Contents

Chapter 8. Experiments

Eric Berberich[1], Matthias Hagen[2⋆], Benjamin Hiller[3⋆⋆], and Hannes Moser[4⋆⋆⋆]

[1] Max-Planck-Institut für Informatik,
D–66123 Saarbrücken, Germany
eric@mpi-inf.mpg.de

[2] Web Technology and Information Systems Group,
Faculty of Media / Media Systems
Bauhaus University Weimar
D–99423 Weimar, Germany
matthias.hagen@uni-weimar.de

[3] Department Optimization, Zuse-Institute Berlin,
D–14195 Berlin-Dahlem, Germany
hiller@zib.de

[4] Institut für Informatik, Friedrich-Schiller-Universität Jena,
D–07743 Jena, Germany
hannes.moser@uni-jena.de

1.1 Introduction

Experimentation plays an important role in the Algorithm Engineering cycle. It is a powerful tool that amends the traditional and established theoretical methods of algorithm research. Instead of just analyzing the theoretical properties, experiments allow for estimating the practical performance of algorithms in more realistic settings. In other fields related to Computer Science, like for instance Mathematical Programming or Operations Research, experiments have been an indispensable method from the very beginning. Moreover, the results of systematic experimentation may yield new theoretical insights that can be used as a starting point for the next iteration of the whole Algorithm Engineering cycle.

Thereby, a successful experiment is based on extensive planning, an accurate selection of test instances, a careful setup and execution of the experiment, and finally a rigorous analysis and concise presentation of the results. We discuss these issues in this chapter.

1.1.1 Example Scenarios

In the Algorithm Engineering cycle, experimentation is one of the four main steps besides design, theoretical analysis, and implementation. There are many reasons why experiments are that important. We give a few examples here.

⋆ Supported in part by a Landesgraduiertenstipendium Thüringen.

⋆⋆ Supported by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/10-3, GR 883/10-4).

⋆⋆⋆ Supported by the Deutsche Forschungsgemeinschaft, project ITKO (iterative compression for solving hard network problems), NI 369/5.

1. The analysis shows a bad worst-case behavior, but the algorithm is much better in practice: The worst-case behavior may be restricted to a small subset of problem instances. Thus, the algorithm runs faster in (almost) all practically relevant cases.
2. A theoretically good algorithm is practically irrelevant due to huge constants hidden in the “big Oh” notation.
3. A promising analysis is invalidated by experiments that show that the theoretically good behaviour does not apply to practically relevant problem instances.
4. A specific algorithm is hard to analyze theoretically. Experimental analysis might provide important insights into the structure and properties of the algorithm.
5. Experiments lead to new insights that can be used in the next cycle of the Algorithm Engineering process.

In the following, we discuss an example for each of these situations in more detail.

Example 1: Quite often experimenters observe a considerably better running time behavior of an algorithm than predicted by theory. Thus, the worst-case behavior is restricted to a very small subset of problem instances. A classic example is the simplex method for linear programming, whose running time is exponential in the worst case. However, its practical running time is typically bounded by a low-degree polynomial [1].

Example 2: In algorithm theory, an algorithm is called efficient if the asymptotical running time is small. However, in many cases there exists a hidden constant factor that makes the algorithm practically useless. An extreme example in graph theory is Robertson and Seymour’s algorithm for testing whether a given graph is a minor of another [52, 53]. This algorithm runs in cubic time, however, the hidden constant is in the order of 10^{150}, making the algorithm completely impractical. Another example of this kind is Bodlaender’s linear-time algorithm which determines for a given graph and a fixed k whether the graph has treewidth at most k [6]. Unfortunately, even for very small values of k, the implemented algorithm would not run in reasonable time. The “big Oh” notation facilitates the design of algorithms that will never get implemented, and the actual performance of an algorithm is concealed. Moreover, algorithms often rely on other algorithms in several layers, with the effect that an implementation would require an enormous effort. Thus, the “big Oh” is in some sense widening the gap between theory and practice.

Example 3: Moret and Shapiro tested several algorithms for the minimum spanning tree problem (Minimum Spanning Tree) using advanced algorithm engineering methods [47]. They analyzed the following algorithms: Kruskal’s, Prim’s, Cheriton and Tarjan’s, Fredman and Tarjan’s, and Gabow et al.’s. They tried several different data structures (i. e., different kinds of heaps) and several variants of each algorithm. Moret gives a concise survey of this work [45]. The interesting result is that the simplest algorithm (Prim’s) was also the fastest in their experiments, although it does not have the best running time in theory. The other algorithms are more sophisticated and have better worst-case asymptotic running time bounds. However, the sophistication does not pay off for reasonable instance sizes. Moret also stresses the value of algorithm engineering: By studying the details of data structures and algorithms one can refine the implementation up to the point of drawing entirely new conclusions, which is a key aspect of algorithm engineering. With this methodology, Moret and Shapiro's fastest implementation of Prim's algorithm got nearly ten times faster than their first implementation.

Example 4: This example is about algorithms whose theoretical analysis is extremely difficult, like for instance Simulated Annealing, Genetic Algorithms, and Union-Find with path compression. Both the analysis of the running time and of the solution quality is very difficult using existing methods. For instance, Union-Find with path compression is relatively easy to describe and was known to yield very efficient behavior. However, its exact characterization took many years till Tarjan achieved a proof of tight bounds [61]. In such cases, experimental analysis can be a fruitful alternative that yields interesting results more efficiently.

Example 5: As a last example we want to mention the Traveling Salesman Problem. In an incremental process, the methods to solve that problem (exactly or approximately) became more and more sophisticated over the years. Beginning with a few hundred cities, researchers are now able to solve instances of more than ten thousand cities [3]. In Section 1.6.1, the Traveling Salesman Problem is also used as an example of how to analyze results of experiments graphically.

1.1.2 The Importance of Experiments

The examples of the last section showed the importance of experimentation in the Algorithm Engineering cycle for just a few situations. This section is dedicated to describe more generally the motivation to conduct experiments. It is based mainly on several articles and surveys [27, 32, 43, 45].

For the second step of the Algorithm Engineering process, that is, the analysis of an algorithm, there exist usually three different methods, namely worst-case analysis, average-case analysis and experimental analysis. The theoretical methods are more sophisticated than experimental analysis. Since the early days of Computer Science theoretical analysis and experiments have been used. Computing pioneers such as Floyd and Knuth combined theoretical analysis and experiments. They used machine-dependent fine-tuning to derive efficient algorithms that performed well both in theory and practice. However, later on the focus has lain on theoretical analysis, whereas experiments were mainly used in other fields. From the two ways of analyzing algorithms, only theoretical analysis developed into a science. Since there is still missing a well-established methodology for experimentation, the quality of works in this discipline varies strongly, and the results are difficult to compare and to reproduce. This disequilibrium has to be balanced by deliberate experimental analysis.

Recently, the interest in experimental analysis has grown. There are various reasons for this newly arisen interest. One might be that computer scientists become aware that theoretical analysis cannot reveal all facets of algorithmic behavior, especially when concerning real-world applications. Of course many other reasons, for instance the fact that computational experiments are much cheaper these days, might have helped too.

This newly arisen interest is also reflected in an increasing number of publications in the field. Some major contributors are:

- Jon Bentley's *Programming pearls* columns in *Communications of the ACM* and his *Software Exploration* columns in *UNIX Review.*
- David Johnson initiated the Annual ACM/SIAM Symposium on Discrete Algorithms (SODA), which also invites a few experimental studies.
- The *ACM Journal of Experimental Algorithmics* (ACM JEA) was initiated to give a proper outlet for publications in the field of computational experiments.
- The *Engineering and Applications Track* at the *European Symposium on Algorithms* (ESA), which was formerly known as *Workshop on Algorithm Engineering* (WAE).
- The *Workshop on Algorithm Engineering and Experiments* (ALENEX).
- The *International Symposium on Experimental Algorithms* (SEA), which was formerly known as *Workshop on Experimental Algorithms* (WEA).

Compared to theoretical analysis, experimentation in Computer Science is still in the "fledgling stages." In other (natural) sciences, like for instance physics, theories are completely based on experiments. Scientists have developed mature methods to derive meaningful results out of experimentation (mature in the sense that they have been revised and approved many times). Computer science lacks such well-established methods, which are generally accepted as a standard for empirical studies by the community. Obviously, Computer Science differs in many ways from other natural sciences. For instance, on the one hand in natural science the results of theories are compared to a golden standard (the nature). On the other hand, in Computer Science we just report results or compare them with another experiment of the same type. Moreover, Computer Science is much easier to understand: In principle, we could derive nearly any information about a given program by profound analysis. In Computer Science, unlike other natural sciences, we know — at least in principle — the underlying mechanisms, like for instance source code, compilers, and computer architecture, that yield our results. But unfortunately, the processes we observe are by far too complex to be understood easily.

Therefore, like in other sciences, we need an empirical science of algorithms to be able to invent evidence-based explanatory theories. Certainly, this does not mean that theoretical Computer Science will become obsolete. There is absolutely no reason to abandon theoretical analysis, as it has proved to serve perfectly to draw many important conclusions, to gain a deeper insight into problems and to help to design new data structures and algorithms. However, theoretical analysis should be supplemented with experimentation, which is exactly the goal of the whole cyclic process of Algorithm Engineering.

With this approach, we would hopefully narrow the big gap between theory and practice, helping people to benefit more directly from the deep understanding of problems and algorithms gained by theory. Since experimental work is often considered not worthwhile and rejected by theorists, it is important to stress that empirical science is not the opposite of theory (e. g., quantum electrodynamics shows that an empirical science can be rather theoretical), at least when it would be evolving to a real science. We think that mainly the deficiency of unassailable and clean scientific experimental work and research principles are the cause for the lack of major success of experiments in algorithmics. In this chapter we give an overview of approaches that aim to resolve these problems.

1.1.3 The Experimentation Process

The task of experimentation is to answer a formulated hypothesis or a question using meaningful test data that has been obtained by some reproducible process, the experiment. Reproducibility here means that the experiment can be repeated, yielding qualitatively the same results and conclusions. A research experiment should have a purpose, be stated and defined clearly prior to the actual testing, and, of course, it is important to state the reason why experimentation is required.

The experimenter has great latitude in selecting the problems and algorithms. He has to decide how to implement the algorithm (see Chapter 6), to choose the computing environment, to select the performance measures, and he has to set the algorithm options. Furthermore, he is responsible for generating a good report which presents the results in an appropriate way. These choices can have a significant effect on the results, the quality, and the usefulness of the experiment as a whole. Therefore, the experimenter has to plan his experiments with care. He should document all decisions such that the experiment can be reproduced at any time. In order to improve the quality of experiments, the planning should be done following some systematics.

In the literature (i. e., [4, 45]), experimentation is a process whose steps can be described as follows.

1. Define the goals of the experiment (Section 1.2).
2. Choose the measures of performance and factors to explore (Section 1.2).
3. Choose a good set of test instances (Sections 1.3 and 1.4).
4. Implement and execute the experiment (Section 1.5).
5. Analyze the data and draw conclusions (Section 1.6, see also Chapter 4).
6. Report the experiment's results (Section 1.7).

Note that this process is in almost every case an iterative process, meaning that it might be necessary to go back to some earlier step to revise some of the decisions made earlier. The process is often also incremental in the sense that the results motivate further experiments to answer new questions. In the following we shortly describe what an experimenter should consider in each step. Then, each step will be described in more detail in the corresponding sections of this chapter.

Define the goals of the experiment. There are manifold types of experiments, having its seeds in different motivations. At first, the researcher has to find out which type of experiment is needed. Depending on that type, the experiment and the presentation of the results have to be adapted properly. For that reason it is always helpful to define primary goals for the experiment. These goals should always be kept in mind during the whole experimentation process. Another important question in the first step is the newsworthiness of the experiment, that is, whether the results are interesting and whether they have the potential to lead to new valuable insights. We discuss these issues briefly in Section 1.2.1, where we also shortly subsume literature we consider worth reading.

Choose the measures of performance and factors to explore. Depending on the problem and the type of experiment, the experimenter has to select the *measures* (e. g., running time) that are suited for a good understanding of underlying processes of the algorithm and that describe its performance at its best. We discuss how to find good measures, and we present some standard measures as well as other important alternatives in Section 1.2.2. With the measures we are also facing the task of obtaining their values. Several techniques exist to improve *data quality* as well as the *speed* of the experiment, which we discuss in Section 1.2.4.

Another important question of the second step is the choice of the *factors*, that is, choosing the properties of the *experimental environment* or setup that influence the result of the experiment (e. g., the input size). Some factors have a major influence on the measures, others are less important. The experimenter's task is to choose the factors that permit to analyze the algorithm as good as possible. This task is described in Section 1.2.3.

Choose a good set of test instances. The test instances used in algorithmic experiments directly affect the observed behavior of the tested algorithms. After characterizing some *fundamental properties* that should influence the choice of test instances in Section 1.3.1 we identify *three different types* of test instances used in most experiments. We analyze their respective strengths and weaknesses in Section 1.3.2 before giving some final suggestions on how to choose good test instances in Section 1.3.3.

For many problems collections of test instances are already available on the Internet. We call such collections *test data libraries* and describe properties of a perfect library in Section 1.4.1. The issues arising in the context of creating and maintaining a library are discussed in Sections 1.4.2 and 1.4.3. A brief compendium of existing libraries follows in Section 1.4.4.

Implement and execute the experiment. Executing the experiments seems to be a trivial task, since the computer actually *does the job*. If done without care, the obtained results are just useless.

Section 1.5.1 explains what to consider when setting-up the *laboratory*, so that the experimenter can work in a nice and clean environment that eliminates systematic errors. The actual work in experimentation is done by the computer. It runs all experiments, but the human operator has also some tasks. Section 1.5.2 gives advice on how to make the running phase simple without losing information or introducing new errors.

Analyze the data and draw conclusions. The data generated by the experiment needs to be analyzed carefully in order to draw sound conclusions. In Section 1.6.1 we give advice on how to employ graphical methods to analyze the data. The focus is on using diagrams to reveal information that might not be obvious.

Section 1.6.2 complements this rather informal approach with an overview on using statistical methods for data analysis. We start giving a brief overview on the basic concept of a *hypothesis test* as a major statistical tool. Instead of going into further details of statistical analysis, we rather try to capture general ideas of using it in the context of algorithm analysis by describing studies and results found in the literature. The goal is to provide an overview and to somehow give the flavor of the methods.

For the more general question of how to use experiments in order to analyze the *asymptotic* running time of algorithms we refer to Chapter 4, especially to Section 4.8. One general suggestion (made in Section 4.8.1) is to make use of the scientific method (known from the natural sciences), that is, to combine theoretical deductive reasoning and experimental analysis to reach the best possible overall result. But apart from that, Section 4.8.2 describes and preliminarily assesses a specific approach to finding *hypotheses* on the asymptotic running time of algorithms *by pure analysis of experimental data.*

Report the experiment's results. Proper reporting of the results and the details of the experiment is very important for a good experimental study. Too many papers reporting experimental results have failed to achieve the main requirement for a good experiment: Being reproducible for doing further research. Section 1.7 deals with good practices for proper reporting and mentions pitfalls and problems to watch out for. We also give some hints on how to make the best out of diagrams and tables, in order to substantiate the claims and findings of the experiment.

Here we give some publications we consider worth and important to read before getting started.

Moret's paper [45] is a good starting point. It generally describes existing experimental work and briefly sketches the whole experimentation process from the planning to the presentation of the results. A more comprehensive work is Johnson's paper, which principally addresses theorists [32]. It describes how to write good papers on experiments, and it includes many recommendations, examples, and common mistakes in the experimentation process. Another recommendable paper from Hooker motivates experimentation in general [27]. It highlights the advantages of experimentation, states with which kind of prejudice it is often confronted. Furthermore, Hooker gives a nice comparison with natural sciences, and he presents some examples where experimentation is successfully applied. The paper by Barr et al. focuses on experiments with heuristic methods [4]. However, people from other areas might also find some interesting aspects and observations in this paper. McGeoch's paper [40] mainly concerns the questions of how to obtain good data from experimentation and how to accelerate experiments significantly. She proposes the use of *variance reduction techniques* and *simulation speedups*.

Each of these publications describes experimental work from a slightly different point of view, however, the authors basically agree in their description of the experimentation process in general.

1.2 Planning Experiments

This section describes the test planning, what an experimenter should think about *before* implementing the algorithm and starting to collect data. The planning of an experiment is a challenging process that takes a considerable amount of time. However, a careful plan of the experiment prevents many types of severe problems in later steps of the experiment. Planning is a necessary requirement in order to do high quality experimental research.

1.2.1 Introduction

First of all, we have to think about the motivation to perform an experiment. There are many reasons to conduct experimental research. In the literature we can find many different types of experiments with diverse motivations [32, 45, 4]. Depending on what an experimenter is trying to show, the corresponding experiment and the report of its results have to be adapted properly. There is a wide range of possible goals of an experiment, the following list states a few.

– Show the superiority of an algorithm compared with the existing ones.
– Show the relevance of an algorithm for a specific application.
– Compare the performance of competing algorithms.
– Improve existing algorithms.
– Show that an existing algorithm performs surprisingly bad.
– Analyze an algorithm/problem to better understand it (experimental analysis).
– Support/reject/refine conjectures on problems and algorithms.
– Checking for correctness, quality, and robustness of an algorithm.
– Develop refined models and optimization criteria.

In the planning step we have to define a clear set of objectives, like questions we are asking and statements we want to verify.

Another important part of that planning step is to verify the newsworthiness of the experiment. That is, whether the experiment would actually give us interesting new insights. One way to achieve newsworthiness is to answer interesting questions on a sound basis, going beyond pure running time comparison. We give a few examples:

– Does the performance of several algorithms for the same problem differ and do some of the algorithms clearly dominate the others? (Statistics can help here.)

- Does dominance hold for all instances or only for some subset? If so, what are the structural properties of this subset?
- What are the reasons for dominance (e. g., structural properties of inputs, comparison of operation counts for critical operations)?
- How much does each phase of the algorithm contribute to the running time/performance of the algorithm?

These questions cannot be answered quickly. They have to be considered in the whole experimentation process.

1.2.2 Measures

By a *measure* of performance we generally mean quantities related to the algorithm and obtained by the execution of the experiment. There are several widely used measures that are quasi standard. However, each measure has its advantages and disadvantages. Thus, the correct choice of an appropriate measure can be crucial for a good understanding and analysis of the experiment.

In the first part of this section, we describe several well-known as well as some more exotic measures that appeared in literature. Then, we briefly describe how to generally find good measures.

Three measures are used in almost any publication about experimental algorithms:

- running time
- space consumption
- value/quality of the solution (heuristics and approximation algorithms)

Depending on the type of experiment, at least one of these measures is a must-have. However, these popular measures should not be used solely. The first two measures highly depend on the chosen programming language, compiler, and computer (processor, cache, memory, . . .), and therefore the results are very difficult to generalize and to compare. Furthermore, they depend on the implementation style and the skill of the programmer. Therefore, some investigators therefore assure that all crucial parts are implemented by the same programmer, e. g., as described in [45]. Running times in particular are problematic when they are very small. Because the system clock's granularity cannot be chosen arbitrarily, we get distorted results. However, this can be resolved by several runs with the same input data set. The choice of the test instances, as described in Section 1.3, also has a strong influence, especially on the value/quality of the solution in the case of heuristics.

Most notably, it is very unlikely that a good understanding of the problem and the algorithm emerges from these measures. They are aggregate measures that do not reveal much about the algorithm's behavior (for instance, we cannot discover *bottleneck operations*, which are fundamental operations that are performed repeatedly by the algorithm and influence the running time at most).

We need other measures in order to gain a deeper understanding of the algorithms to test. Moret recommends to “always look beyond the obvious measures” [45]. In the following we describe some other measures that appear in literature (see, e. g., [1, 4, 45]).

Extensions of running time First of all, it is sometimes useful to extend the notion of running time. For instance, in the case of heuristics, we might measure the time to find the best-found solution, that is, the time required to find the solution that is used to analyze the quality of the heuristics. Moreover, there exists a difference between the time that is required to produce the best-found solution and the total time of the run that produced it. In the case of heuristics that are multi-phase or composite (i. e., initial solution, improved solution, final solution), the time and the improvement of quality in each phase [4] should be measured, too.

Structural measures For a good understanding of the algorithm we need structural measures of various kinds (e. g., number of iterations, number of calls to a crucial subroutine, memory references, number of comparisons, data moves, the number of nodes in a search tree). Several publications recommend the use of memory references (mems) as a structural substitute for running time [1, 34, 45]. But other measures, like for instance the number of comparisons, the number of data moves (e. g., for sorting algorithms), and the number of assignments, should be considered as well, depending on the algorithm to be analyzed.

Bottleneck operation counts The idea of counting the number of calls to a crucial subroutine, or to count the number of executions of a major subtask, leads to the general concept of *asymptotic bottleneck operation*. We call an operation an *asymptotic nonbottleneck operation* if its fraction in the computation time becomes smaller and approaches zero as the problem size increases. Otherwise, we call the operation an *asymptotic bottleneck operation*. In general, there exists no formal method for determining asymptotic bottleneck operations, since an algorithm might behave completely different for small instances than for sufficiently large instances. However, it seems to be a quite useful approach in practice [1]. Bottleneck Operation Counts are also often used when comparing heuristic optimization algorithms. In this case, the evaluation of the fitness function is often the bottleneck when running the algorithm on real-world problems. For a more detailed description we refer to Chapter 4.

Virtual running time Ahuja et al. advocate the use of *virtual running time* [1]. The virtual running time is an estimate of the running time under the assumption that the running time depends linearly on “representative operations” (potential bottleneck operations). The loss of accuracy, that is, the estimate of the running time compared with the actual running time, can be remarkably small. Ahuja et al. present case studies with a difference of at most 7%, in many cases below 3%. Virtual running time can be used to detect asymptotic bottleneck operations, it is particularly well-suited for tests on various systems, and it permits us to eliminate the effects of paging and caching in determining the running times. We refer the reader to Chapter 4 for a more detailed description of this notion.

Finally, we want to describe some notions that are not measures in the strong sense, but considered as such in some publications since their impact is generally underestimated by many experimenters.

The first "measure" of this kind is robustness. If an algorithm performs well or the computed solution has a good quality only for a few problem instances it is evidently not very interesting in a general setting. Therefore, an algorithm should perform well over a wide range of test instances. For instance, one could measure the number of solved instances of a benchmark library of hard instances in order to estimate the robustness of an algorithm. The second "measure" we want to mention is the ease of implementation. There are many examples of algorithms that have been selected for use in practice just because they are easy to implement and understand, although better (but more complicated) alternatives exist. Not only the running time is important, but also the time needed for the implementation of the algorithm. Especially if the running time is not a crucial factor, the ease of implementation (e. g., expressed in lines of code, or by estimating the man-months needed for an implementation) can be an important argument in favor of some algorithm. Note that the ease of implementation depends highly on the underlying programming language, programming tools, and the style of the programmer, among many other influences. The third "measure" to mention is scalability, which basically means that algorithms can deal with small as well as large data sets. Obviously, these "measures" cannot be determined very exactly, but even a very rough estimate can help to better classify the algorithm in question. However, it is important to stress that these "measures" are limited and therefore they should be applied with care. Note that these "measures" are also presented as design goals in Chapter 3.

As stated before, good measures that help understanding an algorithm are usually not the most obvious ones. Therefore, we briefly discuss how to find such good measures in practice. Several authors give various hints on this issue, for instance, Johnson states a nice list of questions to ask in order to find the right measures [32]. At the beginning, it is recommended to do research subsumed as *exploratory experimentation*. One of the first experiments could be to observe how the running time of the algorithm is affected by implementation details, parameter settings, heuristics, data structure choices, instance size, instance structure, and so forth. Furthermore, we might check if a correlation between running time and the count of some operations exists. Then, we try to find out the bottlenecks of the algorithm. It is also interesting to see how the running time depends on the machine architecture (processor, cache, memory, . . .), and how the algorithm performs compared with its competitors. Obviously, these experiments should be conducted with other (standard) measures as well (e. g., replace "running time" with "space consumption" in the above description). Profilers can and should be used to quickly find good structural measures.

In general, a look should be taken at data representing differences as well as ratios. Furthermore, one should use measures that have small variance within a sampling point (which will be defined in the following section) as compared to the variance observed between different sampling points [4].

1.2.3 Factors and Sampling Points

The *factors* of an experiment comprise every property of the experimental environment or setup that influences the result of the experiment (i. e., the measures) [4]. The most obvious factors are the parameters of the algorithm, but we also consider other influences as, for instance, the computing environment. The experimenter has to find out which factors have a major influence on the measures. He has to define what to do with other factors that are not important or cannot be controlled. Factors generally can be expressed by some value, for instance the processor speed, the memory usage, or the value of some configuration variable for an algorithm. We refer to such values as a *level* of a factor [4]. For a run of an algorithm we have to define a *sampling point*, that is, we have to fix the factors at some level. The experimenter has to define which sampling points will be considered in the experiment, and how many runs should be performed for each sampling point.

By applying some preliminary tests, we can find out which factors actually do have a major influence, as for instance the input size, the number of iterations for an approximation algorithm, threshold levels, algorithms to solve subproblems (e. g., sorting and data structures), characteristics of the test instances, and the machine architecture. Among these, the experimenter has to pick out the ones he is interested in. These factors will possibly be altered during the experimental analysis to set up new experiments. For such factors, we have to decide which levels should be selected. This decision depends highly on the purpose of the experiment and the questions that are asked. The levels are the specific types or amounts that will be used in each run. For instance, if the factor is quantitative, then we have to choose the values we consider and how they are spaced (e. g., the selected levels of the factor "input size" could be $10, 10^2, 10^3, 10^4, \dots$). Note that qualitative factors make sense as well. For instance, to classify test instances as small, medium size, or big, or to choose a certain type of data structure (e. g., binary tree, hash), or looking at boolean values (e. g., optimization on or off).

Other factors might not be interesting for the experimenter. In this case, he has to fix them on a certain level for all runs. Obviously, a good reason for choosing a certain level must exist. For instance, the factor "main memory" could be fixed at 1024 MB, but this is only reasonable if there is evidence that the memory usage of the tested algorithm will never even get close to that amount.

Finally, some factors might exist that are ignored, because we assume them not to influence the outcome or having a sufficiently low influence. Of course, there must be evidence for this assumption. For instance, if we measure the running time by looking at the processor time of the algorithm, and if one factor is the "user load" of the machine on which we perform the experiment, then we might ignore the user load because we trust in the operating system that the measured processor time is computed correctly. Other factors that have to be ignored out of necessity are factors that we do not understand or cannot control.

For instance, such a factor could be the total load of the machine where the experiment is performed. Even if we assure that no other important processes are running, the necessary operating system's processes themselves cannot be controlled that easily. Especially for such factors, it is recommended to randomize them if possible, in order to keep an undesired influence as low as possible [42]. The process of finding good factors can take quite some time, and it is important to document the whole process. The finding of good factors is also part of the running phase of the experiment, which is described in more detail in Section 1.5.2.

For each run of the experiment a sampling point has to be chosen. With the number of factors that have to be altered the number of possible sampling points increases, since in theory we could try all possible combinations of factor levels. In most settings, this number is by far too high in order to perform an experiment for every possible sampling point. Therefore, the experimenter has to select a reasonable number of sampling points that reflect the overall behavior of the tested algorithm as good as possible. In order to decrease the variance of the test data, the experimenter also has to consider that the experiment should possibly be run several times for the same sampling point (see also Sections 1.2.4 and 1.6.2). With a good and elaborated prior selection of sampling points (by always having the primary goals of the experiment in mind), the experimenter can avoid many useless experiments that use up expensive resources. Furthermore, it can be avoided to have experiments run again in a later step, because it became clear in the analysis that the used sampling points were not adequate or sufficient.

Finally, we want to mention a comprehensive approach for experimental design called DOE (Design Of Experiments), which especially deals with the careful design of experiments and the choice of factors and sampling points in order to allow a sound subsequent statistical analysis [4, p. 20]. For more information about DOE we refer to Section 1.6.2.

1.2.4 Advanced Techniques

In this section we briefly explain advanced methods that should be considered when planning an experiment, like for instance simulation [38,40,42,41,49], simulation speedup as well as variance reduction techniques [40]. They have the goal to improve the process of obtaining experimental data. In a nutshell, simulation speedup deals with the question of how to speed up the process of obtaining data, i. e., how to make the test runs faster. With faster test runs, we can obtain more data in less time, helping us to decrease the variance notably. However, not only simulation speedup can reduce variance, but other more sophisticated techniques for this purpose exist. Conversely, a reduced variance admits fewer test runs, thus, speeding up the entire process of gathering data. Note that these techniques are not a luxury additive in test design. Often, much improvement may be needed for the data to be useful.

A common paradigm in simulation research is to differentiate between a real-world process (e. g., an economic system, weather, public transport) and a mathematical model of such a process, to predict its future behavior in reality. For the purposes of algorithm design, the real-world process is an application program running in a particular computing environment, whereas the mathematical model is the underlying algorithm. If the algorithm cannot be analyzed sufficiently, then a simulation program is developed, which may be identical with the application program. Not all researchers make this distinction, as well as we did not mention it before this point. However, this point of view is useful to explain the following techniques.

Usually, we have to deal with measures that are influenced by random noise. Thus, we get different numerical values for each test run. In order to get a reliable value for the measure, we repeat the test run several times and compute the mean value over all test runs. However, for measures with high variance (or "spread"), we need a high number of runs in order to get a reliable mean value with low deviation. In the following we outline several known approaches to reduce variance, where only the intuitive idea behind each technique is described. More exact mathematical descriptions of these techniques can be found in the literature (e. g., [40]).

Common Random Numbers This technique should be considered when comparing two algorithms on randomly generated instances and we expect that the compared measure is positively related with respect to the input instances. The idea is to use the same random instance for each test run of the two algorithms, which is equivalent to generate the instance from the same random numbers, hence the name. Since the measure is assumed to be positively related, the variance of the difference of the measures of the two test runs for each random instance is expected to be lower than the variance of the measure of each algorithm separately. A positive side effect of this technique is that we have to compute only half the number of test instances compared to the situation where we generate a random instance for each algorithm separately.

Control Variates If there are two measures of the same algorithm that are positively correlated, then we can make use of this technique to decrease the variance of one such measure. Suppose that the running time and the memory usage of an algorithm correlate positively, i. e., the algorithm needs more memory if it is running for a longer time. For each test run, we compute the difference between the mean value of the memory usage and the memory usage observed in that run. Due to the positive correlation, we can use this difference to "correct" the value of the running time for that test run. This method provably reduces the variance of the running time values.

Antithetic Variates The idea behind this technique is simple: If we have two measures that have the same distribution, but are negatively correlated, then the sum of these two measures has a lower variance. Namely, if the difference between the first measure and its mean is positive, then the difference between the second measure and its mean is likely to be negative. Thus the sum of both measures compensates the deviation of each measure, and therefore the "sum measure" has a reduced variance.

Conditional Expectation This technique is sometimes also called “Conditional Monte Carlo” or “Conditional Mean”. Suppose we have two measures, for which we know that the mean of the first measure is a function of the mean of the second measure. For each test run, rather than obtaining the first measure directly, we can also take the second measure and then compute the corresponding value of the first using the known function. This method works if the variance of the second measure is smaller than the variance of the first measure.

These were just four important techniques that are most likely to be generally applicable. In literature, many other techniques of this type can be found.

Next, we address simulation speedup. Until now, the idea was to implement an algorithm and then perform the tests directly on it. The key idea of simulation speedup is to partially simulate the algorithm. Because sometimes it is not necessary to implement it as a whole, we might skip parts of the implementation and replace them by a simulation. This is more efficient due to knowledge which the implemented algorithm would not have.

Variance reduction and simulation speedup are closely related. With simulation speedup, we are automatically able to reduce variance, as the improved efficiency permits us to take more trials within the same amount of time. Conversely, a smaller variance implies that less trials are needed. Thus, the overall running time decreases. McGeoch [40] gives several examples of algorithms to which these techniques have been successfully applied.

1.3 Test Data Generation

When evaluating algorithms experimentally, the experimenter usually runs the algorithms on several test instances while measuring interesting values. Obviously, the used test instances may substantially affect the observed behavior of the algorithms. Only a good choice of test instances can result in meaningful conclusions drawn from the respective algorithmic experiment. Hence, the decision what test instances to use is one of the crucial points in test design (cf. Section 1.1.3). Due to its importance we address the problem in more detail here.

The outline of this section is as follows. Section 1.3.1 contains basic properties that every experimenter should try to accomplish when choosing test instances. We introduce three different types of test instances in Section 1.3.2 and analyze their respective strengths and weaknesses. Section 1.3.3 contains some final suggestions for test data generation.

1.3.1 Properties to Have in Mind

There is a wide agreement in the literature that choosing test instances is a difficult task since any choice of test instances allows for criticism. But there is also a wide agreement on some basic, potentially overlapping, properties that an experimenter should have in mind while selecting test instances. Note that the properties are not only important for the test instance selection, but for the whole experimental process in general. If the selection of test instances helps to achieve the properties, the result is most likely a good set of test instances. We compiled the following list using the corresponding discussions in several articles [4, 9, 10, 22, 27, 28, 30, 32, 39, 41, 45, 46, 50].

Comparability The results of algorithmic experiments should be comparable to other experiments. While this should be taken for granted, there are lots of algorithmic experiments ignoring it.

If different tests in a paper use test instances with different characteristics, it is mostly not valid to compare the measurements. There may be some occasional exceptions, but more often the comparisons are meaningless and cannot reveal anything. However, *Comparability* should not only hold in one paper, it is also desirable for experiments from different authors.

Today, the standard solution to assure *Comparability* is to make the instances or their generator programs available on the Internet. To ensure that other researchers can use the instances it is advisable to use a widely accepted format to store them. If the publicly availabe instances are included in new experiments, they ensure comparability to already published studies using the same instances. The potential abuse that other researchers might optimize their algorithms exactly for these instances is made harder if the experiments include lots of varied enough instances (cf. *Quantity* and *Variety* below).

In the phase of test design, *Comparability* means to be aware of standard test data libraries (cf. Section 1.4) and instances used in former experiments on similar algorithms.

Measurability For heuristics it is often tested how far from an optimal solution the heuristic's solution of a test instance is. Hence, it is desirable to be able to measure the optimal solution in advance.

Unfortunately, nontrivial instances with known solutions are often very small or too much effort must be spent on measuring an optimal solution, e. g., for NP-hard problems. However, problem generators can construct artificial instances with a built-in optimal solution that is known in advance [35]. But one has to be aware that such a generation process may yield quite unrealistic problem instances (cf. the discussion on artificial instances in Section 1.3.2). Furthermore, for NP-hard problems it is very unlikely to be able to efficiently generate meaningful instances with known solutions [54].

Portability In the early days of algorithmic experiments the large and bulky data of some non-trivial examples caused *Portability* problems. Such instances were too large to be published in journals. Researchers could only obtain them by depending on the cooperation of others that had previously used the same instances.

Today, with the availability of many instances on the Internet there are two main sources of *Portability* problems. One arises when proprietary considerations preclude the supply of the test instances. One should try to exclude such proprietary test instances from experiments since they clearly degrade

the above mentioned *Comparability*. However, there are also circumstances, e. g., in VLSI-design, where it is impossible not to use proprietary instances in the experiment.
Another possible source of *Portability* problems is the format in which the instances are stored. Using a widely accepted or some standard format helps to exchange instances with other researchers. The main reason is that such a standard usually is well-documented and everyone knows how to decode it. There already exist common standards for some areas, like the cnf-format for SAT-instances (cf. Section 1.4.4). These standards are mostly a special `ASCII` or even binary encoding of the instances.
If there is no common format at hand, the experimenter has to choose one considering some important points. First, the instances have to be stored in a way such that everyone can convert them quite simply to another format. This means that the format itself has to be documented by the inventer. Furthermore, the format should avoid redundancies, it should be extensible, there should be an efficient decoding routine, and storing the instances should not need too much memory. In some situations (cf. the CSPLib in Section 1.4.4), even a human readable format may have advantages. Another option is to use `XML`. But keep in mind that `XML` is not designed for the purpose of storing test instances. Hence, usage of `XML` as an instance format is really rare up to now.

Purpose Some studies do not explicitly consider the *Purpose* of the experiment when choosing test instances. An example would be to keep in mind whether the experiment should show the potential of an algorithm (where lots of different instances are needed) or just the practicality of an algorithm in specific situations (where more restricted instances have to be chosen). The used test instances should always match the *Purpose* of the experiment.

Quantity The number of test instances to use depends on the goals of the experiment. Preliminary testing to show feasibility requires only a small number of instances. However, the experiments we have in mind are of another kind. To assess strengths and weaknesses of an algorithm or to compare it against other approaches requires large-scale testing in terms of the number of instances used. Choosing many instances helps to protect being fooled by peculiar experiences with few instances and yields more informative studies. Unfortunately, in many studies the set of test instances is too small compared to the total range of potential instances. Hence, the drawn conclusions tend to be meaningless.

Reproducibility When algorithmic experiments are reported, the test instances have to be given in enough detail that another researcher could at least in principle reproduce the results. If the instances were obtained by using a generator, it usually suffices to give the settings of the important parameters of the generator and the seed of a potentially used random number generator. Nevertheless, generated instances with unique properties that are difficult to reproduce should be given as precise instances. This corresponds with *Comparability* from above since making the instances publicly available also supports *Reproducibility*.

Although *Reproducibility* is widely acknowledged to be important, a lot of published experiments are not really reproducible. One of the main reasons, besides the ignorance of some experimenters, might be the problem of proprietary test instances not available to the public.

Significance To ensure *Significance* of an experiment, instances from widely accepted test data libraries should be included, which also corresponds to *Comparability*. Of further interest are instances that test the limits of the algorithm or even cause it to fail. Too easy instances reveal little on an algorithms behavior on hard instances. Hence, one key to ensure *Significance* is the *Quantity* and *Variety* of the test instances. However, it is a challenge to generate meaningful test instances, especially for the assessment of heuristics. The experimenter often has to trade off the need for the sample of instances to be representative and the cost of obtaining the instances.

Unbiasedness Unintended biases should not be introduced into the test instances used. One such example would be to use only instances that the tested algorithm can easily solve. The potential conclusion that the algorithm solves all instances very fast is heavily biased by the choice of instances. Hence, observance of *Quantity*, *Significance* and *Variety* helps to be unbiased in the problem selection.

Another source of biases can be encountered in the generation process of potentially used artificial instances (cf. Section 1.3.2).

Variety The test instances used in an experiment should have different characteristics to show how algorithmic performance is affected. But in many studies the instances have been too simple and too limited in scope, e. g., when only few instances of small size are used to demonstrate sometimes pathological algorithmic behavior. Instances that are too easy do not allow for good conclusions. Large-scale testing, in terms of the range of the instance sizes and the variety of instance properties, is required since this is the only way to reflect the diversity of factors that could be encountered. Demonstrating the potential and the usefulness of an algorithm also requires a wide *Variety* of test instances since otherwise the strengths and weaknesses cannot be assessed accordingly. However, if the scope is to show practicality in specific situations, much more restrictive sets are allowed. Again, the above mentioned *Purpose* of the experiment is crucial for the decision.

1.3.2 Three Types of Test Instances

Roughly, there are three different types of test instances an experimenter could use. Namely, these are real-world instances, artificial instances, and perturbed real-world instances. In this section we assess their respective strengths and weaknesses according to the properties discussed in Section 1.3.1.

Real-World Instances Real-world instances originate from real applications. In this way they are actual examples and mostly reflect the ultimate *Purpose* of any tested algorithm. Several authors have already discussed the usage of real-world instances [4, 9, 10, 19, 25, 22, 30, 32, 45, 50]. In the following, we give a brief survey of their observations.

The property of being representative for real-world behavior is one of the main reasons that real-world instances should be used in algorithmic experiments whenever possible. Very often, the goal of an algorithmic experiment is to evaluate practical usefulness. Then, real-world instances cannot be excluded from the experiment since they allow an accurate assessment of the practical usefulness of any tested algorithm.

However, in the early days of algorithmic experiments real-world instances usually where handpicked. Hence, the collection and documentation was quite expensive. Today these problems do not carry that much weight since most test data libraries (cf. Section 1.4) already include real-world instances and are easily available on the Internet. Real-world instances that are used in well-documented experiments usually make their way into such a library. Nevertheless, real-world instances may have a proprietary nature and thus may not be available for public use. This causes *Comparability* problems.

Another more serious problem with the usage of real-world instances is that it is often difficult or even impossible to obtain a sufficient number of large enough instances. Small instances can be solved too fast on current machines such that the running times shrink to negligibility. Hence, there often are troubles in achieving *Quantity* and *Variety* just using real-world instances. But even if real-world instances would be available in large enough *Quantity* and *Variety* they usually to not allow to draw general conclusions about how an algorithm operates. The main reason is that typically instance properties cannot be isolated in real-world instances. However, this is necessary to show how changing several properties affects the performance. Before we describe a possible way to overcome these issues by using artificial instances, we close with a short summary of the main advantages and disadvantages of real-world instances.

Advantages

– representative of real-world behavior (*Purpose*)
– allow assessment of practical usefulness

Disadvantages

– only of bounded size (*Variety*)
– only few available (*Quantity*)
– sometimes proprietary (*Comparability*)
– lack of control of characteristics

Artificial Instances Usually, artificial instances are randomly generated by a generator program given a list of parameters. One of the earliest examples is NETGEN which generates network problem instances [33]. Using artificial instances is a possibility to overcome the main disadvantages of real-world instances. Hence, they were already studied by other authors [4, 9, 10, 21, 22, 25, 28, 30, 32, 39, 45, 48, 50]. We compiled our following discussion from these papers.

The usage of generator programs ensures the fast and cheap availability of a very large *Quantity* of instances. If the generator program is well-written, one key property is that it can provide arbitrarily large instances which assists *Va-*

riety. This allows the experimenter to determine the size of a biggest instance that can be solved in reasonable time. If the generator program is written for machine independence and the parameters affecting the generation process are well-documented, they provide an effective means to ensure *Reproducibility* and *Comparability*. Good generator code often becomes a standard and can be found in existing test data libraries (cf. Section 1.4). Generators usually are not proprietary.

A good generator program allows the experimenter to control instance characteristics through adjusting parameters that affect the generation process. Thus, instance properties can be isolated and their effect on the algorithm's performance can be estimated. However, generator programs may be biased in the way that unintended correlations are built into the instances or that only instances with particular characteristics are produced. Hall and Poser analyze existing generation processes for machine scheduling problems and state that some widely used approaches actually are biased in such a way [22]. Further ressources of biases may be rounding problems in the generation process or the used random number generator, e. g., when only the last bits are examined [16]. L'Ecuyer gives some useful hints on the usage of random number generators [38].

When evaluating heuristics or approximation algorithms for intractable problems, an important value is the quality of the found solution (cf. Section 1.2.2). It would be nice to know the value of an optimal solution in advance. Such a feature is offered by some generators. They are able to produce instances with a known optimal solution that is concealed from the tested algorithms. However, restricting the tests only to instances with known solutions is likely to yield unconvincing results. One reason is that instances with a built-in solution often have a narrow and very artificial nature and thus are not representative. Furthermore, instances with known solutions do not constitute the primary goal of heuristics or approximation algorithms that are designed to handle cases where an optimum is unknown and too hard to find.

Being not representative for real-world behavior is one of the main points artificial instances are often criticized for. The argument is that artificial instances can only be meaningful if there is some evidence that they can predict the algorithm's behavior on real-world instances. In fact, very often artificial instances do not resemble real-world behavior and thus drawn conclusions may not be valid for real-world instances. An example is the frequent assumption of a uniform distribution to select artificial instances from an instance population. Of course, there are studies where it can be well justified to generate the instances uniformly at random, e. g., when comparing sorting algorithms on integer sequences [36]. But in most cases real-world instances are not distributed uniformly. Real-world instances include structure, and it is certainly true that unstructured artificial instances tell us little about real-world performance. As an example Johnson states that asymmetric TSP-instances, with independently chosen distance matrix entries from small ranges, often are particularly easy to solve [32]. Algorithms succeeding on them may dramatically fail on more realistic instances. However, there are also examples where it seems impossible to build realistic models. For example, McGeoch points out that network flow problems occur in so many areas that it is very difficult to cover all of them in a generator program [41].

Nevertheless, there are some definitive advantages when using artificial instances. A careful design of the generator program can guarantee that at least some properties of realistic instances are met. And very often this is the best one can hope for since there is one main difficulty when trying to generate perfectly realistic artificial instances. It requires very sophisticated analyses to identify appropriate parameters and corresponding values that would lead to realistic artificial instances. In fact, hardly any generator really can produce realistic data. But the ability to control the instance characteristics should not be underestimated when considering the main advantages and disadvantages of artificial instances.

Advantages

– arbitrary size available (*Variety*)
– arbitrary number available (*Quantity*)
– rarely proprietary (*Comparability*)
– ability to control characteristics

Disadvantages

– lack of realism (*Purpose*)
– difficult to assess real-world performance (*Purpose*)
– susceptible to unintended correlations and biases (*Unbiasedness*)

Perturbed Real-World Instances When comparing the lists of advantages and disadvantages of real-world and artificial instances the observation is that they are inversions of each other. Several researchers suggest a way to try to combine the advantages of both real-world and artificial instances [21, 25, 48, 50, 57]. The following discussion is based on these papers.

Starting from real-world instances, a controlled variation by a generator program yields perturbed real-world instances. Such instances are a compromise of real-world and artificial instances.

Due to the number of perturbed real-world instances that can be obtained from one real-world instance, their available *Quantity* is better than for real-world instances. However, the size of the perturbed instances may not differ dramatically from the size of the original real-world instance, and the inherent structure of the real-world instances nearly stays the same. Hence, perturbed real-world instances cannot support *Variety* in the way artificial instances can.

Another problem is the ability to resemble real-world behavior. On the one hand, perturbed real-world instances cannot be seen as actual real-world instances due to the perturbation. But on the other hand, they retain much of the inherent structure. This causes their degree of realism to fall somewhere inbetween real-world and artificial instances.

However, the most serious problem with perturbed real-world instances is that it is often very hard to identify interesting parameters that have to be changed to obtain useful perturbed real-world instances from given real-world instances. This difficulty cuts back the benefits perturbed real-world instances have. For this reason, Shier suggests to use a meta-algorithm that, given an algorithm and a test instance, would generate interesting perturbed instances [57]. But this still seems to be dreams of the future.

We conclude with a short summary of the advantages and disadvantages of perturbed real-world instances.

Advantages	**Disadvantages**
– better *Quantity* than real-world instances	– *Variety* comparable to real-world instances
– more realistic than artificial instances	– less realistic than real-world instances
	– often hard to identify meaningful perturbation

1.3.3 What Instances to Use

There is no proven right way for the choice of good test instances, and the debate which instances to use continues. Since the choice of test instances usually is limited by time and space restrictions, some tradeoff will always have to be made. We can derive some helpful suggestions from our above discussion and the references therein.

To ensure the *Comparability* with previous studies standard test sets and generators should be used whenever possible. Proprietary instances should only be used under special circumstances and with adequate justification. There may be scenarios where it is impossible not to use proprietary instances, but in most cases the lack of *Comparability* is too large compared to their usefulness.

We have seen that real-world instances and artificial instances are quite contrary in their advantages and disadvantages. The inclusion of real-world instances enables the assessment of the practical usefulness of any tested algorithm. However, since another goal should be to test against a *Variety* of instances, the only way is to additionally include artificial instances. They may rarely be realistic enough to completely substitute the real-world instances, but their most important advantages are their nearly infinite variety, and that the experimenter may isolate special instance properties that directly affect performance. Very often structures can be found in the artificial instances that allow a careful comparison to the real-world instances and thus enable the experimenter to evaluate the predictive quality of the random results.

Very often instances are chosen on which the tested algorithm performs well or it is easy to demonstrate improvement over previous algorithms. But additionally, there should be always included test instances where the tested algorithm is likely to perform poorly. This enables the judgment of potential weaknesses which indicates *Significance* and *Unbiasedness*.

Altogether, our suggestion is to use at least real-world *and* artificial instances. Finding meaningful perturbed real-world instances often is too hard compared to their benefits. It mostly suffices to use artificial instances in addition to real-world instances. Before choosing instances the experimenter should be aware of the practice in the corresponding field to know which instances were used in past experiments. However, the *Purpose* of the experiment should determine the final choice of instances.

As a last remark, we point out that regardless which instances are used in an experiment, they have to be conscientiously documented and made available to the public to ensure *Comparability* and *Reproducibility*. Furthermore, a good talk on algorithmic experiments states what instances were used [44].

1.4 Test Data Libraries

In Section 1.3.1 we have seen that, in order to ensure *Comparability* and *Reproducibility* of algorithmic experiments, it is essential to make the test instances or their generator programs with the corresponding parameter settings publicly available. A convenient way is the usage of test data libraries. These are collections of test instances and generators focused more on a single problem, like the library for the satisfiability problem (cf. Section 1.4.4), or on a set of related problems, like the libraries for constraint solving or theorem proving (cf. Section 1.4.4).

The outline of this section is as follows. Section 1.4.1 contains properties of a perfect library. In Section 1.4.2 we focus on issues relating to the creation of a library, whereas Section 1.4.3 discusses challenges of an already established library. Section 1.4.4 closes with a brief compendium of existing test data libraries that again highlights some of the most important library issues.

1.4.1 Properties of a Perfect Library

Today, very often the test instances used in algorithmic experiments are obtained from test data libraries. Thus, it is important to be able to identify good libraries. We provide some properties that are characteristic for the quality of test data libraries. Our following alphabetical list is based on discussions by several authors [4, 19, 21, 17, 18, 28, 22, 29, 51, 60].

Accuracy A library should be as error-free as possible. This applies to the contained instances or generator programs and as well to their documentations or any other included data. An example would be that the instances should have unambiguous names in the library.
Any error found has to be corrected immediately and documented in a kind of "history of changes".

Availability In the days before the Internet came up, the test data libraries were books or available on magnetic tapes that had to be ordered from the maintainers [15, 51]. Nowadays, lots of libraries for nearly any problem are freely available on the Internet. Thus, they are easy to find and accessible for any experimenter.

Completeness A library should be as comprehensive as possible.
On the one hand, this means that libraries should contain all test instances or generators known for the respective problems. Ideally, an experimenter

should not need to look elsewhere to find appropriate test instances. Newly occurring test instances should be included immediately.

But on the other hand, this also means that additional information on the test instances should be included. Such further information might be the best known solution, performance data of algorithms solving the instances, references to studies where the instances were used, pointers to state-of-the-art algorithms, or any useful statistic.

Coverage A library should contain all meaningful instances of the respective problems. Hence, it should be as large as possible. New instances have to be included whenever available (cf. *Extensibility*). However, very large libraries require a sophisticated design to guarantee the *Ease of Use*.

Difficulty There should be contained neither just hard nor just easy problems in a library. Practical problems often differ very much in their difficulty, and this mix should be reflected in the library. Even very easy problems may be useful for first expositions on a newly developed algorithm. For a newcomer difficulty ratings for the instances would ease the choice of appropriate instances.

Note that even for established libraries there may be doubts concerning the *Difficulty*. For example, Holte showed that for the UCI Machine Learning Repository the accuracy of some very simple classification rules compared very favorably with much more complex rules [26]. However, the practical significance of this result depends on whether or not the UCI instances are representative for real-world instances. On the one hand, many UCI instances were drawn from applications and thus should be representative. But on the other hand, many of the instances were taken from studies of machine learning algorithms. Hence, they might be biased since often experimenters only include instances that their algorithms can solve. The conclusion is that only a careful analysis and selection of instances ensures a wide range of *Difficulty* in a library.

Diversity The instances contained in a library should be as diverse as possible. There should be real-world instances from applications, as well as artificial instances that allow the evaluation of the influence of special instance properties on algorithmic performance.

Unfortunately, very often instances from certain applications are favored while others are neglected. The instances in a library should represent instances from all applications where the respective problem might occur.

The more diverse the instances in a library, the easier it is to choose a varied set of instances. This helps to prevent an experimenter from over-fitting algorithms, testing them only on a few instances with similar structure.

Ease of Use A library should be easy to use. This includes obtaining instances from the library but also reporting errors or suggesting new instances. Ideally, there are software tools that might help to convert instances from the library to another format or that support the submission of new instances.

Also the navigation in the library is a crucial point. An experimenter should have no problems finding the instances that match his purpose or realizing that such instances are not contained in the library.

Extensibility Although desirable, it is very unlikely that a library contains all meaningful instances of a problem (*Completeness*). Hence, a library should be extensible, and the addition of new instances should be as easy as possible (cf. *Ease of Use*).

Another aspect of *Extensibility* is not only the inclusion of new instances of the original problem but the addition of related problems, e. g., including SAT-instances in a 3SAT-library.

Independence A library should be as independent as possible from any algorithm solving the corresponding problems. This means that the instances included in the library should be chosen as unbiased as possible, not preferring only instances where solely one algorithm succeeds.

Furthermore, this means that the format in which the instances are represented in the library should not be proprietary to only a few algorithms.

Topicality A library should be as up-to-date as possible. If, for example, the best known solution to an instance or the rating of an instance changes due to new studies, the respective information in the library has to be updated to prevent duplication of results.

1.4.2 The Creation of a Library

The creation of a library involves very different aspects. First of all, a choice of the data format to represent the instances has to be made. For the main points considering the data format, we refer to the corresponding discussion in the *Portability* part of Section 1.3.1.

Ideally, the decision what format to use in a library should not be done by a single person, but rather by the research community. For example, the DIMACS Implementation Challenges [12] have served to establish common data formats for several problems, like the cnf-format for SAT-instances. Nevertheless, even if there is already a commonly accepted format, there might be some arguments for modifying it, e. g., if it does not allow for future extensions, like including new properties. Hence, as early as in the choice of the data format, future *Extensibility* can be assisted.

But *Extensibility* is not the only property from our above list (cf. Section 1.4.1) that has to be considered already in the creation process. Another one is *Completeness*. Ideally, a new library should include all instances that are known to the community so far. This often simultaneously supports *Difficulty* and *Diversity*. But to collect all the known instances the support of the community is essential. Hence, the library project should be advertised as soon as possible to find lots of contributors that provide test instances. The resulting benefit for the community is one single place where all instances can be found. This should attract researchers to cooperate if they get to know the project through formal and informal advertisements in mailing-lists, on conferences, or in journals. But not only active researchers should be encouraged to provide instances. Also industry should be asked to support applications data—possibly breaking their proprietary nature.

Through the collection of the known instances some problems might arise. The library creator must ensure that the instances are as unbiased as possible. Including only instances from published algorithmic studies might favor feasible instances since most studies only include data that shows how well the studied algorithm performs. Unfortunately, such instances that have already been solved by some algorithm have a selective advantage in a couple of libraries. To protect against such a narrow selection that would influence *Independence*, also existing generator programs have to be included. They can provide lots of instances that are new to almost every algorithm.

The collected instances have to be carefully transformed to the data format of the library to prevent from errors (*Accuracy*). This could be the birth of a tool that is able to convert the instances from one format to another. Such a tool should be included in the library as well (*Ease of Use*).

And last but not least, the *Availability* of the finished library is a crucial point. Today there is a very convenient solution—the Internet. Creating a website for the library not only enables almost everyone to access the library, but also allows to use the features of the Internet, like creating hyperlinks to papers that use the instances from the library. A welcome page of the library could, for example, contain links to the instance and generator program pages, links to descriptions of the problems, and a link to a technical manual describing the library and its data format. All the individual pages of the library should have a common layout to support a consistent representation and the *Ease of Use*.

However, with using the Internet there might occur a problem when having the library at only one server. As Johnson pointed out: “Never trust a website to remain readable indefinitely” [32]. For the library creator, this means that at least one mirror-site of the library on a different server should be created to protect against unavailability.

1.4.3 Maintenance and Update of a Library

After the creation of a library the work is in no way finished. We might just as well argue that it even has started.

Hardware and algorithms become faster, and thus instances might become too easy. For future generations of hardware and algorithms, the typical and demanding problem instances change. This means that a static library would quickly become obsolete since it cannot represent such changes. But besides the major changes, like including new useful instances (*Extensibility*) or revising the data format, there are still other minor activities for a created library. Based on our list of properties from Section 1.4.1, reported errors have to be corrected (*Accuracy*) and new results for the library instances or pointers to new studies should be included (*Topicality*). Hence, a good library has to be continuously maintained and updated. A history of the changes to the library may support the *Ease of Use*.

The responsibility for all that work should be shared by several persons. We refer to them as maintainers of the library. Ideally, the maintainers themselves are active researchers in the library’s field. This allows them to assess the significance

Library	Short Description
CATS	combinatorial optimization and discrete algorithms
CSPLib	constraint satisfaction problems
FAP web	frequency assignment problems
GraphDB	exchange and archive system for graphs
MIPLIB	real-world mixed integer programs
OR-Library	operations research problems
PackLib^2	packing problems
PSPLIB	project scheduling problems
QAPLIB	quadratic assignment problem
QBFLIB	satisfiability of quantified Boolean formulas
SATLIB	satisfiability of Boolean formulas
SNDlib	survivable fixed telecommunication network design
SteinLib	Steiner tree problems in graphs
TPTP	thousands of problems for theorem provers
TSPLIB	traveling salesman and related problems

Table 1.1. Examples of established test data libraries

of submitted instances and to keep track of current trends in the community. However, this also means that there has to be sufficient financial support enabling the maintainers to spend part of their time on the library. Maintaining a library should also be credited by the community, like being on the editorial board of a journal.

Nevertheless, the work should not only be done by a few maintainers. The community as a whole is asked to provide new useful instances, point to new interesting results, and report errors. Contributing researchers should be acknowledged for their suggestions in the library. If the support of the community is missing, a library project might even fail. Lots of researchers should be encouraged to use the library. Thereby, they will most likely become active contributors. This again emphasizes that the role of a broad advertisement of the library cannot be overestimated.

Altogether, a library can be seen as kind of an ongoing open source software project. New stable versions have to be provided in regular intervals by some responsible maintainers, assisted by a community of volunteers.

1.4.4 Examples of Existing Libraries

There is a wide variety of existing test data libraries. Table 1.1 lists some of them, but is by far not meant to be complete. All of these libraries may be easily found on the Internet by using any search engine. We close our discussion of test data libraries with a more detailed view on four example libraries. Thereby, we summarize the most important issues from the previous sections.

CATS The ambitious library CATS was announced in 1998 [19]. Different pages, each devoted to a specific problem, with a unified layout should be maintained by volunteers using contributions from researchers. But a look at today's state of the library only offers pages for two problems—one on Maximum Flow and the other being a draft devoted to Minimum Spanning Tree. Both pages were not updated in the last years. Unfortunately, it seems that the community did not use and support the CATS library as it would have deserved. But as we pointed out in Sections 1.4.2 and 1.4.3, the support of the community is crucial for the success of a library.

CSPLib The first release of CSPLib stems from March 1999 [17,18]. It contained 14 problems in 5 overlapping areas. Today there are 46 problems from 7 areas. Hence, at first glance the library is not very large—although 46 is only the problem not the instance count.

Since very often the solvability of a constraint satisfaction problem depends on data representation, the library creators decided to be as unprescriptive as possible. The only requirement is that the problems are described using natural language. The main point is that no instances have to be given. Thus, the CSPLib is rather a problem than an instance library, which somehow qualifies our above remark on the library size. On the one hand, using natural language description eases the input of new problems which might encourage researchers to contribute. But on the other hand, the derivation of concrete instances from the specification might be quite cumbersome. Both factors influence the *Ease of Use*, which is an important property for every library.

SATLIB It was established in June 1998 [29]. Different from the above described CSP-situation, there is a widely used and accepted data format for SAT-instances—the cnf format from the Second DIMACS Challenge. As we pointed out in Section 1.4.2 such a widely accepted data format is the basis for a wide usage of the library.

Unfortunately, the current stable version 1.4.4 of SATLIB is more than five years old. The SATLIB-page announces an update since 2003, but as we pointed out in Section 1.4.3 this includes lots of work. This is a downside of maintainers being active researchers. Since they also have to do non-library work, necessary activities concerning the library may take a while. A possible way out might be the engagement of some assistants, when major changes are due. But this would require a broad financial support of the library.

TPTP The library containing instances for evaluating automated theorem provers started in 1993 [60]. It has become a nearly perfect library.

Again, different to the situation for constraint satisfaction problems, widely accepted data formats exist, which are used to represent the instances. Over the years, TPTP continuously grew from 2295 instances from 23 domains in release v1.0.0 to currently 8894 instances from 35 domains in release v3.2.0. Instance files include ratings denoting their difficulty.

The library not only includes many of the instances known to the community, but also generator programs for artificial instances. Thus, it is as comprehensive and diverse as it could be. Due to its *Completeness* and *Diversity*, TPTP served as a basis for lots of CADE ATP System Competitions in the last years.

Concluding, we can state that TPTP is a highly successful and influential library for the field of automated theorem proving. Such an impact should be the main purpose of any library.

1.5 Setting-up and Running the Experiment

The setup of experiments and their execution require a precise plan that describes the steps to be taken, in every science. These phase is located between the design idea, its implementation, and the evaluation of results together with insights gained during one run through the cycle of Algorithm Engineering. First, you have to think about what you want to report on, to find falsifiable hypotheses that should be supported or rejected by experiments. Section 1.2 covers this part. If not yet done, you should then implement the algorithm. For details see Chapter 6. How to come up with a meaningful, big amount of input instances has been discussed in Section 1.3. Before we can evaluate results in Section 1.6 it is needed to set up a well-suited *test-bed* and run the algorithms on the data, a non-trivial task. This section focusses on the difficulties that arise in Algorithm Engineering, that mainly consist of two areas of interest. First, experiments in computer science have been ignored for quite a long time. Only in the recent years, researchers rediscover their strength and possibilities. Some of the hints given in this section address in general computer scientist and aim to encourage them to run experiments. The goal is lead the community to good experimental work, as it is the case in other sciences. Thus, the hints mainly adapt state-of-the-art rules, applied in e. g., in natural sciences, and turns them towards computer science. Second, the cycle of Algorithm Engineering naturally forces experimenters to run similar experiments over and over again. Thus, further remarks recommend how to ease this process, while still being accurate as an experimental science demands. For the sake of better distinction, the *setup-phase* is elaborated first in Section 1.5.1, followed by hints applicable in the *running-phase* mentioned in Section 1.5.2. It is useful to learn about pitfalls of both phases before running any experiment. Section 1.5.3 give additional advice for approximation algorithms and collaborative experiments.

Most pieces of advice originate from the very good overview paper written by Johnson [32] and a crisp collection of “DOs” and “DON’Ts” stated by Gent et al. [16]. We extend them by hints presented in a paper of Moret [45]. Given suggestions and motivations are also influenced by personal experiences. These are gained by experimenting in the area of computational geometry done in the past yeast and those planned for the future. Some analogies to natural sciences are taken into account from personal communications.[5] Most of the given hints

[5] with Peter Leibenguth

and suggestions are not problem-specific, since they can be applied to almost any experiment one can think of. Otherwise, special cases are pointed out. This section mainly collects an important set of high-level hints for an experimenter. For problem-specific experiments, possibly proper and case-specific extensions have to be done. Furthermore, as "DON'Ts" describe prohibitions all pieces of advice are stated in a positive, constructive manner. Several ones might overlap with others, which is due to the complexity of the whole area. Thus, the reader should not be bothered, when reading some statements twice. In contrast, this emphasizes argument's importance and points out existing correlations.

1.5.1 Setup-Phase

A well-organized laboratory is essential for a successful experiment in natural sciences. Often, scientists, in these areas, have to deal with a lot of restrictions or have to experiment outside of the laboratory. Unlike finding their laboratory somewhere, algorithmic scientists have to use computers. The experimenter is faced with the possibility to adjust a huge bunch of parameters. Algorithm Engineering aims for a best choice. Experiments serve to support the chosen decisions or falsify some considered hypothesis. Otherwise, if experiments are set up arbitrarily, their results may loose every meaning. In the following, we collect advice, such that an experimenter can avoid some pitfalls that might occur during the setup of an algorithm experiment.

Use available material! When reinventing the wheel, an experimenter may loose huge amounts of his limited time to finish the experiment. Public repositories or selected requests to other researchers should help to save time. To use available material is suggested. Two reasons exist to do so. First, experiments should be finished as soon as possible, which does not mean to carelessly execute them. Second, results need to be related to exiting set-ups and experiments. For the sake of reusability, the focus lies on available test instances and implementations.

Test sets may be obtainable from internet repositories as presented in Section 1.4 or from the authors' homepages. Some journals support publishing of additional material, so this is another place to look. If there is a standard library of instances, you are always supposed to use this instance library. In case the original test set is not available, but was generated artificially in some way, you should regenerate instances with the same parameters as the original one. This requires a detailed description of the generation process and the parameters used in the original paper.

The same holds for the implementation. Sometimes the source code is publicly available. In other cases the authors may be willing to provide it for further experiments. If you do not have access to the source code, you should implement the algorithm yourself, taking into account the implementation details that were reported. A new implementation in your computing environment or recompiling available source code is clearly preferable to make the old results comparable to yours [4]. However, a new implementation may be infeasible, for instance, because the algorithm is too complex or important details of it are unknown (e. g., of an commercial implementation). In this case you have to stick to some reasonably good implementation.

Once an implementation and the original or similar test sets are available, it is a good idea to try to reproduce the original results qualitatively. In particular, this is running the experiments on the test set, measuring the necessary quantities and checking whether the data is consistent with the claims in the original publication. A discrepancy is worth pursuing, usually indicating a flaw in the implementation or test setup.

To enable other researchers to participate in the process of Algorithm Engineering, is is recommended to publish as most as possible. At its best, it is advised to provide source code and full data sets.

Ensure you use reasonably efficient implementations! An efficient implementation is the most fundamental part for the experimentation procedure. Johnson [32] states three major advantages.

– Allow to support claims of practicality and competitiveness.
– Results of inefficient implementations are useless, since they most probably change the picture one would actually expect from implementation in practice.
– Allow to perform experiments on more and respectively larger instances, or to finish the study more quickly.

Chapter 6 already discusses the implementation task with all its aspects. A main source of information how to reach efficient code is contained in Section 6.3.1 that describes tuning techniques. Some researchers forget, ignore or do not have time to implement known speed-up techniques, but still state that their implementation would be competitive to the ones using speed-ups when implementing these tricks for their own algorithms. But this argument lacks plausibility. It is completely unknown, whether certain tricks make sense for any other algorithm, and if so, in how far they actually improve the running time. Similar arguments hold for other performance measures.

Compilation In order to get efficient code, it is important to choose the right combination given by the platform. Note that the programming language matters as much as the compiler and its options. In general, you should take a comparable environment. If you only benchmark your algorithm, then the programming language is quite unimportant, but in case you compare with others, even if only copying their running times, it is advised to use roughly the same technology for the implementation. If existing experiments are implemented using C++, coding your algorithm in `Java` will make a comparison very difficult.

To avoid unnecessary slowdowns, you should always run compiled code instead of interpreted code. For sure, compiling code is a science in itself, but basic rules can be stated here, too. When you compile code for experiments which include timing, switch off all debugging and sanity checks in your code. A print-statement usually takes a huge amount of time, and pre-, post- and assert-conditions also only slow down the computation time. They only aim for the correctness of your code during runtime, while especially for non-trivial routines, they might influence the worst-case running time. Therefore, failures in these conditions on your test data indicate bugs. Testing your code on the data with active conditions is required, but for generating publishable performance measures and also to distribute your software, remember to deactivate them. If now new errors or crashes appear, you can be quite sure, that a sanity check contains a side-effect which should be definitely avoided.

Deactivating debug code and compiling in optimized fashion also holds for supporting libraries used in the implementation of your algorithm(s). Commonly, these supporting libraries are used for subroutines or atomic functionality. Usually, they are called quite often and you have to ensure you select the right implementation. An infamous example is using a $\Theta(n^2)$ sorting routine. Especially when you use experiments to approximate the asymptotic running time of a theoretically unanalyzed algorithm, such an influence is without doubt. Even if it is the case that you have chosen the theoretical best-known algorithm for a subroutine, big constant factors, that play an important role in implementations, will influence the algorithms performances dramatically and may destroy competitiveness of the implementation of your algorithm. Note that in experiments we always compare implementation of algorithms only and not their theoretical behavior.

Coding Section 6.3 already suggests not to spent too much time for fine-tuning. The bottom-line is to produce reasonable efficient code in a reasonable amount of time. Code documentation as explained in Section 6.5 is also demanded. It serves to remember details of the implementation and helps others to understand your software, especially when published under an open source licence. Published software enables other researchers to run your experiments on their own machine, maybe slightly modified due to new algorithmic ideas. Furthermore, they possibly submit bugs to you.

Section 6.2 has covered techniques to avoid bugs. Some bugs should be fixed when entering the experimental phase. Namely, the bugs that make the software crash, and bugs that lead to a wrong output. But there are also bugs that negatively influence the performance of the algorithm. To find such bugs bears out as a non-trivial task and experiments seem to be the main technique to detect such hidden errors. They will never show up voluntarily, you have to search for them, which needs some indication. Otherwise, you just believe that the performances are already optimal. The only chance to find them is some deviation in the results. Profiling your code gives a very good overview which subroutines are called very often, and which consume a lot of time. Unfortunately, some bugs cannot be detected by a profiler, e. g., filter failures as mentioned in Section 6.3.1. Either you design and prove theoretically the lack of such failures, or you have to implement a testing layer in between the high-level parts of the algorithm and its subroutines, to see whether equal objects are not identified. Both methods are rather disappointing and success is not guaranteed. If you do not believe in bugs that destroy the performance of an algorithm look at this example. Consider an implementation of quick-sort whose choice of the pivot element is *not random* due to some wrong variable usage, e. g., by accident. Whenever your algorithm needs to sort some *structured* containers, quick-sort suddenly performs at its worst-case running time of $O(n^2)$.

Systematic Errors It seems that systematic errors during experiments on algorithms can be avoided from scratch, while, for example, physicists face uncertainties in there measurement devices or are unable to measure at the actual point of interest. Unfortunately, these appearances are deceptive. At a first glance, nearly everything seems to be under control, but you have to make sure that you have control of the right points. A wrong position of a timer, e. g., in the innermost loop, changes the whole performance of an algorithm dramatically. This example comes along with having too many timers in the code. You should always scrutinize whether your decisions make sense in your setting and whether your measuring methods keep the experiment free of bad influences.

Check your input data! Before running the actual experiments ensure you use correct input data. In general, if your input data set covers a significant part of the allowed input space for your algorithm, you are doing the right thing. Section 1.3.1 discusses in detail what needs to be considered to find a set of instances with enough variety, and Section 1.3.2 deals with advantages and disadvantages of artificial and real-world data. A useful set of instances contains a balanced mixture of both.

Your data sets might be corrupted or faulty generated. Real-world data are often corrupted or need special pre-processing. Buggy generators may create artificial data that do not produce the desired sets. Careful experimentation checks the appropriateness of such data before running time-consuming algorithms. As explained in Section 1.3.2 non-random numbers might also bias the data generated sets. In general, each single data instance should be free of redundancies, for example, the same points twice when computing the convex hull of points. Otherwise, you only check the caching strategy rather than running the algorithm on a bigger instance. Of course, it is useful to see whether an implementation handles redundancies optimally, but better check this performance with its own experiment.

Data sets might also be to simple. This mainly addresses input data that are processed within a fraction of a second. Instances should be chosen in a way that measurement's noise does not affect the results. Hereby, noise denotes points which affect running time in general independent of the specific algorithm which should be tested. Today's computers consists of several units, e. g., pipeline, register, memory hierarchy. Hence, it takes some setup time until all operational units of the computer fully work on executing an algorithm. This is denoted by measurements noise, which barely can be ignored if your input instance runs only for some milliseconds. On current machines, a good advice is to use input data that run at least a second. The situation might be different, when aiming for counts only or when dealing with real-time computations.

Use different platforms! The best implementation is only as good as the supporting environment. This means that the environment plays an important role in setting up an experiment. In computer science, the performance of an implemented algorithm crucially depends on the used hard- and software. The following questions need to be answered. Which processor is used? How much memory is available? How is memory hierarchy organized? How many registers are available? What is the underlying operation system? Which compiler was used, additionally given chosen compiler options? Which supporting libraries are utilized? Obviously, you cannot test your algorithm for all possible combinations of hard- and software, but restricting yourself to implement it only in one specific environment may change the picture. Beyond, it may lead to hypothesizing non-existing conclusions of the data due to specific behavior in the external environment, e. g., a special caching strategy of the operating system which influences the movement of data.

It is strongly recommended to test algorithms at least on a small set of different architectures and with different compilers. A good balance between different environments and fine-tuning of code for each one should be found. Running the same experiments on different platforms helps to avoid to draw the wrong conclusions. Whenever these implementations show surprising differences in their performance measure, it is a must to ask why and to find the answer. These differences show whether the implementation in the environments are free of dependencies to the setup and therefore allow to draw setup-independent conclusions.

Aiming for comparability of experiments, the calibration of the machine(s) is a lot more useful than just stating the architecture and the speed of your processor. Architectures change within a couple of years dramatically, which makes it difficult to relate new results to old ones then. Calibrating the machine means to compile and run in your setup a small piece of code that is publicly available and known as well as accepted in the community. Its output states more about the problem-specific performance of the machine than the CPU speed does. Future researchers can then adapt their machines the same way which enables them to normalize the old results to their own new results. There is a chance that this normalization fails. But in most cases it is much more valuable than normalizing to the pure CPU speed and obviously better than forgetting about it.

Use appropriate formats! Section 1.3.1 already discusses the need to carefully select the format for input instances. On the output side of the algorithm we will see a bunch of results, at least the measures selected during the design of the experiment, e. g., running time. See Section 1.2.2 for more details. If we are purely interested in the primary measure(s), we can just print this information to the console. But thinking a little bit further, it can be seen that it is really useful to have self-documenting programs. Consider a question of a reviewer that comes months after you actually run the experiments or you want to do a follow-up study. In both cases you have to remember most of the old results, and expect that your personal memory might forget most of these things. Therefore, it is strongly recommended to create some self-explaining output format for each run that collects all relevant data, which, for example, consists of the following list.

- Main measures, like CPU time, solution quality, and memory usage.
- Algorithm data, like the name and version of the algorithm, its parameter settings.
- Meta data, like the date, the name of the instance.
- Setup data describing the used machine, memory hierarchy, used compiler and its flags.
- Supplemental measures that can be useful when evaluating the data in the future, like intermediate values, operation times or simple counts of operation calls.

Especially, when computing the additional data, one should avoid to harm the overall performance of the algorithm. If an additional value can be stated without extra costs, it would be careless to omit it, since otherwise, you need to rerun the experiment to get its data, which would be really expensive. A sophisticated design of the experiment that checks which data should be outputted before starting the actual running phase is strongly advised.

The used output format should be of clear and effective syntax. Furthermore, avoid using abbreviations, since every value that cannot be interpreted correctly in the future is useless. `XML` might be a good candidate as format choice, but one should definitively check whether it fits all needs while staying simple enough.

We want to go even further and enforce every setup to combine input instances, implementations, and results into a common framework.

Do use version control! So far, we have learned that the setup for an empirical study is far away from naively implementing some small environment. In contrast, most experiments start with some initial setup and are constantly evolving. They become larger and more complex, e. g., new algorithms are being added, methods are changing, additional instances should be tested and bugs will be fixed. In summary, this is a perfect setting for a version control system like *Concurrent Version System* or *Subversion* that have already been recommended for the implementation of an algorithm in Section 6.6.5. A version control system allows to store snapshots of the current system in a common repository, which can be also accessed and fed by a group of developers.

Putting all changes of your setup constantly on a repository provides several advantages:

- It allows you to go back in time by checking out old versions, it *ensures reproducibility*. You are able to rerun all your experiments.

– It also provides tools to compare two versions, which offers the possibility to check which changes result in better or worse algorithmic performance.
– Not using a version control system is a quick way to loose control over different versions of your environment. You may store your files within time-stamped directories. But then, fixing a bug in one version, while improving a heuristic in another one will quite surely lead to a third version, which contains the bug again. Version control cares for these changes. Thus, a fixed bug cannot appear again in the future as it might happen when human beings maintain different copies of a file.
– Storing subsidiary data close to the executables of the experiment is much better than maintaining a bunch of files, or even to use your personal memory that might be more forgetful than everyone hopes for.

Obviously, the version control on a central server only makes sense, when the repository is under control of a reliable backup system. Otherwise, you may lose your complete work which contradicts the aim of reproducibility. Assuming that version control systems are error free may lead to useless experiments, too. Although system are quite matured, having an eye on its operation, e. g., whether `diff` works fine and versions are properly stored, is good advice.

Use scripting! As we know, Algorithm Engineering, and especially experimentation, consists of an iterative procedure to progress and to reach publishable results. Several tasks have to be performed a repeated number of times or similar jobs should be controlled over and over again. Instead of starting each single run manually, it is advised to analyze the structure of the experimentation in detail. Its evaluation will lead to a bunch of scripts and proper pipelining. At its best, it suffices to only press *the red button*. In the end, processed data are collected and may be already presented in figures which are a fundamental help in data evaluation. Scripting and version control are fundamental partners. With such a setup, the researcher can concentrate on developing algorithms and selecting or generating instances while the actual experiments run automatically, maybe scheduled at regular times on your machine or during night.

With `ExpLab` [23] a set of tools is provided that collect the mentioned parts out-of-the-box. It offers scripts that allow to set up and run computational experiments, while also automatically documenting the environment in which an experiment is run. Assuming that the same environment is still available, it allows to easily rerun the experiments and to have a more accurate comparison of computational results. Finally, `ExpLab` provides text output processing tools that help dramatically to eliminate some tasks needed for collecting and analyzing the output. Its overall goal is to augment existing tools to reach a comfortable experimentation environment. Unfortunately, its development has been stopped and it is built on top of `cvs` instead of `svn`.

1.5.2 Running-Phase

Once the laboratory is prepared and set up, it is time to start the experiment. During setup you tried to exclude all environmental errors for the experiment.

But obviously, while running an experiment further errors can be made. In a natural science experiment, wrong timing may destroy the whole result. Furthermore, forgetting to write down parameters disposes you of the possibility to publish any valid result. In some way, these strict rules seem to be forgotten when publishing experimental results in computer science, especially performance measures of algorithms. In contrast, reviewers would be very happy to get informed about the main facts the experiment was run with. Therefore, applying adapted methodology from natural science to your experimental running phase prevents you from having no answer to questions asked by colleagues or, even worse, reviewers. It is recommended to check in how far the following hints should already be considered during the setup, although they address directly the running-phase.

Keep notes! Each experiment in natural sciences is only valid, if any other similarly equipped laboratory can reproduce the same result. That requires a detailed description of what you did and what you found out. In computer science this should also apply.

During setup you already decided, which data will be collected and how to store them. The claimed hypotheses also prompt you to combine algorithms with data instances. A script lets them run and produces a vast amount of output. Ensure that this output is also accessible in the future, which means to put them under version control, too. In case you get asked you can present all details, or you can also test other instances in the future and relate them quite accurately to your original results.

Additionally, you should remember to write down and store all good and bad conditions of your algorithm. You might rely on your personal memory, but to be sure, it is a better idea to store them explicitly. Furthermore, consider the possibility of handing over the laboratory to a colleague. He can only build on the content stored in the repository since he has no direct access to your memory.

Change of Factors! Some algorithms can be fine-tuned by one or several external parameters, also known as *factors* as explained in Section 1.2.3. The actual behavior of a heuristic or the overall algorithm can be influenced. For an experimental setup it is necessary that the parameters are either completely fixed and reported or they purely depend on the data given in the instances to be computed. In the first case, reproducibility forces to assign some values. Usually, it would be very interesting to know how the algorithm behaves with other settings. If not fixing parameters, most people experiment with the settings to find out which combination leads to the best algorithm's performance. Due to this fact, before determining the parameters, an algorithm is actually not properly specified. Only by searching for the proper values experimentally, the algorithm will be finally determined. But this may result in having different algorithms for different instances. In contrast, we also want to encourage you to experiment in certain boundaries with the parameters, since these might lead to unexpected good performances.

To sum up, if you use different parameter settings, where each applies to a set of different instances, the choices must be well-defined or should be determined algorithmically from the instances. You also need to report and describe the adjustments in all details, as well as the running times spent to find the optimal parameter settings in your publication.

Change only one thing at a time! This advice is closely related to the preceding one, as well to the planning phase of the experimental work. While the first one deals with the parameter settings of an algorithm, the latter one must be considered when testing different instances. Thus, to reach reliable results it forbids to vary more than one parameter from one run to the next. In the example of different instances this means that you either change the size of the input, its complexity, or you chose another type of variation. If you need to change parameters of your algorithm, also make sure that you only tune one parameter at the same time.

This rule originates from natural sciences, where you also change, for instance, either temperature or pressure, but never both at the same time. Obviously, it forbids changing the type of the instance in combination with tuning some parameters. Especially in this case you get performance values that never mark valuable comparison results.

Run it often enough and with appropriate data sets! Once everything is set up, relying on a single run may lead to conclusions without value. Each proposed claim should be supported by a set of independent runs. You also need a significant amount of runs to reduce the influence of external factors, i. e., to average and probably get rid of the noise. Especially randomly generated data might have a big variety, even when they originate from the same generator. If you want to check the performance of several algorithms on randomly generated data, it is a good choice to use the same set of instances for all algorithms instead of generating them independently for each run.

It is recommended [32, 16] to look at as large instances as possible. First, this gives a better indication of the asymptotic behavior of the running time or the approximation gap of your algorithms. Second, important aspects and effects may only occur at large instance sizes due to some boundary conditions, e. g., caching effects. Looking at huge instance sizes also strengthens your claims, especially when the instances are bigger than the ones you expect in practical environments.

If you are using scripts or the tools proposed in Section 1.5.1 it should be easy to set up a powerful and automatic schedule which can run during the night and present you a list of results the next morning, depending on the algorithm(s) and data sets.

Look at the results! This sounds like an obvious piece of advice, but it is a crucial one. First of all, check whether the actual output, the result the algorithm is implemented for, is correct. If not, you have to search for bugs. If the results are correct, check whether the global picture is consistent. If you are in the lucky position, that your scripts have produced some plots automatically, these pictures help to find out whether the algorithm(s) on the checked instances behave smoothly. Either you will see a picture, as you might have expected, which then supports your claimed hypotheses, or, in contrast, some anomalies occur, e. g., exceptional high or low running time for a specific instance or family of data sets. Have a closer look at them and explain why they behave differently. An origin might be a bug. Or you will detect that this behavior is intrinsic to the algorithm, because this certain family of data always forces the algorithm to compute it that special way.

In Section 1.6 we explain how to evaluate the performance values more detailed. The now following tools already have to be considered during the running phase of the experiment.

– Be sure that all important and interesting subsidiary values are contained in the output of every run. You may have identified them in advance. Now, they help to understand the runs with more insight and maybe they give the right hint why certain data forces the algorithm to work differently.
– Much more insight to the algorithmic operations can be gained from the results of a profiler, a tool that collects run time information of a program, i. e., it measures the frequency and duration of function calls. Well-known profilers are `gprof`[6] and more recently `callgrind/KCachegrind`.[7] Analyzing such gathered information presents quite exactly and itemized where runtime is spent, how often functions are called, where the algorithm behaves as expected and where not. In general it helps to optimize the code. Here, the number of function calls define a very good picture on the topological structure of the algorithm. In a sense, it gives a function to each called subroutine in the input size. Computed ratios of used time and number of function calls, i. e., normalization, show which subroutines take longer than others. By having a close look at the profiled runs, you can also find out why a worst-case algorithm of $O(n^2)$ behaves in most cases similar to $O(n \log n)$ or why a quite complicated algorithm outperforms a simple one.
  Unfortunately, it depends on the specific algorithm how to analyze all these information and what can be derived from these information. Note that you have to check whether the output of the profiler really makes sense. In most combinations, it is necessary that every involved code is compiled for profiling. Otherwise, durations of subroutine calls, say of external libraries, are assigned uniformly instead of assigning them to the actual calling functions. Let us consider the example, where fast integer multiplication is provided by an external library. The algorithm has two fine-tuned subroutines, one that needs to multiply quite small integers (< 50 bits), another one is multiplying quite long integers (> 200 bits). If both subroutines are roughly called the same number, then a wrong configuration, where functions cannot be

[6] http://www.gnu.org/
[7] http://kcachegrind.sourceforge.net

profiled in detail, will present you that both roughly need the same amount of time, since it just averages over all calls to integer multiplication. But actually, the second routine takes much more time than the first one. The bottom-line is to be careful when interpreting presented data of a profiler.

Do unusual things! At a first glance, this final suggestion sounds spooky. Do not get it wrong. Obviously, you should avoid following futile ideas all the time. But in some situations, it might be helpful to vary an implementation a little bit. In most cases, it will be only justified, that you bark up the wrong tree. But maybe your algorithm turns out to behave better. An example can be a randomized choice of pivot versus a deterministic choice. Bearing away can help to understand better what your algorithm does. Sometimes, you should allow yourself to open your mind to crazy ideas. It helps to be more creative. A lot of serendipities in other sciences originate from doing crazy things rather than following the rules, or, as you might know from famous examples, they happen by accident. In software experiments this may relate to an implementation that is actually buggy, but has a better performance.

1.5.3 Supplementary Advice

The last parts listed in detail how to setup the environment for good experimentation and how to run them with care, based on the assumptions that you know what you want to see. But experimentation is more than just executing algorithms and evaluating the results that support or disprove some hypotheses. Note that experimentation actually consists of more than the setup-phase and a single round of the running-phase. Only a cycle of testing and refinement, supported by profilers, measurement and evaluation of data, allows to identify bottlenecks, to reduce the usage of the memory, or to find out, which intermediate values should be cached and many more. All these efforts may lead to a speed-up in running time, sometimes by an order of magnitude, or even several orders of magnitude.

In case you started with some open research questions, the first results of a running-phase may quickly lead to new questions. It is a law by itself, that good experiments constitute a rich source of new conjectures and hypotheses. Actually, some exploratory experiments, without going too much into details, help to find good initial questions. These may show whether an algorithm is competitive or not. We propose, you spent the first half of your time to generate lots of data and search for patterns and anomalies. Based on this, you can finalize the implementation while you consider the advice mentioned before. Namely, design the important questions and then, perform trustworthy experiments to support your claims. Evaluation may already lead to newsworthy results, or you iterate. Experimenting is a dynamic process, but ensure to fix a point where you stop it.

The end of this section covers two additional subjects. First, we give hints when dealing with approximation algorithms, and second, we outline some details, when jointly experimenting within a group of researchers and sites, respectively.

Approximation or Heuristic Algorithms Up to now, we mainly focused on the running time as the main performance measure of an algorithm. In contrast, an approximation algorithm needs to be handled differently. Its main performance measure consists of the solution quality. In most cases, approximation algorithms compute better solutions the longer they run. So running time and approximation value may be related. However, researchers often choose the wrong stopping criterion, since otherwise their algorithms would run very long. Thus, approximation algorithms usually deal with NP-hard problems, where one obviates handling the exponential number of possibilities, or in problem areas where running time is crucial, whereas the result does not need to be optimal.

Choose the right stopping criterion! There are two critical stopping criteria for an approximation algorithm, namely running time and a known optimal value. For the latter one there is an exception: When an algorithm can prove the found solution to be optimal it is admissible to stop immediately. Unfortunately, most algorithms fail to designate such a proof. But taking a known optimal solution as an a-priori stopping value raises the question why to run the approximation algorithm at all. In practical settings this criterion is purely without any sense, since for any interesting input the optimal solution is surely unknown. Usually, one seeks for near optimal performance values in relation to some other quantity. For example, a good travelling salesman tour with respect to low query times. But, tests with only these special data sets omit to reflect performance in practice and fail to be reproducible. You will see dramatically different running times for similar instances, depending on whether the optimal solution is known or not. As an option one can think of determining a performance measure needed to obtain optimal values, like a certain number of iterations, and use this bound as stopping criterion for data sets whose optimal value is not known. When switching machines, one has to apply benchmarking and normalization as explained in Section 1.5.1.

Fixing a certain amount of time as the running time of an approximation algorithms also contradicts the need of reproducibility. Note that "run-the-algorithm-for-an-hour" is, in some sense, an undefined algorithm. Changes in the setup, e. g., all factors described in Section 1.2.3 that define the experiment like machine, operating system, or implementation, lead easily to results of another quality. In some settings this idea looks like the perfect choice to compare the quality of algorithms, but if you run the same tests on a machine which is much faster, all algorithms will, hopefully, perform better, which is less critical, while a change in the relative ranking is more substantially.

Much better than time bounds are structural measures such as number of branching steps, number of comparisons, or maximal depth of tree as introduced in Section 1.2.2. Using such measures enables us to have a well-defined algorithm, whose running time and quality of solution is expressible as a function of this combinatorial count. At least the latter should be reproducible now. It is possible to combine the solutions with the running time in relation to the combinatorial count. Listings of these relation allows future researchers to compare their solutions to yours and to detect the influences two different environments have. Another possibility of a stopping criterion may be a result that is close to a bound. Such a bound must be easy to compute while *close* means to differ from it only by a small factor, like 0.01. Consider, for example, a minimization problem with an optimal value OPT for a certain instance and for which we know a lower bound LB. The current approximation of a algorithm is given by APP. If $APP < 1.01LB$ we know that $APP < 1.01OPT$, since $LB < OPT$, and the algorithm decides to stop in this case. Note that this requires that the approximation algorithm is able to improve its result with more invested time.

Joint Work Especially when comparing different algorithms one might expect that common work is undesired by competitors. Obviously, no one is interested in losing a game. But at the same time, you might spent more time on other tasks than on implementing someone else's algorithm as enthusiastically as your algorithm, with the goal of an efficient implementation.

What prevents researchers from working together more closely? It is not as worse as it seems. Indeed first steps are already done, e. g., by maintaining common databases for instances. Of course, there are already collaborations when running experiments, but too few at the large scale. If we assume for now, that people of a community are willing to and decide to set up a common laboratory, then we have to check what else needs to be considered in addition to the previous suggestion.

Split the work! As said, no one has the time to do the whole job. One solution is that everyone who wants to participate in the common laboratory has to concentrate only on his small specific task. In terms of algorithmic experiments this equals to provide an efficient implementation of an algorithm. Using version control, it is quite easy to commit new software to a common repository that is set up following the general rules mentioned earlier. Of course, a single site or person has to install the environment, but compared to implementing several efficient algorithms, this is quite an easy task. A discerning reader may come up with the question of how to ensure the same quality for all implementations such as using the same new speed-up tricks. This is indeed a problem, the community has to deal with. We propose to publish the results on a website.[8] Significant changes in the performance values are immediately visible to everyone involved, which results in asking questions and starting discussions, why some algorithm performs much better than others. In general, implementing an algorithm needs to follow some common guidelines to be constituted when starting the collaboration. All questions related to this should be covered in Chapter 6.

Generating the actual results requires two following steps. First, one has to select on which instance sets the algorithms should be tested. Second, one has to run the experiments. Instances that should obviously use the same common

[8] The community has to decide whether the site is publicly available or closed to members.

format, either come from an instance database, or researchers can put their own generated data sets also under version control and then combine it with algorithms. In some cases, the group may agree to have a committee to decide which combinations make sense. For the second task, the group may rely on some scripts to be written. Actually, the whole procedure especially makes sense when running the experiments regularly, while algorithms, or at least their implementations, are still under development, which means that there are still untested ideas.

Finally, everyone profits from the collaboration since progress in some algorithm is visible to all, and people start to discuss and to improve their own implementation based on this knowledge. Additionally, a regular execution of the committed experiments makes it easy to check out, how a new heuristic and algorithmic idea performs. Since the work assigned to an individual is quite small, while constantly comparing with others, the idea of a common laboratory as presented here, seems to be a fruitful environment for experimental research in algorithmics, and we encourage communities to install corporate laboratories.

1.6 Evaluating Your Data

After you have run your experimental setup you are left with a bunch of data. The next task is to figure out whether this data supports your working hypothesis and what else may be deduced from it.

The first thing to keep in mind is to look at the data without being biased by your working hypothesis. Of course, the working hypothesis provides a starting point for the investigations.

In general, it is important to observe patterns in the data and to try to explain them. This explanation step may involve a more detailed analysis and also new experiments. For example, you might discover that a branch-and-bound algorithm using your new pruning rule performs worse than using the old one. The reason for this might be either poor pruning or too much time spent for the pruning so that in total it does not pay off. To investigate this question you would need to look at the number of nodes visited by the algorithm and the fraction of the time spent for pruning. Depending on your experimental setup, you may be able to derive this additional data from the results you already have. Otherwise you would need to rerun your experiments.

It usually pays off to let your experimental setup generate "raw" data, i. e., instead of averages and maybe minimum and maximum record all values, as well as related quantities which might be of interest. Although this may create large amounts of data it saves you from running your probably time-consuming experiments often. Nevertheless you should always think about whether the data you have is really sufficient to provide support or discrepancy for your hypothesis. If this is not the case you need to gather more data.

The significance of your findings is increased if you can provide explanations or more detailed accounts. For instance, it is not only interesting which algorithm runs faster, but also where the respective running times come from,

i. e., which parts of the algorithm contribute to the running time. Sometimes it is possible to look at more machine-independent running time measures, for example nodes evaluated in a branch-and-bound algorithm, improvement steps taken in a local search heuristic, or simply the number of iterations. It may be worthwhile to investigate how these measures depend on instance size, since the machine-independence of these measures gives better insights in the algorithm rather than the computing environment.

So far we have only talked about the general evaluation philosophy. In the following, we describe two ways of actually deriving something interesting from your data. The first method is *graphical analysis*, which uses diagrams and plots to discover patterns. Although this sounds simple, it is indeed a standard tool of statisticians for arriving at good hypotheses. Graphical analysis provides key insights and can also give some evidence for conclusions.

Then we give an overview of *statistical analysis*, which provides numerical methods that can be used to check hypotheses, e. g., those obtained via graphical analysis. Statistical analysis is a tool that is widely used in other experimental areas, but has rarely been applied to experiments on algorithms. However, Barr et al. [4, p. 22] recommend to employ statistical analysis wherever possible.

A drawback of statistical analysis is that it assumes certain experimental setups which sometimes cannot be achieved. In this case, statistical analysis is not applicable; but graphical analysis always is.

1.6.1 Graphical Analysis

Pictures and diagrams can be of great help to realize what is going on, since they can represent vast amounts of data in a succinct way, at least if done properly. This makes it easy to spot patterns which otherwise would be lost in a pile of numbers or in large tables. The main issue here is to find the "right" diagram that reveals the things we are interested in. This diagram serves two purposes: First, it gives you some insight you did not have before and thus guides your investigations. On the other hand, it may be useful to communicate your results to other people. Further hints on this use will be given in Section 1.7.2.

There are some guidelines on using diagrams for analyzing numerical data in the statistics literature. Other sources of inspiration on how to employ diagrams can be found in the literature on experimental algorithms. The paper of Sanders [55] gives extensive advice on how to use diagrams to report experimental results in algorithmics and has been a major source for this section. Most of this advice is helpful for analysis too, so we present it here. As examples, we will just name a few types of diagrams commonly used in the experimental literature and highlight their uses.

The diagram type most often encountered in the experimental literature on algorithms plots some metric (e. g., running time) as a function of some parameter (e. g., input size of instance). The usual interpretation is that the variable on the x-axis is "independent", whereas the variable on the y-axis is "dependent", i. e., there is a functional relation between the two. This relation is most often interpreted to imply causality, so this diagram type is most suited for settings

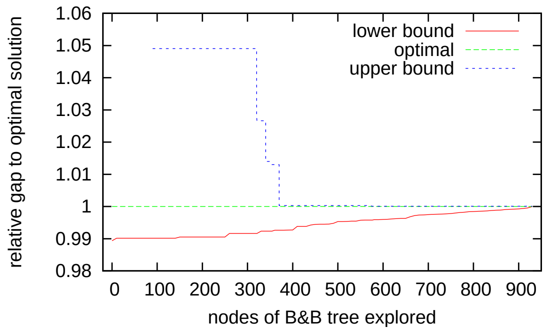


Figure 1.1. A functional plot showing the typical behavior of branch-and-bound algorithms. Displayed are the upper and lower bounds relative to the optimal value evolving with the number of nodes of the tree that have been explored so far. Every time an improved solution is found, the upper bound drops and remains on this level for some time. Note that an optimal solution has been found after about 370 nodes, but it takes another 550 nodes to raise the lower bound such that optimality can be proven.

where assuming this causality is reasonable. In the example, this causality is given: an increase in input size causes an increase in running time. We will call this diagram type *functional plot*.

Figure 1.1 gives an example of a functional plot, which is in fact a special case, namely a time series, where time is shown on the x-axis. Time is not given explicitly here, but the number of nodes explored so far is of course some sort of time-scale. Time series are often used to show the convergence of algorithms.

If functional plots are used to give average values, they should be augmented to depict more information of the range of the data. One way to do this is to provide error bars which indicate the standard deviation of the data. Even more information is contained in box plots which characterize the underlying distribution by five values and are explained in more detail later.

A diagram type often used to investigate the relationship of two variables is the *scatter plot*. The graph of the scatter plot is just the set of points corresponding to the measurements, see the example in Figure 1.2. It is adequate if it is unclear which of the variables is "independent" or "dependent". A scatter plot can be applied if the data are not ordered quantities and thus cannot be associated to points on the axes. For instance, it is not clear in which order to put the instances you measured solution quality for in order to come up with a suitable functional diagram. Instead, you can use use a scatter plot relating instance size to solution quality. A scatter plot can also be used to compare many different measurements, e. g., the performance of many TSP heuristics in Figure 1.2.

Other famous diagram types are the *bar chart* and the *histogram*. The bar chart consists of bars, whose heights represent numerical quantities and are scaled proportionally. Thus they ease visual comparison and are appropriate

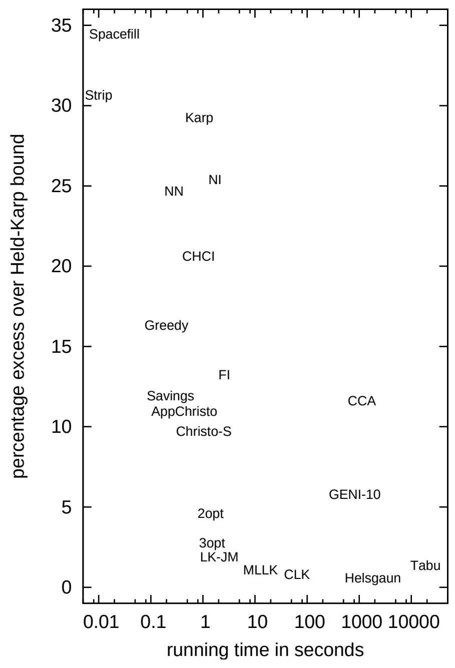


Figure 1.2. A scatter plot showing the approximation ratio and the running time of some heuristics for the symmetric TSP from the report of Johnson and McGeoch [31], p. 382. The data represents averages over a set of 10,000-city random Euclidean instances. Each heuristic's label depicts its average excess over the Held-Karp lower bound, which is a well-known and rather good lower bound for TSP problems. In this special case of a scatter plot the data points are marked with the name of the heuristic they arise from.

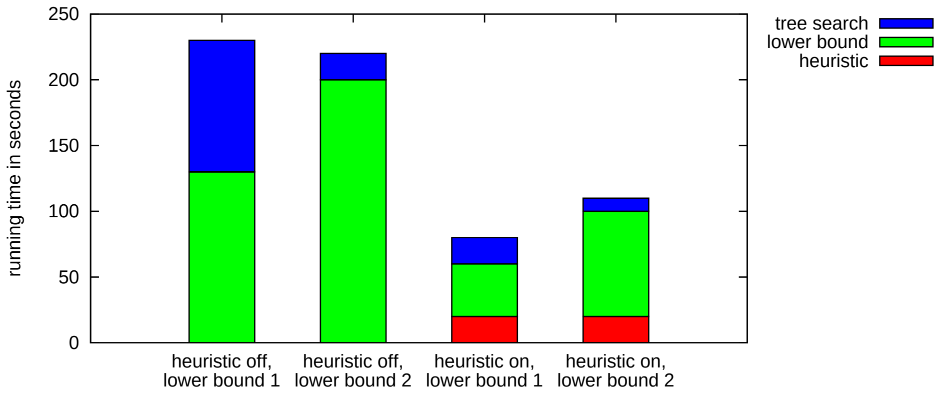


Figure 1.3. A bar chart showing hypothetical data for a branch-and-bound algorithm. The diagram shows running time data for four different settings used to solve the same instance. An initial heuristic can be used or not and there is a choice between two kinds of lower bounds. Lower bound 1 runs fast and gives weak lower bounds, whereas lower bound 2 runs longer and gives stronger bounds.
Obviously, the better quality of the lower bounds provided by method 2 significantly decreases the time spent for searching the tree, since fewer nodes need to be visited. However, it only pays off to use lower bound 2 if the heuristic is not used, since the total time is lowest if the heuristic and lower bound 1 are used.

in situations where multiple quantities need to be compared. Figure 1.3 gives an example of a bar chart used for assessing the usual tradeoff on a branch-and-bound algorithm.

A special kind of a bar chart with a different purpose is the histogram. Histograms are used to analyse distributions of some quantity. To this end, the range of the quantity is divided in so-called buckets, i. e., intervals of equal size, and for each bucket the percentage of values lying in this interval gives the height of the bar in a bar chart. Figure 1.4(a) shows a variant of a histogram known as *frequency polygon* [37], where the data points of the histogram are connected by lines instead of being represented by bars. This type of diagram is better suited to comparing a set of distributions.

In statistics, distributions are often compared using the already mentioned box plots, also known as box-and-whisker diagram. Box plots are based on quartiles, which are special quantiles. The (empirical) p-quantile a_p for $0 \leq p \leq 1$ of a sample of n numbers $x_1, \ldots, x_n$ is defined as

$$a_p := x_{\lceil pn \rceil}$$

i. e., a_p is the smallest value such that at least pn values of the sample are less than a_p. The box plot uses quartiles, which are the quantiles $a_0, a_{0.25}, a_{0.5}, a_{0.75}, a_1$.

Notice that a_0 is the minimum, $a_{0.5}$ the median and a_1 the maximum of the distribution. A box plot of a distribution consists of a line ranging from a_0 to a_1, where the interval $(a_{0.25}, a_{0.75})$ is drawn as a larger box, which contains an extra line indicating $a_{0.5}$. See Figure 1.4(b) for an example that gives the same data as the frequency polygon diagram in Figure 1.4(a).

Of course, sometimes these diagram types do not fit the purpose or the data to analyze. In this case you should try to make up your own kind of visualization for your data or look into one of the many sources on statistical graphics and exploratory data analysis, e. g., [63, 62, 14].

Apart from choosing a suitable type of diagram there is a lot to be gained by using appropriate scales on the axes and focusing on the most interesting part of the diagram. Most common are linear and logarithmic scales, where the first is appropriate if the numbers are in a relatively small range whereas the latter is useful if the numbers are of different orders of magnitude. For instance, if one is interested in the asymptotic behavior of the running times of some algorithms as a function of the instance size, instance sizes usually grow by a constant factor in order to cover instances sizes of different magnitudes with few instances. In that case, both axes should be logarithmic, since instance sizes grow exponentially by setup, and running times are exponential too if they are at least linear in the input size. Similarly, if instance sizes grow additively by a constant but the running times of the considered algorithms are known to be roughly exponential, it is a good idea to use a logarithmic y-axis scale.

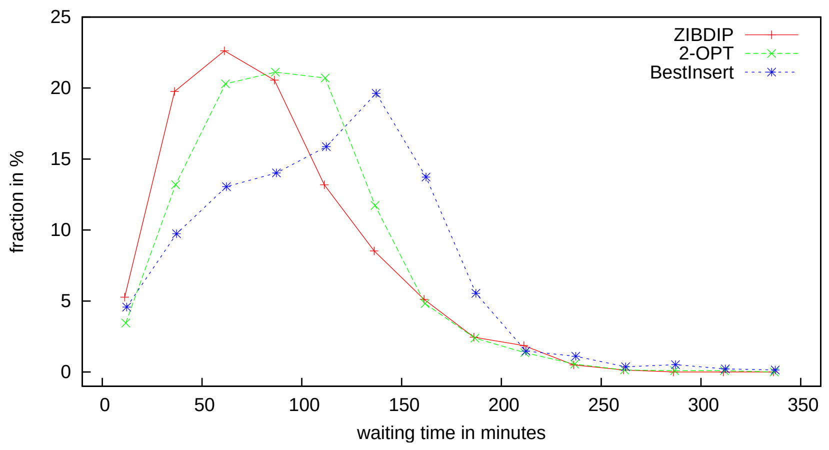
Sometimes the problem itself suggests a suitable unit for one of the axes, which may even allow to get rid of one degree of freedom [55].

It may pay off to invest problem-specific knowledge to find an interesting view on the data. Normalization, as suggested by Johnson [32], is an example. Suppose you know a lower bound for a set of functions you want to compare, e. g., $f_1(n), f_2(n) \in \Omega(n)$. Then it may be helpful to look at $f_1'(n) := f_1(n)/n, f_2'(n) = f_2(n)/n$ instead of f_1, f_2, since the "common part" is factored out and thus differences become more visible. Although you lose the possibility to directly read off the values from the normalized diagram, it is still possible to get a good intuition. As Sanders [55] points out it is usually possible to find intuitive names for this new quantity.

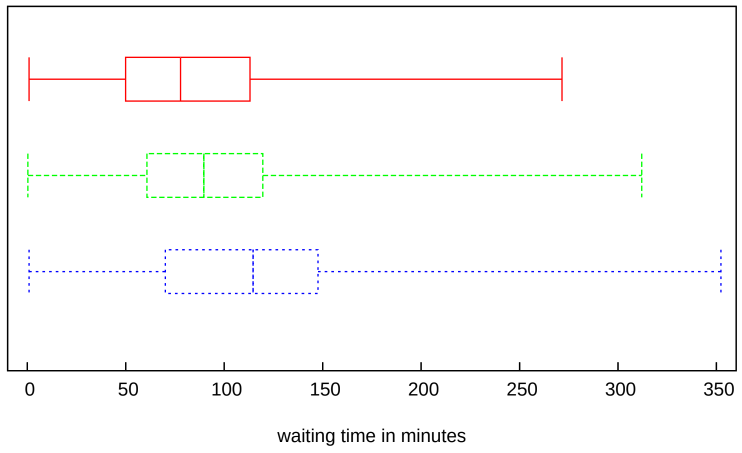
A simpler "close-up" effect can be gained by adjusting the plotted range $[y_{\min}, y_{\max}]$ of the y-axis. However, this has the disadvantage that relative comparisons are no longer possible visually. The y-range can also be narrowed down by clipping extreme values of clearly dominated algorithms.

To give an impression on what a good diagram can achieve, we cite the following example from Johnson [32, p. 26]. Consider the data in Table 1.2, which gives running time in seconds for five different algorithms, depending on the input size. From the way the data is arranged it is obvious that the running times of different algorithms are ranked in a consistent way over all instance sizes.

In order to learn more about the data we generate a diagram. The first shot is diagram 1.5(a) in Figure 1.5, which depicts just the data as-is on a linear

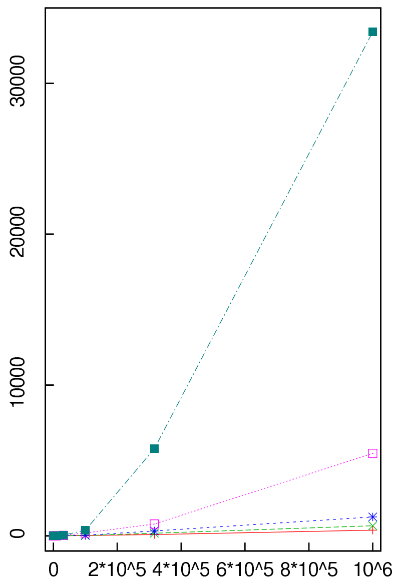


(a) A frequency polygon plot of the waiting time distributions.

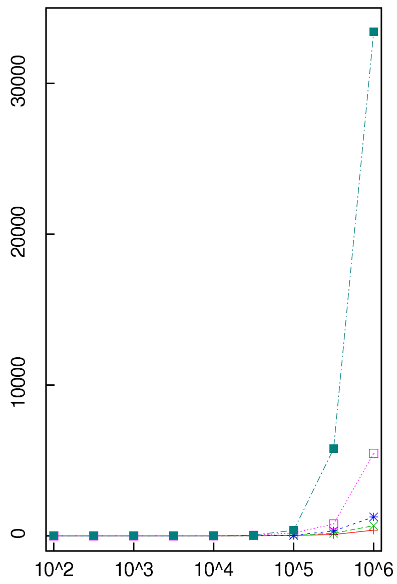


(b) A box plot of the waiting time distributions.

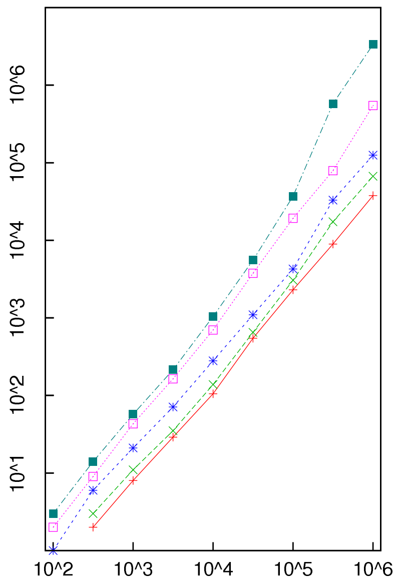
Figure 1.4. Comparison of waiting time distributions achieved by some vehicle dispatching algorithms. The data is taken from a computational study that compares algorithms for dispatching service vehicles, where the customers' waiting times are the major quality of service criterion [24]. BestInsert and 2-OPT are heuristics based on local search, whereas ZIBDIP is an exact algorithm based on Integer Programming techniques.
Both diagrams indicate that optimization algorithms achieve a much better waiting time distribution than simple heuristics. In the frequency polygon plot the distribution of ZIBDIP is the one that is farthest left, indicating short waiting times for many customers. In the box plot, all quartiles of ZIBDIP's distribution are smaller than the respective quartiles of the heuristics' distributions, giving the same conclusion.



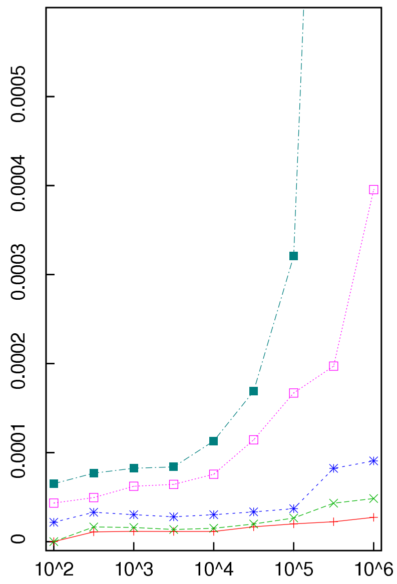
(a) Raw data with linear axes.



(b) Data with a logarithmic x-scale.



(c) Data with both axes scaled logarithmically and multiplied by 100 for convenience.



(d) Data with logarithmic x-axis and normalized by $n \log n$, which is an overall lower bound.

Figure 1.5. Effect of different choices for the scaling of the diagram. Clearly, the amount of information discernible from the diagram increases: In diagrams 1.5(c) and 1.5(d) it is evident that the ordering of the algorithms is consistent. However, only diagram 1.5(d) reveals that the performance of algorithms D and E are asymptotically much worse than the lower bound, a fact that cannot be seen directly from the table.

instance size	100	316	1000	3162	10000	31623	100000	316227	1000000
Algorithm A	0.00	0.02	0.08	0.29	1.05	5.46	23.0	89.6	377
Algorithm B	0.00	0.03	0.11	0.35	1.38	6.50	30.6	173.3	669
Algorithm C	0.01	0.06	0.21	0.71	2.79	10.98	42.7	329.5	1253
Algorithm D	0.02	0.09	0.43	1.64	6.98	37.51	192.4	789.7	5465
Algorithm E	0.03	0.14	0.57	2.14	10.42	55.36	369.4	5775.0	33414

Table 1.2. Running time data of Johnson's example [32].

scale. Algorithms E and D seem to be much worse than the other three on large instances. However, there is no clear picture for smaller instances, since the plots essentially coincide. Furthermore, almost all data points are in the left half of the diagram. Changing to a logarithmic x-scale (diagram 1.5(b)) fixes this, but still there is much coincidence of the plots. If both axes are logarithmic (diagram 1.5(c)), the consistent ranking of the running times becomes apparent. Now all five plots seem to have approximately the same slope, which would indicate the same asymptotic behavior. We know from the first diagram that this is not quite true. Let us now put some more knowledge in the game. Johnson states that there is a lower bound of $\Theta(n \log n)$ for all algorithms. We can use this to normalize the running time of the algorithms and still keep the logarithmic x-axis to obtain diagram 1.5(d). This diagram is really revealing: It shows the consistent ranking, brings out the asymptotically worse running times of algorithms E and D, and indicates that the other three algorithms are asymptotically optimal (up to a constant).

Some of the suggestions involved a lot of work, e. g., playing around with different types of diagrams, transformation of the data, looking at different combinations of measures and so on. However, much of this work can be automated using scripting languages, which even makes it fun to do these things. For example, languages like Perl and Python can be used to extract exactly those numbers you are currently interested in and to convert them in a format suitable for further processing. This processing can be done by a spreadsheet application or a graph-drawing scripting language such as gnuplot.

You should also keep in mind that your data has only a limited precision, which is usually smaller than the number of digits available. This is especially true for running times, which often vary much depending on factors you cannot control. This variance can be reduced by producing multiple measurements and using the average or more sophisticated variance reduction methods mentioned in Section 1.2.4. The danger in pretending too much precision is to end up analyzing the noise of the measurements.

1.6.2 Statistical Analysis

The purpose of this subsection is to give an impression of how statistical analysis works and how it can be applied to analyzing data describing the performance of algorithms. We will review the basic concepts and main ideas and give specific examples from the literature.

Why should one be willing to use statistical analysis when the kind of ad-hoc numerical data analysis used before seemed appropriate? One reason is that statistical analysis, applied properly, can give much stronger support for claims or indicate that claims are not justified by the data. It is a tool for assessing the explanatory power and significance of your data. Moreover, you can gain a deeper understanding of the data, for instance, it is possible to analyze the impact of parameter choices on the performance of the algorithm and to distinguish between significant and insignificant parameters. An example will be discussed later in this section. Finally, a statistical analysis of your data may suggest directions for further experiments.

A key ingredient for a proper and successful statistical analysis is a carefully designed experiment. In fact, there is a whole branch in statistics concerned with this, naturally it is called *Design of Experiments (DOE)*. Barret al. [4, p. 20] suggest that "all doctoral students of operations research should receive training in DOE". Even if you do not use its methods, knowing the methodology leads to clearer thinking about experiments and their pitfalls.

DOE methodology is widely used in other experimental fields, such as psychology, the social sciences or industrial product development. It provides methods for designing experiments such that certain systematic errors can be eliminated or at least reduced and the influence of nuisance factors can be controlled. Furthermore, there are some well-established so-called *experimental designs*, which describe how to carry out the experiment. For these designs, DOE provides analysis methods as well as methods to check the model assumptions a posteriori.

The general tool to analyze the data is *hypothesis testing*. A statistical test is characterized by a so-called *null hypothesis*, assumptions on the experiment, i. e., how the data is generated, and a *test statistic*, which is a number computed from the data. The purpose of the test is to check whether some data is consistent with the null hypothesis or not. The null hypothesis is the converse of the research hypothesis of the experimenter, and the research hypothesis makes up the alternative hypothesis. If the null hypothesis is not consistent with the data, it is rejected and there is some evidence that the research hypothesis is true.

Before doing the test, you need to choose a number $0 < \alpha < 1$, the *significance level*, which is something like the confidence you want to achieve. For example, $\alpha = 0.05$ tells you that you are ready to accept 5% error, i. e., when doing the test very often (of course with different data), the result may be wrong for 5% of the trials. Then you just compute the test statistic for your data, compute or look up the so-called p-value of the statistic. If the p-value is smaller than the chosen significance level, the null hypothesis is rejected.

The formal background of hypothesis testing is the following. It is assumed that the assumptions and the null hypothesis hold. One can then compute the probability that the realization of the test statistic is obtained under these assumptions; this is exactly the p-value. If this probability is very low, in particular smaller than the confidence level, this result is rather unlikely and thus provides evidence against the null hypothesis, leading to its rejection.

As an example, we will describe the famous sign test (see [58]). It works on a sample $(x_1, \ldots, x_n)$ of n real numbers. The assumptions are that all the x_i are drawn independently from the same distribution. The null hypothesis is that 0 is the median of the distribution. To compute the test statistic S, remove all x_i that are 0 and decrease n accordingly. Now define S by

$$S := |\{i \mid x_i > 0\}|$$

Notice that only the sign of x_i matters, hence the name of the test. If the null hypothesis is true, the probability that x_i is greater than zero is the same as that it is smaller than zero, i. e., this probability is $1/2$. Therefore, S is distributed according to a binomial distribution with parameters $1/2$ and n. Suppose the observed value of S is k, w. l. o. g. $k \geq n/2$. Now the p-value is easy to compute: It is just the probability that S is *at least* k, that is $1/2^n \sum_{i=k}^{n} \binom{n}{i}$. For example, if $n = 15$ and $k = 12$ we get a p-value of 0.018, leading to a rejection of the null hypothesis at a significance level of $\alpha = 0.05$. Instead, we have some evidence that the real median is *greater than* 0. Note that we could not conclude this if we selected a significance level of $\alpha = 0.01$.

A standard application of the sign test is to compare pairwise samples from two different distributions. For comparing two algorithms, suppose there are two samples $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ where x_i and y_i are performance measures of both algorithms on the same instance i. The question is: Is it true that the first algorithm is better than the second? To answer this question, consider the sequence of *differences*, given by $d_i = y_i - x_i$ and do the sign test on this sample. The null hypothesis is that the medians of the performance distribution are equal, i. e., the performance of both algorithms is the same. If sufficiently many d_i are positive, this null hypothesis is rejected and there is evidence that the first algorithm is better. Notice, however, that the null hypothesis is also rejected if there are *too few* positive d_i, which would indicate that the second algorithm is better.

One final note about the assumptions of the sign test in this application. These were that the differences are drawn independently and from the same distribution. Clearly, the assumption “same distribution” is no problem, since we look at the distribution of running times difference on all possible instances. If the instances are generated independently at random, the independence assumption is obviously fulfilled. However, this is not true if we look at selected (real-world) instances. Applying the sign test in such a setting is only valid if we can be sure that the selected instances are reasonably representative and diverse or we restrict ourselves to instances that “look like these sample instances”.

Let us now turn to some example applications of statistical analysis from the literature.

The Sign Test and Heuristics for the TSP This example is taken from Golden and Stewart [20], who compare a new heuristic for the Euclidean Traveling Salesman Problem (TSP). They also give some introduction to the statistical methods used.

Golden and Stewart introduce the new algorithm *CCAO* which combines four techniques. It starts constructing a partial tour from the *c*onvex hull of the cities, includes remaining cities via criterions known as *c*heapest insertion, *a*ngle selection and finally improve this solution via a postprocessor known as Or-opt. Other successful postprocessors are 2-opt and 3-opt, which try to find better tours by exchanging 2 or 3 edges of the current tour until no further improvement is possible. It is known that solutions produced by 3-opt are usually a bit better than those of Or-opt, which in turn are much better than those of 2-opt. Unluckily, the gain in solution quality comes at the price of substantially longer running time.

The study is based on only eight instances, which seem to be among the largest ones that have been published at that time (1985). It is not clear that this selection of instances is representative as required for a good test set as explained in Section 1.4. Moreover, usually a larger number of samples is required in order to draw statistical significant conclusions. In fact, Design of Experiments theory provides methods to compute in advance how many samples are necessary to reach a given significance level. However, the main purpose of the paper is to promote the use of statistical methods for assessing algorithms.

In a first experiment the authors compare CCAO to other heuristics. Applying the sign test to assess solution quality indicates that CCAO is better than heuristics with a weak postprocessor, i. e., 2-opt. They also realize that CCAO is as good as those with a strong postprocessor, i. e., Or-opt or 3-opt.

In their second experiment they evaluate the influence of accuracy and efficiency of the postprocessor. This is done by combining the first three ingredients of their algorithm ("CCA") with each of the three postprocessor and the value without postprocessing. Applying the sign test again, they are able to verify the following:

– The running time of 2-opt is smaller than that of Or-opt which is smaller than 3-opt on all 8 instances.
– The solution quality of 2-opt is worse than both Or-opt and 3-opt.
– The solution quality of Or-opt and 3-opt is statistically indistinguishable.

They also did further experiments to assess the contribution of the algorithm's ingredients.

There is an extension to the sign test, namely the Wilcoxon test, which takes the value of the differences into account and allows stronger conclusions at the price of stricter assumptions. Although applicable, Golden and Stewart did not apply the Wilcoxon test for their worked-out analyses, but encourage the reader to do so.

Using Design of Experiments Methods to Assess Network Algorithms Amini and Barr [2] conducted an elaborate study regarding the performance of network algorithms for reoptimization, as it often arises e. g., in branch-and-bound algorithms. Their goal is to find out which of the three algorithms PROPT, DROPT and KROPT is best suited for reoptimization.

To this end, they want to perform the following kind of experiment. Starting from a base instance they generate a series of sub-instances, which are randomly modified versions of the base instance, with only small changes between them. This is typical for reoptimization-based algorithms.

Amini and Barr study the following five factors:

Factor	**Levels**
class of network problem	transportation, transshipment
problem size	small, medium, large
type of change	cost, bound, RHS
percentage change	5%, 20%
type of reoptimizer	PROPT, DROPT, KROPT

Two other factors, the number of sub-instances per series and the number of changes, are fixed to 200 and 20, respectively, after some pilot experiments (which are evaluated by statistical analysis). All in all there are 108 experimental conditions to be studied.

The analysis is based on a *split plot design*, which is an advanced design from the theory of Design of Experiments, see e. g., [11]). A main feature of the split plot design is that the influence of a subset of the factors is better estimated than the influence of combinations of the remaining factors, which are called blocked factors. However, a split plot design enables good statements about the influence of non-blocked factors for a *fixed* combination of the blocked factors. In this case, the blocked factors are problem class and size. This means that the experiment yields insight about how the non-blocked factors (type of change, percentage change, and reoptimizer) should be combined for each problem class / problem size combination, which is really interesting.

The actual experiment is run as follows: In advance, four base instances per problem class and problem size combination have been fixed. Now one out of the 108 conditions is selected at random, the base instance is chosen randomly and 200 subproblems according to the remaining parameters are generated randomly. Finally, all three reoptimizers are run on them and the total CPU time is recorded. All in all, 86,400 subproblems are solved.

The authors report the following results obtained by using Tukey's HSD (Honestly Significant Difference) Test [56, 11]. This test yielded detailed information on the influence of combinations of factors. For example, considering the two factors type of change and reoptimizer, the HSD test indicates that it is best to choose PROPT if cost coefficients have changed, whereas DROPT deals best with changes to the bounds or the RHS. Looking at the four factors problem class, type of change, problem size and reoptimizer, the TSD results were (cf. Table 1.3):

– transportation problems:
  - PROPT performs best for medium and large problems with cost changes

  - DROPT performs best for bound changes on large problems and for RHS changes on medium and large problems
  - on all other combinations, PROPT and DROPT are indistinguishable, but better than KROPT
- transshipment problems:
  - PROPT performs again best for medium and large problems with cost changes
  - all three algorithms are indistinguishable for bound and cost changes on small problems
  - in the remaining cases, PROPT and DROPT are indistinguishable, but better than KROPT

All of these results were obtained using a significance level of 5%.

It is important to note that the careful design of the experiment allowed the application of HSD test, which in turn provided very detailed information on when to choose which algorithm.

Linear Regression for Comparing Linear Programming (LP) Algorithms In their overview paper on statistical analysis of algorithms Coffin and Saltzmann [7] propose a method for comparing algorithms they call head-to-head comparison. They illustrate this method on data from the literature, which evaluates the interior-point LP solver OB1 to the simplex-algorithm-based LP solver MINOS.

The fundamental idea of head-to-head comparison is to express the running time of one algorithm depending on the running of the other, allowing a direct comparison. Coffin and Saltzmann propose the following dependence

$$y = \beta_0 x^{\beta_1} \epsilon,$$

where x and y denote the running time of MINOS and OB1, respectively, ϵ is a (random) error and β_0, β_1 are unknown constants. This relation has interesting desired properties. First, if the running time for MINOS is 0, the running time for OB1 is 0, too. Second, assuming $\beta_0, \beta_1 > 0$ we have that if the running of MINOS increases, those of OB1 does also. Notice that this need not hold for particular instances (differences there go in the error ϵ), but describes a general trend. Finally, $\beta_1 = 1$ indicates that the running times are proportional. A drawback of this model is that as the running time of MINOS increases, so does the variance of OB1's running time, which is undesired since it hinders using tests and regression methods.

This drawback can be alleviated if using a log-transformation, yielding

$$\log y = \log \beta_0 + \beta_1 \log x + \log \epsilon.$$

This transformation has two positive aspects. Once, it reduces variance. Second, we now have essentially a linear model. Thus linear regression can be used, giving $\beta_0 = 1.18$ and $\beta_1 = 0.7198$ as estimates. Furthermore, a hypothesis test for the

(a) Results for transportation problems

type of change	problem size	KROPT	PROPT	DROPT
cost	small		○	○
	medium		●	
	large		●	
bound	small		○	○
	medium		○	○
	large			●
RHS	small		○	○
	medium			●
	large			●

(b) Results for transshipment problems

type of change	problem size	KROPT	PROPT	DROPT
cost	small	○	○	○
	medium		●	
	large		●	
bound	small	○	○	○
	medium		○	○
	large		●	
RHS	small	–	–	–
	medium		○	○
	large		○	○

Table 1.3. Dominance relations between PROPT, DROPT, and KROPT for the 4-factor combination (problem class, problem size, type of change, reoptimizer) extracted from the experimental data of Amini and Barr [2]. A "•" indicates that this algorithm dominates the others, whereas a "∘" indicates algorithms that could not be distinguished from each other, but dominated the remaining algorithms. Finally, situations in which no results could be obtained are marked "–".

null hypothesis $\beta_1 = 1$ can be done, yielding to reject this hypothesis at a p-value of 10^{-4}. Thus it is reasonable to assume that OB1 is asymptotically faster than MINOS.

Apart from this case study, Coffin and Saltzmann give many more case studies and lots of hints for statistical analysis of experiments on algorithms. The examples presented here are supposed to give a flavor of how statistical analysis can be applied to experimental analysis of algorithms. It has to be stressed, however, that the methods of statistical analysis have to be applied with great care to get meaningful results. We will say a little bit more on this in the next section.

1.6.3 Pitfalls for Data Analysis

So far we introduced some methods to analyze experimental data. We want to conclude this section by mentioning common pitfalls to watch out for.

Graphical Analysis As mentioned in the section on graphical analysis, on the one hand a good diagram can greatly contribute to the analysis. On the other hand, using a bad diagram can be misleading. Therefore it is important to use a diagram type that is suitable for the type of analysis done. For instance, it may happen that due to a logarithmic scale a small absolute difference seems to be substantial and thus leads to wrong conclusions.

As Bast and Weber [5] point out, one has to be careful when dealing with averages, especially if different performance measures are involved. In particular, if algorithm A is better on average than algorithm B with respect to one performance measure, this does not say anything about the relation with respect to another performance measure, even if there is a monotone transformation between the performance measures. To see this, just suppose that algorithm A is good on average for the first performance measure, but is very bad on some instances, whereas algorithm B is not as good, but never very bad. If the other performance measure now penalizes bad behavior more strongly, algorithm B may become better than algorithm A. Bast and Weber emphasise that even if the standard deviation intervals that are usually indicated by error bars are disjoint, it is possible that the order of the averages reverses.

The solution to this issue is of course to evaluate each performance measure on the raw data and only average afterwards. This is another reason for collecting raw data instead of averaged or aggregated data.

Statistical Analysis Every statistical test requires some assumptions on the stochastic nature of the data. A statistical test is invalid if these assumptions are violated and therefore conclusions drawn from them may not be trustworthy. Therefore, it has to be checked and possibly discussed whether the assumptions are reasonable. If some of them are not, it is often possible to ressort to some weaker test. Furthermore, some tests are more robust than others. The literature on nonparametric statistics usually contains hints on the robustness of tests and the assumptions required, see e. g., [59, 8, 58].

A similar problem may arise when analysing data from a designed experiment. One usually uses some kind of probabilistic model for the data. For any analysis to make sense, the model should be appropriate in the sense that it "fits" the data (or vice versa). There are some ways to test the "fit" and the fulfillment of the assumptions which are discussed in the DOE literature ([11]). These tests should always be done *before* any analysis is carried out. Furthermore, the type of analysis done has to be applicable to the design and model used.

1.7 Reporting Your Results

When reporting your results you usually want to convince the reader of the scientific merit of the work. An important requirement for this is that you raise and answer interesting questions. However, for experimental work it is equally important that the results are reproducable. When talking about reproducibility, we do not mean that an experiment can be redone exactly as it was, since this is unachievable, given the rapid development of computing equipment. Instead, we think of a weaker form of reproducibility: An experiment is reproducable, if a very similar experiment can be set up which gives the same quantitative results and conclusions.

These requirements lead to some principles for reporting which are considered to be good practice [32,4]) and will be discussed in detail here. Finally, we provide hints on good use of tables and diagrams for reporting experimental data and conclusions from it.

1.7.1 Principles for Reporting

This section is organized around the following principles for good reporting, which are slightly adapted from the list of principles given in Johnson [32].

– Ensure newsworthiness of results.
– Indicate relation to earlier work.
– Ensure reproducibility and comparability,
– Report the full story.
– Draw well-justified conclusions and look for explanations.
– Present your data in informative ways.

Ensure Newsworthiness of Results. This principle directly relates to the scientific merit of your experimental work. Clearly, it is necessary to deal with interesting questions on a sound basis, regarding your experimental methodology. As mentioned earlier, it is often more appealing to go beyond pure running time comparison. These questions were explained in more detail in Section 1.2 and others.

A good report states clearly the motivation for the work and describes the context of it, explaining the specific contribution of this work. The motivation may come from e. g., questions raised in earlier experimental papers, assessing the "practical" performance of algorithms studied only theoretically, and from real-world applications.

Indicate Relation to Earlier Work. Of course, you should have read the relevant literature to know what already has been done.

You should compare your results to those from the literature. This comparison can be a hard task for a number of reasons. First of all, you will most likely be using different computing hardware and software. Since running times are influenced by many factors, e. g., machine speed and architecture, compiler, and sophistication of implementation, a direct comparison is not very meaningful. Another obstacle is that earlier publications may not focus on aspects you are interested in, use other performance measures and so on.

A part of this difficulty can be overcome if it is possible to use the test set and the implementation of the original work. If available, you should use these. How to proceed when they are not available has been discussed in Section 1.5.1. The main benefit of using the original implementation is that you get the best comparability possible, since you can run the original algorithm and your new one on the same equipment.

A fallback method to make running times of earlier papers roughly comparable to your measurements is to estimate the relative speed difference of the machines. This can be done using benchmark values obtainable for both machines. Sometimes problem-specific benchmarks are available. For instance, the DIMACS Implementation Challenge on the Traveling Salesman Problem [31,13] employed a benchmark implementation to normalize the running times across a wide range of different platforms. To this end, every participant had to run this benchmark implementation on his machine and to report the running time, which in turn was used for normalization. Johnson and McGeoch [31] report that accuracy was about a factor of 2, which was sufficient for the running time differences that occurred.

In any case you should clearly report on how you tried to make these values comparable.

Ensuring Reproducibility and Comparability. This principle is in some sense the counterpart of the preceding one. The goal is to make life of future researchers who want to build on your work easier, which essentially means providing enough detail to allow qualitative reproduction of your results.

To this end, you should give a detailed description of the experimental setup. This encompasses information such as machine type, processor number and processor speed, operating system, implementation language and compiler used, but also experimental conditions like run time or space limits. If you used a generator to create test instances you need to describe the generator and the parameters used for test set creation, too.

Of course it is necessary to describe the implementation of your algorithm detailed enough to facilitate reproduction. This implies that you mention and describe all non-straightforward implementation details which have a significant impact on your results. For complex heuristics, this includes the stopping rule used (if the heuristic has no natural way to terminate) and the values of potential parameters used to achieve your results. These parameters must not be set on a per-instance basis, since this is not generizable. However, you may use some kind of rule to determine parameters from instance parameters which then needs to be described as well.

The best way to ensure reproducibility is to publish both the instances used for the experiment and the source code of your implementation. Some journals already support and even encourage this. For example, the ACM Journal on Experimental Algorithmics invites submitters to also publish supplementary files, which can be source code or data files. Publishing the source code requires a certain level of documentation to make it useful for other people. You also have to make sure that the data you publish is actually consistent with the source code, i. e., binaries will produce essentially this data.

Instances should be made available in a machine-readable, well-known and well-documented format, cf. Section 1.4. If there is already an instance library for this specific problem, it may be possible to extend the library by some of your instances, since instance libraries are often maintained to reflect progress. Although it would suffice to publish the instance generator used, it is usually better to make the actually used instances available.

It is a good idea to archive the raw data of your experiments (not just the "processed" data used and given in the report) at a safe place so you can later access it. This can be useful if you or somebody else is interested in doing further research. Again, version control systems can be useful here.

Report the Full Story. It is good scientific practice to report results (i. e., data) as they are. This also implies that anomalous results contradicting your conclusions must not be omitted. Instead, it is worthwhile to investigate their origin and, if no explanation can be found, to state this clearly. Any anomalies in the data, e. g., those contradicting your or other's results, should be noted in the paper. It is then clear that their occurrence is not due to typographical or other error.

When reporting running times for heuristics without stopping criterion (such as local search) do report the total running time used by the heuristic, not just the time until the best solution was found. As Johnson [32] points out, considering only the time for the best solutions essentially means pretending clairvoyance of the heuristic, since it has no way to decide that no better solution will be found. The running time should also include time spent for preprocessing and setup, which should be given separately. Reporting the total running time of the heuristic gives a clear indication of the effort needed to get this solution and allows better comparison to competing methods.

For similar reasons, it is also desirable to report the total running time invested in your computational study, since omitting this time can give a distorted picture. For instance, if it took some time to find the parameters that make a heuristic perform well the real effort to get good solutions with this heuristic is much larger than just running the heuristic once with those parameters. A similar effort might be necessary to suit the heuristic to differently structured instances. It is also interesting to know how much you gained by tuning the parameters, i. e., you should indicate the typical solution quality before and after tuning.

When evaluating heuristics it is important to assess the quality of solutions, since this allows quantifying the time / quality tradeoff when using this heuristic. Preferably, you should compare the heuristic to exact solution values. If exact solutions turn out to be too expensive to compute, you may resort to good lower bounds which can be obtained by e. g., Linear Programming or Lagrangian relaxations. As a last resort, you can compare solution values to best-known ones or to those of other heuristics.

The purpose of heuristics is to produce hopefully good solutions in much shorter time than exact methods can. Complex heuristics may produce a sequence of improving best solutions. For these you should indicate how solution quality evolves in time. This can be done using diagrams, showing solution quality as a time series. Another possibility suggested by Barr et al. [4] is to use derived descriptive measures. They suggest the ratio

$$r_{0.05} = \frac{\text{time to within 5\% of best}}{\text{time to best found}},$$

which measures how fast the heuristic converges to its best attainable value. This metric is not suitable for comparing different algorithms, since the value of the best solution found may differ significantly.

Another interesting point to investigate is how solution quality changes with growing instance size. In fact, this is just a special case of *robustness*, which is discussed in Section 3.5: An algorithm should perform well on a large set of instances. Similarly, if the behavior of a heuristic depends on some parameters, its solution quality should not deteriorate with small changes of "good" parameter settings. The robustness of an algorithm should be addressed and reported, for example by giving standard deviations for the quality in a quality-time graph [4], indicating the spread of quality after a fixed computation time for the whole instance set studied.

To get a better understanding of a complex algorithm and its specific features, the contribution of each strategy or phase should be assessed and reported on.

You should try to look at more machine-independent running time measures, as suggested in Section 1.6 and account on these findings in detail.

It is also worthwhile to mention unsuccessful algorithmic ideas that you tried. This may save other researchers from spending further effort on them. For instance, if your heuristic was not able to find a feasible solution on a certain class of instances, this is something to report.

Draw Well-justified Conclusions and Look for Explanations. Reporting on an experimental study requires interpreting the data. It is clearly not sufficient to just describe the algorithm and to give a table of numbers. The data you provide in your report needs to be explained in a convincing and consistent way by suitable claims. Be sure to support your claims with convincing diagrams and tables. These must not hide any contradicting data; instead, you need to argue why they can be neglected for your claims. Of course your claims need to be supported by the data.

In order to support or challenge your claims, it may be worthwhile to employ statistical analysis (see Section 1.6.2). This can provide additional evidence and confidence or rejection for your claims.

As mentioned several times before and recommended in the literature [32,16], you should look at and report on as large instances as possible. Looking at huge instance sizes provides stronger support for claims, especially on asymptotic behavior. Moreover, the reader gets an impression on how the algorithms scale with problem size.

Present Data in Informative Ways. Large amounts of numbers are usually considered to be rather dull and boring. This need not be the case, however, it is necessary to present the data in interesting and revealing ways. Using appropriate diagrams and clearly-structured tables can help a lot here. See Section 1.7.2 for more detailed hints.

Statistical methods to support your claims are best used for general conclusions, such as recovering trends and correlations between variables. Usually, it is not interesting to do lots of experimental runs just to get tight confidence intervals, since these apply only to the specific setting.

You should also avoid reporting too much data. If you generated much data for many instances you should try to cluster similar instances and report representive results for each cluster. It is also possible to report averages and similar summary statistics (e. g., minimum and maximum, medians, quartiles) to get an impression of the results. The full data could be put in the appendix or made available electronically via the Internet. You can safely omit the results for dominated algorithms, but you should indicate in the paper that they are dominated and therefore dropped.

1.7.2 Presenting Data in Diagrams and Tables

Experimental studies usually yield large amounts of data which are in a sense the result of the study and thus need to be reported on in some sensible way. There are two ways to present that data: pictures (i. e., diagrams) and tables. Both have their advantages and drawbacks which will be discussed here. We also give advice on how to make best use out of them.

Diagrams are useful for recognizing patterns, trends, etc.; their use for analyzing data has already been discussed. They give a quick impression and quick overview and can make vast amounts of data comprehendable and ease comparison of different data sets. However, they tend to hide details (which is an advantage, too) and make it hard to figure out exact values. Tables, on the other hand, reporting the data as it is, although this might be hard to interpret.

The natural conclusion is to use tables for small amounts of data. Tufte [62, p. 56] recommends using tables for sets of 20 numbers or less. Tables may also be useful to report exact values for larger data sets in addition to some diagram. Larger tables are particularly out of place at oral presentations [44].

Tables When using tables (especially larger ones) it is important to structure them in order to highlight important information and aspects of the data [32]. Tables can often be made more accessible by choosing a sensible ordering of rows and columns. The sorting should reflect properties of the data. For instance, the rows in Table 1.2 on page 52 have been ordered according to the running time of the algorithms, which makes the the consistent ranking of the algorithms apparent. Similarly, it is better to sort instances by their size than their names.

Tables should not only give the data as measured, but also provide interesting related information contributing to the interpretation of the data. The obvious example is when you give a solution value and a lower bound, then you should include a column indicating the resulting optimality gap.

Of course, tables and the reported data need to be labeled properly. This encompasses stating the exact meaning of the rows and the columns and the units of quantities as well as further details important for interpretation. If you include numbers from different sources, try to make them comparable and indicate their origin.

Diagrams Most fundamental things for creating good diagrams have already been discussed in Section 1.6.1 since they are useful both for data analysis and presentation. We therefore focus on more detailed hints which become more important for reporting.

The general advice is to avoid too much information in one diagram. Although you as the expert for your experimental setup and analysis can probably cope with more information in one diagram, this same diagram may be too complicated for your audience. One issue might be too many data sets in a diagram, e. g., too many curves. The number of curves which can be displayed in a reasonable way depends on their overall complexity or information density. If the curves lie close together or you cannot tell on first sight which is above or below these are indications that you should think about improving the diagram.

The following hints on how to cope with too many curves have been collected by Sanders [55]. A first possibility is to consider different scaling of the axes as explained in Section 1.6.1 in order to find a better view on the data. It may be possible to remove dominated curves and to indicate that removal. In some cases, similar curves can be combined to a single one. For example, to show that an algorithm is always better than some other ones it suffices to plot the best result of all the other algorithms. Finally, you should consider decomposing a diagram into different ones with differing y-scales, both showing only a subset of the original plots.

Consistency in the diagram is important, since inconsistency is confusing and tends to distract the reader's attention to resolving that discrepancy. Consistency is reflected in many details. For example, if results for one algorithm are presented in several diagrams, be sure to use the same line and point styles and color for plots of that algorithm. Similarly, algorithm labels for corresponding plots should be in the top-down order of the plots.

The design of the diagram should be as clean as possible. You should use marks for data points which are clearly distinguishable, but not too large. Data points belonging to the same data set can be connected to better indicate that they belong together. However, as Johnson [32] points out, such lines implicitly suggest a trend and/or that interpolation between the data points is possible or sensible. Connecting the points should therefore be avoided if possible. If necessary, you should use unobtrusive (e. g., thin gray) straight lines to do this – splines are a no-no since they amplify the implicit "interpolation" claim.

There are some books on diagram design, for instance the book of Tufte [62]. He introduces the principle of data-ink maximization which essentially requires to make best use of the ink used to draw the diagram. For example, he suggests to avoid grids since they usually interfere too much with the data drawn. He also gives hints and examples on how to improve existing diagrams as well as inspiration to design new ones.

Finally, your diagrams need to be labelled clearly and completely. Ideally, they are understandable on their own, without having to read the corresponding text passages. To achieve this, you should try to succinctly provide all information needed for interpretation. At the least, you should explain or mention unusual axis scales (e. g., log, normalized), what has been measured and is displayed. You should highlight important features and any specialty of your diagram.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.
2. Mohammad M. Amini and Richard S. Barr. Network reoptimization algorithms: A statistically designed comparison. *ORSA Journal on Computing*, 5(4):395–409, 1993.
3. David L. Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2006.
4. Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G. C. Resende, and William R. Stewart Jr. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, 1995.
5. Holger Bast and Ingmar Weber. Don't compare averages. In Sotiris E. Nikoletseas, editor, *4th International Workshop on Experimental and Efficient Algorithms (WEA)*, number 3503 in Lecture Notes in Computer Science, pages 67–76. Springer, 2005.
6. Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Computing*, 25(6):1305–1317, 1996.
7. Marie Coffin and Matthew J. Saltzmann. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS Journal on Computing*, 12(1):24–44, 2000.
8. William J. Conover. *Practical Nonparametric Statistic.* John Wiley & Sons, 1980.
9. Harlan P. Crowder, Ron S. Dembo, and John M. Mulvey. Reporting computational experiments in mathematical programming. *Mathematical Programming*, 15:316–329, 1978.

10. Harlan P. Crowder, Ron S. Dembo, and John M. Mulvey. On reporting computational experiments with mathematical software. *ACM Transactions on Mathematical Software*, 5(2):193–203, 1979.
11. Angela Dean and Daniel Voss. *Design and Analysis of Experiments*. Springer Texts in Statistics. Springer, 1999.
12. DIMACS Implementation Challenges. http://dimacs.rutgers.edu/Challenges/, 2006.
13. DIMACS TSP challenge. http://www.research.att.com/~dsj/chtsp/, 2006.
14. Exploratory data analysis. http://www.itl.nist.gov/div898/handbook/eda/eda.htm, 2006.
15. Christodoulos A. Floudas and Panos M. Pardalos. *A Collection of Test Problems for Constrained Global Optimization Problems*, volume 455 of *Lecture Notes in Computer Science*. Springer, 1990.
16. Ian P. Gent, Stuart A. Grant, Ewen MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith, and Toby Walsh. How not to do it. Technical Report 97.27, School of Computer Studies, University of Leeds, May 1997.
17. Ian P. Gent and Toby Walsh. CSPLIB: a benchmark library for constraints. Technical Report APES-09-1999, Department of Computer Science, University of Strathclyde, Glasgow, 1999.
18. Ian P. Gent and Toby Walsh. CSPLIB: A benchmark library for constraints. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *Lecture Notes in Computer Science*, pages 480–481. Springer, 1999.
19. Andrew V. Goldberg and Bernard M. E. Moret. Combinatorial algorithms test sets [CATS]: The ACM/EATCS platform for experimental research. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland*, pages 913–914, 1999.
20. Bruce L. Golden and William R. Stewart. *The Traveling Salesman Problem – A Guided Tour of Combinatorial Optimization*, chapter Empirical analysis of heuristics, pages 207–249. John Wiley & Sons, 1985.
21. Harvey J. Greenberg. Computational testing: Why, how and how much. *ORSA Journal on Computing*, 2(1):94–97, 1990.
22. Nicholas G. Hall and Marc E. Posner. Generating experimental data for computational testing with machine scheduling applications. *Operations Research*, 49(7):854–865, 2001.
23. Susan Hert, Lutz Kettner, Tobias Polzin, and Guido Schäfer. ExpLab - a Tool Set for Computational Experiments. http://explab.sourceforge.net, 2003.
24. Benjamin Hiller, Sven Oliver Krumke, and Jörg Rambau. Reoptimization gaps versus model errors in online-dispatching of service units for ADAC. *Discrete Appl. Math.*, 154(13):1897–1907, 2006. Also available as ZIB-Report ZR 04-17.
25. Karla L. Hoffman and Richard H. F. Jackson. In pursuit of a methodology for testing mathematical programming software. In John M. Mulvey, editor, *Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981*, volume 199 of *Lecture Notes in Economics and Mathematical Systems*, pages 177–199. Springer, 1982.
26. Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63–91, 1993.
27. John N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.

28. John N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.
29. Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT 2000, Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283–292. IOS Press, 2000.
30. Richard H. F. Jackson, Paul T. Boggs, Stephen G. Nash, and Susan Powell. Guidelines for reporting results of computational experiments. Report of the ad hoc committee. *Mathematical Programming*, 49:413–425, 1991.
31. David Johnson and Lyle McGeoch. Experimental analysis of heuristics for the STSP. In Gutin and Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic Publishing, Dordrecht, 2002.
32. David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59 of *DIMACS Monographs*, pages 215–250, 2002.
33. Darwin Klingman, H. Albert Napier, and Joel Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20(5):814–821, 1974.
34. Donald E. Knuth. *The Stanford Graphbase: A Platform for Combinatorial Computing*. ACM Press, 1993.
35. Balakrishnan Krishnamurthy. Constructing test cases for partitioning heuristics. *IEEE Transactions on Computers*, 36(9):1112–1114, 1987.
36. Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana*, pages 370–379, 1997.
37. David Lane, Joan Lu, Camille Peres, and Emily Zitek. Online statistics: An interactive multimedia course of study. http://onlinestatbook.com/index.html, 2006.
38. Pierre L'Ecuyer. Simulation of algorithms for performance analysis. *INFORMS Journal on Computing*, 8(1):16–20, 1996.
39. Gideon Lidor. Construction of nonlinear programming test problems with known solution characteristics. In John M. Mulvey, editor, *Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981*, volume 199 of *Lecture Notes in Economics and Mathematical Systems*, pages 35–43. Springer, 1982.
40. Catherine C. McGeoch. Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24(2):195–212, 1992.
41. Catherine C. McGeoch. Challenges in algorithm simulation. *INFORMS Journal on Computing*, 8(1):27–28, 1996.
42. Catherine C. McGeoch. Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing*, 8(1):1–15, 1996.
43. Catherine C. McGeoch. Experimental analysis of algorithms. *Notices of the AMS*, 48(3):304–311, 2001.
44. Catherine C. McGeoch and Bernard M. E. Moret. How to present a paper on experimental work with algorithms. *SIGACT News*, 30(4):85–90, 1999.

45. Bernard M. E. Moret. Towards a discipline of experimental algorithmics. In Michael H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, volume 59 of *DIMACS Monographs*, pages 197–213. American Mathematical Society, 2002.
46. Bernard M. E. Moret and Henry D. Shapiro. Algorithms and experiments: The new (and old) methodology. *Journal of Universal Computer Science*, 7(5):434–446, 2001.
47. Bernhard M.E. Moret and Henry D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *Computational Support for Discrete Mathematics*, volume 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 99–117, 1994.
48. Richard P. O'Neill. A comparison of real-world linear programs and their randomly generated analogs. In John M. Mulvey, editor, *Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981*, volume 199 of *Lecture Notes in Economics and Mathematical Systems*, pages 44–59. Springer, 1982.
49. James B. Orlin. On experimental methods for algorithm simulation. *INFORMS Journal on Computing*, 8(1):21–23, 1996.
50. Ronald L. Rardin and Benjamin W. Lin. Test problems for computational experiments – Issues and techniques. In John M. Mulvey, editor, *Evaluating Mathematical Programming Techniques, Proceedings of a Conference held at the National Bureau of Standards, Boulder, Colorado, January 5–6, 1981*, volume 199 of *Lecture Notes in Economics and Mathematical Systems*, pages 8–15. Springer, 1982.
51. Gerhard Reinelt. TSPLIB—A traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
52. Neil Robertson and Paul D. Seymour. Graph minors. XIII: the disjoint paths problem. *J. Comb. Theory Ser. B*, 63(1):65–110, 1995.
53. Neil Robertson and Paul D. Seymour. Graph minors. XX. Wagner's conjecture. *J. Comb. Theory Ser. B*, 92(2):325–357, 2004.
54. Laura A. Sanchis. On the complexity of test case generation for NP-hard problems. *Information Processing Letters*, 36(3):135–140, 1990.
55. Peter Sanders. Presenting data from experiments in algorithmics. Number 2547 in Lecture Notes in Computer Science, pages 181–196. Springer, 2002.
56. David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* CRC Press, 2007.
57. Douglas R. Shier. On algorithm analysis. *INFORMS Journal on Computing*, 8(1):24–26, 1996.
58. Sidney Siegel. *Nonparametric Statistics for the Behavioral Sciences.* McGraw-Hill, 1956.
59. Peter Sprent and N. C. Smeeton. *Applied nonparametric statistical methods.* Chapman & Hall/CRC, 2001.
60. Geoff Sutcliffe and Christian B. Suttner. The TPTP problem library - CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
61. Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
62. Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 1983.
63. John W. Tukey. *Exploratory Data Analysis.* Reading, MA. Addison-Wesley, 1977.