8.1 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.

8.1.1 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.

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 behavior 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.

* 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.
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 [15].

**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 [675][676]. 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 $k$ [113]. 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 [586]. 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 [584]. 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