware. As before, the same hardware is likely unavailable, so some form of elapsed time scaling might be warranted.

We think this approach is more equitable than comparing the solution of B to the solution obtained by running A once with elapsed time equal to the total CPU time (elapsed runtime summed across all processors) of B. Under this alternative approach to comparing metaheuristics, both metaheuristics have the same total CPU time but A is given more elapsed time than B. Having a larger elapsed time than B might be especially beneficial to metaheuristic A if it is an evolutionary procedure

that slowly evolves toward high-quality solution spaces, as such procedures might not find good solutions if run many times for a short runtime, but might find better solutions if run once for an extended period of time. Meanwhile, we would expect little difference between the two evaluation approaches if metaheuristic A is a procedure that makes frequent random restarts, e.g., an iterated local search metaheuristic with random restarts.

It is also unfair, by extension, to ignore elapsed runtime and only consider total CPU time when comparing serial and parallel metaheuristics published in the literature, a comparison approach that has been used previously (see, e.g., [65]). After all, the objective of parallelization is to minimize elapsed runtime. Therefore, parallel metaheuristics should be judged, in large part, by the elapsed runtimes associated with them. It is perfectly reasonable, however, to *parallelize* a stochastic serial metaheuristic A in order to compare it to a parallel metaheuristic B given a predetermined elapsed runtime of t units.

Finally, consider the comparison of a new parallel metaheuristic (A) against a stochastic serial metaheuristic (B) published in the literature. Assume that B was tested with n replicates per problem instance to test its variability to seed, and assume that for each problem instance the publication provides the maximum and/or average elapsed runtime and best solution quality across the n replicates. Following our approach from Sect. 18.6.2, the n experiments to test variability to seed represent a "parallelized" version of B, so we would ideally perform a comparison between A and B by running A on n processors of the same hardware, with elapsed time equal to the maximum runtime of the n replicates of B (or the average elapsed time, if the maximum is not reported). As before, the same hardware is likely unavailable, so some form of elapsed time scaling might be warranted. Naturally, this approach can only be used if variability to seed is assessed for metaheuristic B.

Though in this section we have provided some guidance on how to compare parallel metaheuristics with other metaheuristics, it should be clear that there are additional challenges that were not present when all approaches being compared were serial. We leave several unanswered questions, such as how to compare parallel metaheuristics when they cannot be run on the same number of processors, and how to compare a new parallel metaheuristic to a serial metaheuristic from the literature when the serial heuristic's code is not available and no variability to seed information is reported. Clearly, much more work remains on the topic of computational comparisons with parallel metaheuristics.

In light of the challenges in comparisons involving parallel metaheuristics, we conclude with the recommendation to report as many details as possible about the comparison being performed, to give readers as much context as possible. Details that could be helpful to the reader include: computing environment (including details of the network linking parallel processors), programming language used and compiler flags, number of processors, final solution quality, elapsed runtime, test datasets (real-world, standard, random), number of parameters (fewer is preferred), whether stochastic or deterministic, if stochastic then the number of repetitions used in testing, speed of convergence to an attractive solution, speed of convergence to the final solution, stopping rules (based on time or solution quality), and lastly, for a parallel algorithm a notion of speedup.

18.7 Conclusion

We believe following the procedures described in this chapter will increase the quality of metaheuristic comparisons. In particular, choosing an appropriate testbed and distributing it so other researchers can access it will result in more high-quality comparisons of metaheuristics, as researchers will test on the same problem instances. Further, expanding the practice of creating geometric problem instances with easy-to-visualize optimal or near-optimal solutions will increase understanding of the optimality gap of metaheuristic solutions.

Furthermore, it is important to recognize that the number of algorithm parameters has a direct effect on the complexity of the algorithm and on the number of parameter interactions, which complicates analysis. If the number of parameters is considered in the analysis of metaheuristics, this will encourage simpler, easier-to-analyze procedures.

Finally, good techniques in solution quality and runtime comparisons will ensure fair and meaningful comparisons are carried out between metaheuristics, producing the most meaningful and unbiased results possible. Since parallel metaheuristics have become much more widespread in the recent research literature than before, it is important to establish fair and straightforward guidelines for comparing parallel and serial metaheuristics with respect to computational effort. In this chapter, we have taken a number of steps toward reaching this goal.

References

1. R. Ahuja, J. Orlin, Use of representative operation counts in computational testing of algorithms.INFORMS J. Comput. **8**(3), 318–330 (1996)
2. R.K. Ahuja, M. Kodialam, A.K. Mishra, J.B. Orlin, Computational investigations of maximum flow algorithms. Eur. J. Oper. Res. **97**(3), 509–542 (1997)
3. T. Akhtar, C.A. Shoemaker, Multi objective optimization of computationally expensive multimodal functions with RBF surrogates and multi-rule selection. J. Glob. Optim. **64**(1), 17–32 (2016)
4. E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends. Int. Trans. Oper. Res. **20**(1), 1–48 (2013)
5. D. Bailey, Twelve ways to fool the masses when giving performance results on parallel computers. Supercomput. Rev. **4**(8), 54–55 (1991)
6. M. Bull, L. Smith, L. Pottage, R. Freeman, Benchmarking Java against C and Fortran for scientific applications, in *ACM 2001 Java Grande/ISCOPE Conference* (2001), pp. 97–105
7. I.-M. Chao, *Algorithms and solutions to multi-level vehicle routing problems*. PhD thesis, University of Maryland, College Park, MD, 1993
8. S. Chen, B. Golden, E. Wasil, The split delivery vehicle routing problem: applications, algorithms, test problems, and computational results. Networks **49**(4), 318–329 (2007)
9. P. Chen, B. Golden, X. Wang, E. Wasil, A novel approach to solve the split delivery vehicle routing problem. Int. Trans. Oper. Res. **24**(1–2), 27–41 (2017)
10. F.R.K. Chung, *Spectral Graph Theory*, vol. 92 (American Mathematical Society, Providence, 1997)
11. C. Coello, Evolutionary multi-objective optimization: a historical view of the field. IEEE Comput. Intell. Mag. **1**(1), 28–36 (2006)

12. M. Coffin, M.J. Saltzman, Statistical analysis of computational tests of algorithms and heuristics. INFORMS J. Comput. **12**(1), 24–44 (2000)
13. D.W. Corne, A.P. Reynolds, Optimisation and generalisation: footprints in instance space, in *International Conference on Parallel Problem Solving from Nature* (Springer, Berlin, 2010), pp. 22–31
14. S. Coy, B. Golden, G. Runger, E. Wasil, Using experimental design to find effective parameter settings for heuristics. J. Heuristics **7**(1), 77–97 (2001)
15. J. Culberson, A. Beacham, D. Papp, Hiding our colors, in *CP95 Workshop on Studying and Solving Really Hard Problems* (1995), pp. 31–42
16. K. Deb, S. Agarwal, Understanding interactions among genetic algorithm parameters, in *Foundations of Genetic Algorithms* (Morgan Kauffman, San Mateo, 1998), pp. 265–286
17. J. Dongarra, Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee, 2014
18. M.M. Drugan, Generating QAP instances with known optimum solution and additively decomposable cost function. J. Comb. Optim. **30**(4), 1138–1172 (2015)
19. I. Dunning, S. Gupta, J. Silberholz, What works best when? A systematic evaluation of heuristics for Max-Cut and QUBO. INFORMS J. Comput. (2018, to appear)
20. G. Erdoğan, G. Laporte, A.M. Rodríguez Chía, Exact and heuristic algorithms for the Hamiltonian *p*-median problem. Eur. J. Oper. Res. **253**(1), 280–289 (2016)
21. M. Fischetti, J.J. Salazar González, P. Toth, A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. Oper. Res. **45**(3), 378–394 (1997)
22. I. Gamvros, B. Golden, S. Raghavan, The multilevel capacitated minimum spanning tree problem. INFORMS J. Comput. **18**(3), 348–365 (2006)
23. M. Gendreau, G. Laporte, F. Semet, A tabu search heuristic for the undirected selective travelling salesman problem. Eur. J. Oper. Res. **106**(2–3), 539–545 (1998)
24. C. Groër, B. Golden, E. Wasil, A parallel algorithm for the vehicle routing problem. INFORMS J. Comput. **23**(2), 315–330 (2011)
25. A.A. Hagberg, D.A. Schult, P.J. Swart, Exploring network structure, dynamics, and function using NetworkX, in *Proceedings of the 7th Python in Science Conference*, Pasadena, 2008, pp. 11–15
26. M. Held, R.M. Karp, The traveling-salesman problem and minimum spanning trees. Oper. Res. **18**(6), 1138–1162 (1970)
27. M. Held, R.M. Karp, The traveling-salesman problem and minimum spanning trees: Part II. Math. Program. **1**(1), 6–25 (1971)
28. R. Jans, Z. Degraeve, Meta-heuristics for dynamic lot sizing: a review and comparison of solution approaches. Eur. J. Oper. Res. **177**(3), 1855–1875 (2007)
29. A. Jaszkiewicz, Do multi-objective metaheuristics deliver on their promises? A computational experiment on the set-covering problem. IEEE Trans. Evol. Comput. **7**(2), 133–143 (2003)
30. D.S. Johnson, A theoretician's guide to experimental analysis of algorithms, in *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, Providence, 2002, ed. by M.H. Goldwasser, D.S. Johnson, C.C. McGeoch, pp. 215–250
31. D.S. Johnson, C.R. Aragon, L.A. McGeoch, C. Schevon, Optimization by simulated annealing: an experimental evaluation: part II, graph coloring and number partitioning. Oper. Res. **37**(6), 865–892 (1989)
32. D.F. Jones, S.K. Mirrazavi, M. Tamiz, Multi-objective meta-heuristics: an overview of the current state-of-the-art. Eur. J. Oper. Res. **137**(1), 1–9 (2002)
33. R.M. Karp, Reducibility among combinatorial problems, in *Complexity of Computer Computations* (Springer, Berlin, 1972), pp. 85–103
34. P. King, How the NFL schedule was made, 2017. Retrieved from https://www.si.com/mmqb/2017/04/21/nfl-2017-schedule-howard-katz-roger-goodell
35. F. Krzkakała, A. Pagnani, M. Weigt, Threshold values, stability analysis, and high-q asymptotics for the coloring problem on random graphs. Phys. Rev. E **70**(4), 046705 (2004)
36. M. Kulich, J.J. Miranda-Bront, L. Přeučil, A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment. Comput. Oper. Res. **84**, 178–187 (2017)

37. F. Li, B. Golden, E. Wasil, Very large-scale vehicle routing: new test problems, algorithms, and results. Comput. Oper. Res. **32**(5), 1165–1179 (2005)
38. F. Li, B. Golden, E. Wasil, The open vehicle routing problem: algorithms, large-scale test problems, and computational results. Comput. Oper. Res. **34**(10), 2918–2930 (2007)
39. F. Li, B. Golden, E. Wasil, A record-to-record travel algorithm for solving the heterogeneous fleet vehicle routing problem. Comput. Oper. Res. **34**(9), 2734–2742 (2007)
40. J. Liu, D. Wang, K. He, Y. Xue, Combining Wang-Landau sampling algorithm and heuristics for solving the unequal-area dynamic facility layout problem. Eur. J. Oper. Res. **262**(3), 1052–1063 (2017)
41. M. López-Ibánez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration. Oper. Res. Perspect. **3**, 43–58 (2016)
42. G. Luque, E. Alba, *Parallel Genetic Algorithms: Theory and Real World Applications* (Springer, Berlin, 2011)
43. D. Montgomery, *Design and Analysis of Experiments* (Wiley, New York, 2006)
44. J. Nummela, B. Julstrom, An effective genetic algorithm for the minimum-label spanning tree problem, in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (ACM, New York, 2006), pp. 553–557
45. Y.W. Park, Y. Jiang, D. Klabjan, L. Williams, Algorithms for generalized clusterwise linear regression. INFORMS J. Comput. **29**(2), 301–317 (2017)
46. A. Paul, D. Freund, A. Ferber, D.B. Shmoys, D.P. Williamson, Prize-collecting TSP with a budget constraint, in *25th Annual European Symposium on Algorithms (ESA 2017)*, ed. by K. Pruhs, C. Sohler. Leibniz International Proceedings in Informatics (LIPIcs), vol. 87 (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, 2017), pp. 62:1–62:14
47. N. Pholdee, S. Bureerat, Comparative performance of meta-heuristic algorithms for mass minimisation of trusses with dynamic constraints. Adv. Eng. Softw. **75**(1), 1–13 (2014)
48. R. Plackett, J. Burman, The design of optimum multifactorial experiments. Biometrika **33**, 305–325 (1946)
49. R.L. Rardin, R. Uzsoy, Experimental evaluation of heuristic optimization algorithms: a tutorial. J. Heuristics **7**(3), 261–304 (2001)
50. G. Reinelt, TSPLIB—a traveling salesman problem library. ORSA J. Comput. **3**(4), 376–384 (1991)
51. G. Rinaldi, RUDY: a generator for random graphs (1996). http://web.stanford.edu/~yyye/yyye/Gset/rudy.c. Accessed 30 Sept 2014
52. K.L. Sadowski, D. Thierens, P.A.N. Bosman, GAMBIT: a parameterless model-based evolutionary algorithm for mixed-integer problems. Evol. Comput. (2018, to appear)
53. H. Sawai, S. Kizu, Parameter-free genetic algorithm inspired by "disparity theory of evolution", in *Parallel Problem Solving from Nature – PPSN V*, ed. by A. Eiben, T. Bäck, M. Schoenauer, H.-P. Schwefel. LNCS, vol. 1498 (Springer, Berlin, 1998), pp. 702–711
54. J. Silberholz, B. Golden, The effective application of a new approach to the generalized orienteering problem. J. Heuristics **16**(3), 393–415 (2010)
55. K. Smith-Miles, S. Bowly, Generating new test instances by evolving in instance space. Comput. Oper. Res. **63**, 102–113 (2015)
56. K. Smith-Miles, L. Lopes, Measuring instance difficulty for combinatorial optimization problems. Comput. Oper. Res. **39**(5), 875–889 (2012)
57. K. Smith-Miles, J. van Hemert, Discovering the suitability of optimisation algorithms by learning from evolved instances. Ann. Math. Artif. Intell. **61**(2), 87–104 (2011)
58. K. Smith-Miles, D. Baatar, B. Wreford, R. Lewis, Towards objective measures of algorithm performance across instance space. Comput. Oper. Res. **45**, 12–24 (2014)
59. S. Subramanian, M.C. Jeffrey, M. Abeydeera, H.R. Lee, V.A. Ying, J. Emer, D. Sanchez, Fractal: an execution model for fine-grain nested speculative parallelism, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17 (ACM, New York, 2017), pp. 587–599
60. D. Sudholt, Parallel evolutionary algorithms, in *Springer Handbook of Computational Intelligence*, ed. by J. Kacprzyk, W. Pedrycz (Springer, Berlin, 2015), pp. 929–959

61. E.-G. Talbi, M. Basseur, A.J. Nebro, E. Alba, Multi-objective optimization using metaheuristics: non-standard algorithms. Int. Trans. Oper. Res. **19**(1–2), 283–305 (2012)
62. D. Taş, M. Gendreau, O. Jabali, G. Laporte, The traveling salesman problem with time-dependent service times. Eur. J. Oper. Res. **248**(2), 372–383 (2016)
63. L. Tierney, A.J. Rossini, N. Li, H. Sevcikova, Simple network of workstations (Package 'snow'), 2016. https://cran.r-project.org/web/packages/snow/snow.pdf
64. V.V. Vazirani, *Approximation Algorithms* (Springer, Berlin, 2013)
65. T. Vidal, T.G. Crainic, M. Gendreau, C. Prins, Heuristics for multi-attribute vehicle routing problems: a survey and synthesis. Eur. J. Oper. Res. **231**(1), 1–21 (2013)
66. X. Wang, B. Golden, E. Wasil, The min-max multi-depot vehicle routing problem: heuristics and computational results. J. Oper. Res. Soc. **66**(9), 1430–1441 (2015)
67. D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization. IEEE Trans. Evol. Comput. **1**(1), 67–82 (1997)
68. Y. Xiong, B. Golden, E. Wasil, A one-parameter genetic algorithm for the minimum labeling spanning tree problem. IEEE Trans. Evol. Comput. **9**(1), 55–60 (2005)
69. J. Xu, J. Kelly, A network flow-based tabu search heuristic for the vehicle routing problem. Transp. Sci. **30**(4), 379–393 (1996)