Comparison of Metaheuristics

John Silberholz and Bruce Golden

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 ([12]), tabu search ([16]), and genetic algorithms ([23]) have emerged. Due to the unique functionality of each type of metaheuristic, comparison of metaheuristics is in many ways more difficult than other algorithmic comparisons.

In this chapter, we discuss techniques for meaningful comparison of metaheuristics. In Section 2, we discuss how to create and classify instances in a new testbed and how to make sure other researchers have access to the problems for future metaheuristic comparisons. In Section 3, we discuss the disadvantages of large parameter sets and how to measure complicating parameter interactions in a metaheuristic's parameter space. Last, in Sections 4 and 5, we discuss how to compare metaheuristics in terms of both solution quality and runtime.

2 The Testbed

It seems natural that one of the most important parts of a comparison among heuristics is the testbed on which the heuristics are tested. As a result, the testbed should be the first consideration when comparing two metaheuristics.

John Silberholz
Center for Scientific Computing and Mathematical Modeling, University of Maryland, College Park, MD 20740 e-mail: josilber@umd.edu

Bruce Golden
R.H. Smith School of Business, University of Maryland, College Park, MD 20740 e-mail: bgolden@rhsmith.umd.edu

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, trends with regards to specific types of problem instances in the testbed can be made, making analysis of the new metaheuristic simpler.

2.2 Developing New Testbeds

While ideally testing on an existing testbed would be sufficient, there are many cases when this is either insufficient or not possible. 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 [15, 25], a different testbed will need to be used. Last, if the existing testbed is insufficient (often due to being too small to effectively test a heuristic), a new one will need to be developed.

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.

One 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 while giving the advantage of a guaranteed optimal solution. It is, therefore, critical that metaheuristic testing occurs on the large problems for which optimal solutions could not be calculated in reasonable runtimes. As discussed in [19], 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, [28]), it is typically time intensive to do this sort of data collection. Often real-world data is proprietary and, therefore, difficult to obtain. Furthermore, the problem instances generated will typically be small in number and size. For instance, real-world problem instances used for testing on the Generalized Traveling Salesman Problem proposed in [13] all had fewer than 50 nodes. In addition, there were only two real-world problem instances proposed; nearly all of the problem instances used in that paper did not come from real-world data, and all of the larger problem instances were artificial.

As a result, it is generally more reasonable to create a testbed based on existing well-known problem instances than it is to create one from scratch. For instance, many testbeds have been successfully made based off the TSPLIB. In the case of the Generalized Traveling Salesman Problem, [13] established a well-used testbed based on a simple extension to TSPLIB problem instances. Another example of such an extension can be found in [2], in which the authors tested several different modified versions of 10 benchmark VRP problems and reported computational results on each variation.

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 instance, the clustering algorithm proposed in [13] 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 the problem instances tested is another effective way to make them accessible. This was an effective technique used, for instance, in [7, 31].

In developing a new testbed, capturing real aspects of a problem is important. For instance, in the problem instances found in [13], the clustering algorithm placed nodes in close proximity to each other in the same cluster, capturing real-life characteristics of this problem.

2.2.3 Geometrically Constructed Problem Instances

One problem in the analysis of metaheuristics, as discussed in more detail in Section 4 is finding errors for the algorithms. Even when using advanced techniques, it is typically difficult to determine optimal solutions for large problem instances. A way to minimize the difficulty in this step is to use geometrically constructed solutions for which optimal or near-optimal solutions are apparent. This removes the burden on the 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 [6], 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 [5, 20, 21, 22].

2.3 Problem Instance Classification

Regardless of whether an existing or new testbed is used, classifying the problem instances being tested is critical to the proper analysis of heuristics. Differentiating factors between problem instances should be noted prior to any experimentation, and heuristic performance on each type of problem instance should be discussed. A good example of such analysis is found in [17], an experimental evaluation of heuristics for the resource-constrained project scheduling problem. That paper split problem instances by three key problem instance parameters, the network complexity, resource factor, and resource strength, analyzing the effects of each on the performance of the heuristics. 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 to determine which method will work the best for them.

3 Parameters

Though deciding upon a quality testbed is critical when comparing solution qualities and runtimes, it is also important to compare the actual algorithms. This can be accomplished in part by considering the complexity of the algorithms; if two algorithms produce similar results but one is significantly simpler than the other, then the simpler of the two is a superior algorithm. 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 and analyze.

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, as 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 algorithmic complexity is the number of parameters used in the algorithm.

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 instance, creating a genetic algorithm with a population size of 50) or based on the problem instance (for instance, 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 predetermined by the algorithm designer.

Each major type of metaheuristic has a number of parameters that must be set before algorithm execution. Consider Table 1, which lists basic parameters required for major types of metaheuristics. Though these are guidelines for the minimum

Table 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	None

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 [33] 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 [32] uses just one parameter, which functions to both control the population size and to serve as a termination criterion.

3.1 Parameter Space Visualization and Tuning

Metaheuristics using many parameters are more complex than procedures with few parameters for a number of reasons. First, the effort needed to tune or understand these parameters is far greater 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 n^m 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 2,187 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 in-

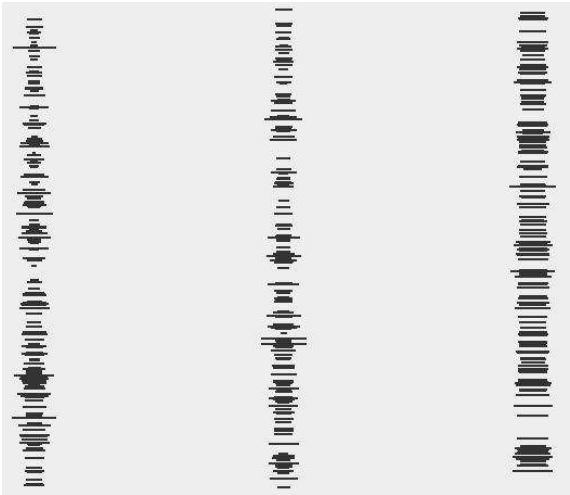

Fig. 1 Depiction of solution quality of a metaheuristic for the generalized orienteering problem over its 2-dimensional parameter space. The x-axis is the parameter i at 3 separate values and the y-axis is the parameter t over a large range of values. The widths in the figure represent error of the algorithm; a small width represents a small error.

creases. While, of course, there are far better ways to search for good parameter combinations than brute-force search, 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.

Larger numbers of parameters also make 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. We analyzed the 2-parameter metaheuristic due to [30] 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 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 from all the TSPLIB-based instances used. The optimal solutions are known for each of the five problem instances tested.

The resulting image, shown in Figure 1, is a testament to the simplicity of analysis of an algorithm with just 2 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 horizontal line is plotted with width normalized by the average error of the algorithm over the 20 runs for each of the five problem instances tested. A narrow width corresponds to an average error near the best performance of the testing, which is 2.53%, while a wide width corresponds to an average error near the worst performance of the testing, which is 4.08%. In a dense parameter space, the same sort of visualization could be gleaned by coloring dots with colors relating to the error or by presenting a 3-dimensional depiction, where the z-coordinate is the error.

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, ignoring a single outlier with higher error for low *i* and high *t*.

This sort of simplistic visual analysis becomes more difficult as the dimensionality of the parameter space increases. It is certainly possible to visualize a 3-dimensional parameter space in which the color at each point represents the solution quality of the algorithm with that parameter set, though this technique suffers from difficulties in viewing the interior points in a cubic parameter space with the exterior points in the way. Though visualizations of 4-dimensional spaces do exist (see, for instance, [18]), the visualizations do not provide information that is nearly as intuitive, decreasing the simplicity with which the parameter space can be visualized. Certainly no simple visualizations are available for 32-dimensional parameter spaces.

3.2 Parameter Interactions

This is not the only downside of metaheuristics with a large number of parameters. Another shortcoming is apparent in the susceptibility of a large parameter set to exhibit complex parameter interactions. 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 [10] 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 typically have at least the three parameters considered by Deb and Agrawal. However, there have been genetic algorithms developed that operate using only one parameter [32] or none [29], actually eliminating the possibility of parameter interactions.

Though to some degree there is parameter interaction in algorithms with a small number of parameters, we believe that the level of interaction increases dramatically with the number of parameters. To our knowledge, no research has been done on the effects of the number of parameters in a metaheuristic or heuristic on the parameter interactions for that algorithm. However, we propose a simple experiment to test this hypothesis.

First, the experimenter should select a combinatorial optimization problem for which a large number of metaheuristics have been developed. Reasonable choices might be the Traveling Salesman Problem or the Vehicle Routing Problem. Next, the experimenter should obtain implementations of a number of those metaheuristics, preferably of different types (genetic algorithm, tabu search, simulated annealing, ant colony optimization, etc.) and necessarily with a range of number of parameters.

The next step would be to test the parameter interactions using methods designed for this purpose on a representative set of problem instances for the problem considered. One method that could capture parameter interactions of any order would be a full factorial design, in which a reasonable maximum and minimum value is selected for each parameter and each combination of high and low values for each parameter is tested. However, the number of configurations tested with this method is exponential; a 32-parameter algorithm would require 4,294,967,296 configurations to be tested, which is almost certainly not reasonable. Even a 10-parameter algorithm, which is not uncommon in metaheuristics today, would require tests on over 1,000 configurations, likely a computational burden.

Thus, a better design might be the Plackett-Burman method [27], which requires 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 [24].

In either of these two designs, the number and magnitude of parameter interactions will be measured for each of the algorithms, and a comparison of the intensity of the interactions will be possible. We believe that not only will the number and magnitude of second-order interactions increase as the size of the parameter set increases, but the same will be true for the higher-order interactions measured through the full-factorial design (if it is possible to use this design).

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 these 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. The complementary subset can be used for testing and comparing metaheuristics. A full description of one such methodology can be found in [9].

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 plans. 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. Thus, when comparing metaheuristics it is not sufficient to determine if each heuristic meets a required solution quality threshold; comparison among the heuristics is necessary.

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, as 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. However, this is a presupposition that likely cannot always be made. If optimal solutions are available for every problem instance tested upon, the problem instances being considered are likely not large enough, since exact algorithms can provide solutions in reasonable runtimes.

This introduces the need for new metrics that can provide meaningful information without access to the optimal solution for all (or potentially any) problem in-

stances. 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 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 an optimal solution. An example of the successful application of this approach can be found in [14]. In that paper, a metaheuristic, optimal solution, and relaxation of that optimal solution are all created. Though the optimal solution was not run on the largest problem instances due to the excessive runtime required, the low error of the metaheuristic from the optimal solution on the smaller problems (0.25%) reinforces moderate deviations from the relaxed solutions over all problem instances (6.09%).

The metric can also be used for problems for which no optimal solution has been published, though the resulting deviations are less meaningful. It is unclear to a reader how well the algorithm performs without an understanding of how close the best-known solutions or tight lower bounds are to optimal solutions.

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

4.2 Multi-objective Solution Quality Comparisons

Though this section has focused on solution quality comparisons of single-objective heuristics, much work has also been done on the comparison of heuristics seeking to optimize multiple objective functions. For a detailed overview of multi-objective optimization and some of the difficulties encountered in comparing multi-objective metaheuristics, see [8]. For an example of the application of metaheuristics to multiobjective optimization problems, see [26].

5 Runtime Comparisons

While it is necessary that a metaheuristic demonstrate good solution quality to be considered viable, having a fast runtime is another critical necessity. If metaheuristics did not run quickly, there would be no reason to choose these approaches over exact algorithms.

At the same time, runtime comparisons are some of the most difficult comparisons to make. This is fueled by difficulties in comparing runtimes of algorithms that compiled with different compilers (using different compilation flags) and executed on different computers, potentially on different testbeds.

5.1 The Best Runtime Comparison Solution

The best solution is, of course, to get the source code for the algorithm, compile it on the same computer with the same compilation flags as your own code, and run both algorithms on the same computer. This is certainly the best solution in terms of runtime comparison, as the runtimes for a given problem are then directly comparable. Further, assuming the code can be obtained, this is a relatively simple way to compare the solution qualities. However, this technique for comparing algorithm runtimes 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 solution qualities (see, for instance, [4]), these benchmarks are susceptible to the type of program 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. As most papers do not report compiler optimization flags along with computational results, it would be difficult to obtain the exact scalar multiplier for a C program without additional information. Therefore, while the technique of obtaining a scalar multiplier between programming languages will almost certainly allow comparisons accurate to within an order of magnitude between algorithms coded in different programming languages, these methods cannot provide precise comparisons.

5.2 Other Comparison Methods

It is sometimes not possible to obtain the source code for the algorithm to which we compare. 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. While this does make runtime comparison harder, it does not excuse authors from performing

these computations — they are critical to the comparison of two algorithms. Two major approaches remain for a researcher to compare runtimes between the two algorithms, each with advantages and disadvantages.

The first is to reimplement another researcher's code in the same language as your code, running it on the same computer on the same problem instances. This has the advantage of actually running the same algorithm on the same hardware with the same compiler on the same computer, all positive attributes of a comparison. However, this approach suffers from two major weaknesses. First, some algorithms are not clear on certain details of the approach, making an exact reimplementation difficult. While statistical tests can be used to prove that solution qualities returned by the two algorithms are not statistically significantly different between the two implementations, this makes direct comparison of the results more 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, we believe it is fair to say that reimplementations typically overstate the runtime performance of a new algorithm over an existing one (see [3] for a humorous view of issues such as these).

The other approach does not suffer from these weaknesses. In this approach, published results of an algorithm over a publicly available dataset are compared to a new algorithm's results on the same dataset. While the dataset being tested is 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 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 [11] to derive conservative runtime multipliers between the two algorithms. These comparison tables are built by running a benchmarking algorithm (in the case of [11], 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 makes sure to mention 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." Hence, the multipliers gathered in this way can only provide a rough idea of runtime performance, clearly a downside of the approach.

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. Even in metaheuristics where such analysis is possible (for instance, a genetic algorithm with a fixed number of generations before termination), the worst-case runtime will often not be representative of how the algorithm will actually perform on problem instances, decreasing its value. As a result, the worst-case runtime is a bad choice for asymptotic analysis.

A much better approach for asymptotic analysis is fitting a curve to the runtimes measured for each of the algorithms. Regression analysis is a well-known technique that matches functions to a set of measurement points, minimizing the sum-of-squares error of the matching. These asymptotic 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), asymptotic runtime trends are key to runtime analyses. Even if one algorithm runs slower than another on small- or medium-sized problem instances, a favorable asymptotic runtime suggests the algorithm may well perform better on large-sized problem instances, where metaheuristics are most helpful.

5.4 An Alternative to Runtime Comparisons

Though the focus thus far has been on runtime comparisons, there are other forms of computational complexity comparison that do not rely on runtimes. One of the most intriguing, counting the number of representative operations the algorithm uses, is discussed in [1]. In this scheme, the number of a selected set of bottleneck operations is compared without any regard for the total execution time of the algorithms being compared.

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, providing complexity measures that are easier to replicate by other researchers. However, this approach suffers from the fact that it is often difficult to identify good operations that each algorithm being compared will implement. 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.

6 Conclusion

We believe following the procedures described in this paper 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. In addition, expanding the practice of creating geometric problem instances with easy-to-visualize optimal or near-optimal solutions will increase understanding of how metaheuristics perform in a global optimization sense.

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.

References

1. Ahuja, R., Orlin, J.: Use of representative operation counts in computational testing of algorithms. INFORMS Journal on Computing **8**(3), 318–330 (1996)
2. Archetti, C., Feillet, D., Hertz, A., Speranza, M.G.: The capacitated team orienteering and profitable tour problems (2009). Accepted for publication in *Journal of the Operational Research Society*
3. Bailey, D.: Twelve ways to fool the masses when giving performance results on parallel computers. Supercomputing review **4**(8), 54–55 (1991)
4. Bull, M., Smith, L., Pottage, L., Freeman, R.: Benchmarking Java against C and Fortran for scientific applications. In: ACM 2001 Java Grande/ISCOPE Conference, pp. 97–105 (2001)
5. Chao, I.M.: Algorithms and solutions to multi-level vehicle routing problems. Ph.D. thesis, University of Maryland, College Park, MD (1993)
6. Chen, S., Golden, B., Wasil, E.: The split delivery vehicle routing problem: Applications, algorithms, test problems, and computational results. Networks **49**, 318–329 (2007)
7. Christofides, N., Eilon, S.: An algorithm for the vehicle dispatching problem. Operational Research Quarterly **20**(3), 309–318 (1969)
8. Coello, C.: Evolutionary multi-objective optimization: A historical view of the field. IEEE Computational Intelligence Magazine **1**(1), 28–36 (2006)
9. Coy, S., Golden, B., Runger, G., Wasil, E.: Using experimental design to find effective parameter settings for heuristics. Journal of Heuristics **7**(1), 77–97 (2001)
10. Deb, K., Agarwal, S.: Understanding interactions among genetic algorithm parameters. In: Foundations of Genetic Algorithms, pp. 265–286. Morgan Kauffman, San Mateo, CA (1998)

11. Dongarra, J.: Performance of various computers using standard linear equations software. Tech. rep., University of Tennessee (2009)
12. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
13. Fischetti, M., Salazar González, J.J., Toth, P.: A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. Operations Research **45**(3), 378–394 (1997)
14. Gamvros, I., Golden, B., Raghavan, S.: The multilevel capacitated minimum spanning tree problem. INFORMS Journal on Computing **18**(3), 348–365 (2006)
15. Gendreau, M., Laporte, G., Semet, F.: A tabu search heuristic for the undirected selective travelling salesman problem. European Journal of Operational Research **106**(2–3), 539–545 (1998)
16. Glover, F.: Tabu search: A tutorial. Interfaces **20**(4), 74–94 (1990)
17. Hartmann, S., Kolisch, R.: Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research **127**(2), 394–407 (2000)
18. Hollasch, S.: Four-space visualization of 4d objects. Ph.D. thesis, Arizona State University, Tempe, Arizona (1991)
19. Jans, R., Degraeve, Z.: Meta-heuristics for dynamic lot sizing: A review and comparison of solution approaches. European Journal of Operational Research **177**(3), 1855–1875 (2007)
20. Li, F., Golden, B., Wasil, E.: Very large-scale vehicle routing: New test problems, algorithms, and results. Computers & Operations Research **32**(5), 1165–1179 (2005)
21. Li, F., Golden, B., Wasil, E.: The open vehicle routing problem: Algorithms, large-scale test problems, and computational results. Computers & Operations Research **34**(10), 2918–2930 (2007)
22. Li, F., Golden, B., Wasil, E.: A record-to-record travel algorithm for solving the heterogeneous fleet vehicle routing problem. Computers & Operations Research **34**(9), 2734–2742 (2007)
23. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer (1996)
24. Montgomery, D.: Design and Analysis of Experiments. John Wiley & Sons (2006)
25. Nummela, J., Julstrom, B.: An effective genetic algorithm for the minimum-label spanning tree problem. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 553–557. ACM (2006)
26. Paquete, L., Stützle, T.: Design and analysis of stochastic local search for the multiobjective traveling salesman problem. Computers & Operations Research **36**(9), 2619–2631 (2009)
27. Plackett, R., Burman, J.: The design of optimum multifactorial experiments. Biometrika **33**, 305–325 (1946)
28. Reinelt, G.: TSPLIB—a traveling salesman problem library. ORSA Journal on Computing **3**(4), 376–384 (1991)
29. Sawai, H., Kizu, S.: Parameter-free genetic algorithm inspired by "disparity theory of evolution". In: A. Eiben, T. Bäck, M. Schoenauer, H.P. Schwefel (eds.) Parallel Problem Solving from Nature – PPSN V, *LNCS*, vol. 1498, pp. 702–711. Springer Berlin / Heidelberg (1998)
30. Silberholz, J., Golden, B.: The effective application of a new approach to the generalized orienteering problem (2009). Accepted for publication in *Journal of Heuristics*
31. Wang, Q., Sun, X., Golden, B.L.: Using artificial neural networks to solve generalized orienteering problems. In: C. Dagli, M. Akay, C. Chen, B. Fernández, J. Ghosh (eds.) Intelligent Engineering Systems Through Artificial Neural Networks: Volume 6, pp. 1063–1068. ASME Press, New York (1996)
32. Xiong, Y., Golden, B., Wasil, E.: A one-parameter genetic algorithm for the minimum labeling spanning tree problem. IEEE Transactions on Evolutionary Computation **9**(1), 55–60 (2005)
33. Xu, J., Kelly, J.: A network flow-based tabu search heuristic for the vehicle routing problem. Transportation Science **30**(4), 379–393 (1996)