algorithms

Article

Guidelines for Experimental Algorithmics: A Case Study in Network Analysis

Eugenio Angriman [1,†], Alexander van der Grinten [1,†], Moritz von Looz [1,†], Henning Meyerhenke [1,*,†], Martin Nöllenburg [2,†], Maria Predari [1,†] and Charilaos Tzovas [1,†]

[1] Department of Computer Science, Humboldt-Universität zu Berlin, 10099 Berlin, Germany
[2] Algorithms and Complexity Group, Institute of Logic and Computation, TU Wien, 1040 Vienna, Austria
* Correspondence: meyerhenke@hu-berlin.de
† These authors contributed equally to this work.

Received: 1 June 2019; Accepted: 19 June 2019; Published: 26 June 2019

Abstract: The field of network science is a highly interdisciplinary area; for the empirical analysis of network data, it draws algorithmic methodologies from several research fields. Hence, research procedures and descriptions of the technical results often differ, sometimes widely. In this paper we focus on *methodologies* for the experimental part of algorithm engineering for network analysis—an important ingredient for a research area with empirical focus. More precisely, we unify and adapt existing recommendations from different fields and propose universal guidelines—including statistical analyses—for the systematic evaluation of network analysis algorithms. This way, the behavior of newly proposed algorithms can be properly assessed and comparisons to existing solutions become meaningful. Moreover, as the main technical contribution, we provide SimexPal, a highly automated tool to perform and analyze experiments following our guidelines. To illustrate the merits of SimexPal and our guidelines, we apply them in a case study: we design, perform, visualize and evaluate experiments of a recent algorithm for approximating betweenness centrality, an important problem in network analysis. In summary, both our guidelines and SimexPal shall modernize and complement previous efforts in experimental algorithmics; they are not only useful for network analysis, but also in related contexts.

Keywords: experimental algorithmics; network analysis; applied graph algorithms; statistical analysis of algorithms

---

1. Introduction

The traditional algorithm development process in theoretical computer science typically involves (i) algorithm design based on abstract and simplified models and (ii) analyzing the algorithm's behavior within these models using analytical techniques. This usually leads to asymptotic results, mostly regarding the worst-case performance of the algorithm. (While average-case [1] and smoothed analysis [2] exist and have gained some popularity, worst-case bounds still make up the vast majority of (e.g., running time) results.) Such worst-case results are, however, not necessarily representative for algorithmic behavior in real-world situations [3], both for $\mathcal{NP}$-complete problems [4,5] and poly-time ones [6,7]. In case of such a discrepancy, deciding upon the best-fitted algorithm solely based on worst-case bounds is ill-advised.

Algorithm engineering has been established to overcome such pitfalls [8–10]. In essence, algorithm engineering is a cyclic process that consists of five iterative phases: (i) modeling the problem (which usually stems from real-world applications), (ii) designing an algorithm, (iii) analyzing it theoretically, (iv) implementing it, and (v) evaluating it via systematic experiments (also known as *experimental algorithmics*). Note that not all phases have to be reiterated necessarily in every cycle [11]. This cyclic approach aims at a symbiosis: the experimental results shall yield insights that lead to further theoretical improvements and vice versa. Ideally, algorithm engineering results in algorithms that are asymptotically optimal *and* have excellent behavior in practice at the same time. Numerous examples where surprisingly large improvements could be made through algorithm engineering exist, e.g., routing in road networks [12] and mathematical programming [5].

In this paper, we investigate and provide guidance on the experimental algorithmics part of algorithm engineering—from a network analysis viewpoint. It seems instructive to view network analysis, a subfield of *network science* [13], from two perspectives: on the one hand, it is a collection of methods that study the structural and algorithmic aspects of networks (and how these aspects affect the underlying application). The research focus here is on efficient algorithmic methods. On the other hand, network analysis can be the process of interpreting network data using the above methods. We briefly touch upon the latter; yet, this paper's focus is on experimental evaluation methodology, in particular regarding the underlying (graph) algorithms developed as part of the network analysis toolbox. We use the terms *network* and *graph* interchangeably.)

In this view, network analysis constitutes a subarea of empirical graph algorithmics and statistical analysis (with the curtailment that networks constitute a particular data type) [13]. This implies that, like general statistics, network science is not tied to any particular application area. Indeed, since networks are abstract models of various real-world phenomena, network science draws applications from very diverse scientific fields such as social science, physics, biology and computer science [14]. It is interdisciplinary and all fields at the interface have their own expertise and methodology. The description of algorithms and their theoretical and experimental analysis often differ, sometimes widely—depending on the target community. We believe that uniform guidelines would help with respect to comparability and systematic presentation. That is why we consider our work (although it has limited algorithmic novelty) important for the field of network analysis (as well as network science and empirical algorithmics in general). After all, emerging scientific fields should develop their own best practices.

To stay focused, we concentrate on providing *guidelines for the experimental algorithmics* part of the algorithm engineering cycle—with emphasis on graph algorithms for network analysis. To this end, we combine existing recommendations from fields such as combinatorial optimization, statistical analysis as well as data mining/machine learning and adjust them to fit the idiosyncrasies of networks. Furthermore, and as main technical contribution, we provide SimexPal, a highly automated tool to perform and analyze experiments following our guidelines. For illustration purposes, we use this tool in a case study—the experimental evaluation of a recent algorithm for approximating *betweenness centrality*, a well-known network analysis task. The target audience we envision consists of network analysts who develop algorithms and evaluate them empirically. Experienced undergraduates and young PhD students will probably benefit most, but even experienced researchers in the community may benefit substantially from SimexPal.

2. Common Pitfalls (And How to Avoid Them)

2.1. Common Pitfalls

Let us first of all consider a few pitfalls to avoid. (Such pitfalls are probably more often on a reviewer's desk than one might think. Note that we do not claim that we never stumbled into such pitfalls ourselves, nor that we always followed our guidelines in the past.) We leave problems in modeling the underlying real-world problem aside and focus on the evaluation of the algorithmic method for the problem at hand. Instead, we discuss a few examples from two main categories of pitfalls: (i) inadequate justification of claims on the paper's results and (ii) repeatability/replicability/reproducibility issues (the terms will be explained below; for more detailed context please visit ACM's webpage on artifact review and badging: https://www.acm.org/publications/policies/artifact-review-badging).

Clearly, a paper's claims regarding the algorithm's empirical behavior need an adequate justification by the experimental results and/or their presentation. Issues in this regard may happen for a variety of reasons; not uncommon is a lack of instances in terms of their number, variety and/or size. An inadequate literature search may lead to not comparing against the current state of the art or if doing so, choosing unsuitable parameters. Additionally noteworthy is an inappropriate usage of statistics. For example, arithmetic averages over a large set of instances might be skewed towards the more difficult instances. Reporting only averages would not be sufficient then. Similarly, if the difference between two algorithms is small, it is hard to decide whether one of the algorithms truly performs better than the other one or if the perceived difference is only due to noise. Even if no outright mistakes are made, potential significance can be wasted: Coffin and Saltzmann [15] discuss papers whose claims could have been strengthened by an appropriate statistical analysis, even without gathering new experimental data.

Reproducibility (the algorithm is implemented and evaluated independently by a different team) is a major cornerstone of science and should receive sufficient attention. Weaker notions include *replicability* (different team, same source code and experimental setup) and *repeatability* (same team, same source code and experimental setup). An important weakness of a paper would thus be if the description does not allow the (independent) reproduction of the results. First of all, this means that the proposed algorithm needs to be explained adequately. If the source code is not public, it is all the more crucial to explain all important implementation aspects—the reader may judge whether this is the current community standard. Pseudocode allows to reason about correctness and asymptotic time and space complexity and is thus very important. But the empirical running time of a reimplementation may deviate from the expectation when the paper omitted crucial implementation details.

Even the repetition of one's own results (which is mandated by major funding organizations for time spans such as 10 years) can become cumbersome if not considered from the beginning. To make this task easier, not only source code needs to documented properly, but also input and output data of experiments as well as the scripts containing the parameters.

2.2. Outline of the Paper

How do we avoid such pitfalls? We make this clear by means of a use case featuring betweenness approximation; it is detailed in Section 3. A good start when designing experiments is to formulate a hypothesis (or several ones) on the algorithm's behavior (see Section 4.1). This approach is not only part of the scientific method; it also helps in structuring the experimental design and evaluation, thereby decreasing the likelihood of some pitfalls. Section 4.2 deals with how to select and describe input instances to support a certain generality of the algorithmic result. With a similar motivation, Section 4.3 provides guidance on *how many* instances should be selected. It also discusses how to deal with variance in case of non-determinism, e.g., by stating how often experiments should be repeated.

If the algorithm contains a reasonable amount of tunable parameters, a division of the instances into a tuning set and an evaluation set is advisable, which is discussed in Section 4.4.

In order to compare against the best competitors, one must have defined which algorithms and/or software tools constitute the state of the art. Important aspects of this procedure are discussed in Section 4.5. One aspect can be the approximation quality: while an algorithm may have the better quality guarantee in theory, its empirical solution quality may be worse—which justifies to consider not only the theoretical reference in experiments. The claim of superiority refers to certain measures—typically related to resources such as running time or memory and/or related to solution quality. Several common measures of this kind, their performance indicators, and how to deal with them are explained in Section 4.6. Some of them are hardware-independent, many of them are not.

A good experimental design can take you far—but it is not sufficient if the experimental pipeline is not efficient or lacks clarity. In the first case, obtaining the results may be tedious and time-consuming.

Or the experiments simply consume more computing resources than necessary and take too long to generate the results. In the second case, in turn, the way experimental data are generated or stored may not allow easy reproducibility. Guidelines on how to setup your experimental pipeline and how to avoid these pitfalls are thus presented in Section 5. The respective subsections deal with implementation issues (Section 5.1), repeatability/replicability/reproducibility (Section 5.2), job submission (Section 5.3), output file structure for long-term storage (Section 5.4), retrieval and aggregation of results (Section 5.5).

As mentioned, betweenness approximation and more precisely the KADABRA [16] algorithm will serve as our prime example. We have implemented this algorithm as part of the NetworKit toolbox [17]. As part of our experimental evaluation, a meaningful visualization (Section 6) of the results highlights many major outcomes to the human reader. Since visualization is typically not enough to show statistical significance in a rigorous manner, an appropriate statistical analysis (Section 7) is recommended. Both, visualization and statistical analysis, shall lead to a justified interpretation of the results.

3. Use Case

Typically, algorithmic network analysis papers (i) contribute a new (graph) algorithm for an already known problem that improves on the state of the art in some respect or (ii) they define a new algorithmic problem and present a method for its solution. To consider a concrete example contribution, we turn towards betweenness centrality, a widely used and very popular measure for ranking nodes (or edges) in terms of their structural position in the graph [18].

3.1. Betweenness Approximation as Concrete Example

Betweenness centrality [19] measures the participation of nodes in shortest paths. More precisely, let $G = (V, E)$ be a graph with $|V| = n$ vertices and $|E| = m$ edges; the (normalized) betweenness centrality of a node $v \in V$ is defined as

$$\boldsymbol{b}(v) := \frac{1}{n(n-1)} \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{1}$$

where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that cross v (as intermediate node). Computing the exact betweenness scores for all nodes of an unweighted graph can be done in $\mathcal{O}(nm)$ time with Brandes's algorithm [20]. Since this complexity is usually too high for graphs with millions of nodes and edges, several approximation algorithms have been devised [16,21–24]. These algorithms trade solution quality for speed and can be much faster.

For illustration purposes, we put ourselves now in the shoes of the authors of the most recent of the cited algorithms, which is called KADABRA [16]: we describe (some of) the necessary steps in the process of writing an algorithm engineering paper (let us emphasize that the authors of the original KADABRA paper did not do a bad job in their presentation of the results—*au contraire*. Our main reasons for selecting KADABRA are (i) the high likelihood that the betweenness problem is already known to the reader due to its popularity in the field and (ii) the fact that approximation algorithms display important aspects of experimental algorithmics quite clearly)—with a focus on the design and evaluation of the experiments.

3.2. Overview of the KADABRA Algorithm

The KADABRA algorithm approximates the betweenness centrality of (un)directed graphs within a given absolute error of at most ϵ with probability $(1 - \delta)$ [16]. The main rationale of the algorithm is to iteratively select two nodes s, t uniformly at random and then sample a shortest path π from

s to t (again uniformly at random). This leads to a sequence of randomly selected shortest paths $\pi_1, \pi_2, \ldots, \pi_\tau$. The betweenness centrality of each node $v \in V$ is then estimated as:

$$\widetilde{\boldsymbol{b}}(v) = \frac{1}{\tau} \sum_{i=1}^{\tau} \widetilde{\boldsymbol{x}}_i(v), \tag{2}$$

where $\widetilde{\boldsymbol{x}}_i$ is 1 iff v occurs in π_i and 0 otherwise.

Compared to earlier approximation algorithms that employ similar sampling techniques (e.g., [23]), the novelty of KADABRA relies on the clever stopping condition, used to determine the number of rounds τ. Clearly, there is a number ω, depending on the input size but not the graph itself, such that if $\tau \geq \omega$, the algorithm achieves the desired solution quality with probability $(1 - \delta)$ (for example, if ω is chosen so that almost all vertex pairs are sampled. In reality, ω can be chosen to be much smaller). KADABRA, however, avoids to run a fixed number of ω rounds by using *adaptive sampling*. At each round of the algorithm, it is guaranteed that

$$P\left(\boldsymbol{b}(v) \leq \widetilde{\boldsymbol{b}}(v) - f\right) \leq \delta_L(v) \quad \text{and} \quad P\left(\boldsymbol{b}(v) \geq \widetilde{\boldsymbol{b}}(v) + g\right) \leq \delta_U(v), \tag{3}$$

where $P(\cdot)$ denotes the probability and $f = f(\widetilde{\boldsymbol{b}}(v), \delta_L(v), \omega, \tau)$ and $g = g(\widetilde{\boldsymbol{b}}(v), \delta_U(v), \omega, \tau)$ are (rather lengthy) expressions depending on $\widetilde{\boldsymbol{b}}(v)$, per-vertex probabilities $\delta_L(v)$ and $\delta_U(v)$, the current number of rounds τ and the static round count ω. $\delta_L(v)$ and $\delta_U(v)$ are chosen such that $\sum_{v \in V} \delta_L(v) + \delta_U(v) \leq \frac{\delta}{2}$. Once $f, g < \epsilon$ during some round, the algorithm terminates. Note also that for each round, the algorithm draws a number of samples and performs occurrence updates without checking the stopping condition. This number is determined by parameter c. In the original implementation of KADABRA (but not reported in the paper), c is fixed to 10 (without further explanation).) Algorithm 1 displays the corresponding pseudocode with some expository comments. Besides adaptive sampling, KADABRA relies on a *balanced bidirectional BFS* to sample shortest paths. For details we refer the interested reader to the original paper.

Algorithm 1 KADABRA algorithm (absolute error variant)

```
1:  procedure KADABRA($G = (V, E), \epsilon, \delta$)
2:      $\omega \leftarrow$ non-adaptive number of iterations
3:      compute $\delta_L, \delta_U$ from $\delta$                ▷ Requires upper bound of the diameter.
4:      $\tau \leftarrow 0$                                    ▷ Number of sampled paths.
5:      for all $v \in V$ do
6:          $\widetilde{x}(v) \leftarrow 0$                    ▷ Occurrences of $v$ in sampled paths.
7:      end for
8:      while $\tau < \omega$ do
9:          if $f(\widetilde{x}(v)/\tau,\ \delta_L(v),\ \omega,\ \tau) < \epsilon$ and $g(\widetilde{x}(v)/\tau,\ \delta_U(v),\ \omega,\ \tau) < \epsilon$ then
10:             break                                          ▷ Approximation is already good enough.
11:         end if
12:         for $i \in \{1, \ldots, c\}$ do                    ▷ Draw $c$ samples per round.
13:             $s, t \leftarrow$ samplePair($G$)              ▷ Uniformly at random.
14:             $\pi \leftarrow$ sampleShortestPath($G, s, t$) ▷ Uniformly at random.
15:             for all $v \in \pi$ do
16:                 $\widetilde{x}(v) \leftarrow \widetilde{x}(v) + 1$
17:             end for
18:             $\tau \leftarrow \tau + 1$
19:         end for
20:     end while
21:     return $\widetilde{x}/\tau$                            ▷ Betweenness centrality is estimated as: $\widetilde{\boldsymbol{b}} = \widetilde{x}/\tau$.
22: end procedure
```

4. Guidelines for the Experimental Design

Now, we want to set up experiments that give decisive and convincing empirical evidence that KADABRA is better than the state of the art. This section discusses the most common individual steps of this process.

4.1. Determining Your Hypotheses

Experiments are mainly used for two reasons; as an exploratory tool to reveal unknown properties and/or as a means to answer specific questions regarding the proposed algorithm. The latter suggests the development of certain hypotheses on the behavior of our algorithm that can be confirmed or contradicted by experiments. In the following, we group common hypotheses of algorithm engineering papers in network analysis into two categories:

1. Hypotheses on how our algorithm performs compared to the state of the art in one of the following metrics: running time, solution quality or (less often) memory usage. Ideally, an algorithm is deemed successful when it dominates existing algorithms in terms of these three metrics. However, in practice, we are usually content with algorithms that exhibit a good tradeoff between two metrics, often running time performance and solution quality. As an example, in real-time applications, we may be willing to sacrifice the solution quality (up to a certain threshold), in order to meet running time bounds crucial for the application.
2. Hypotheses on how the input instances or a problem-/algorithm-specific parameter affects our algorithm in terms of the aforementioned metrics. For example, a new algorithm might only outperform the existing algorithms on a certain type of graphs or an approximation algorithm might only be fast(er) if the desired approximation quality is within a certain range. If a hypothesis involves such restrictions, it should still be general enough for the algorithmic result to be relevant—overtuning of algorithms to uninteresting classes of inputs should be avoided. Other aspects of investigation may be decoupled from the state of the art to some extent and explore the connection between instance and algorithm properties: for instance, a hypothesis on how the empirical approximation error behaves over a range of input sizes.

KADABRA Example

In the context of our betweenness approximation study, we formulate three basic hypotheses:

1. KADABRA has a better running time and scalability (with respect to the graph size) than other algorithms, specifically the main competitor RK [23].
2. There is a significant difference between the solution quality of KADABRA and that of RK. (In Section 4.6, we explain how to evaluate the solution quality for approximation algorithms in more detail).
3. The diameter of input graphs has an effect on the running time of KADABRA: The KADABRA algorithm computes the betweenness values faster for graphs with low diameter than for graphs with large diameter.

The first two hypotheses belong to the first category, since we compare our implementation of the KADABRA algorithm to related algorithms. The other hypothesis belongs to the second category and is related to the performance of the KADABRA algorithm itself. Namely, we test how a data-specific parameter may influence the running time in practice. To evaluate these hypotheses experimentally, we need to select instances with both low and high diameter values, of course.

4.2. Gathering Instances

Selecting appropriate data sets for an experimental evaluation is a crucial design step. For sake of comparability, a general rule is to gather data sets from public sources; for network analysis purposes, well-known repositories are KONECT [25], SNAP [26], DIMACS10 [27], SuiteSparse [28], LWA [29] and Network Repository [30].

An appropriate data collection contains the type(s) of networks which the algorithm is designed for, and, for every type, a large enough number of instances to support our conclusions [8]. For instance, the data collection of an influence maximization algorithm should include social networks rather than street networks; a data collection to evaluate distributed algorithms should include instances that exceed the memory of a typical shared-memory machine.

The selection of an appropriate subset of instances for our experimental analysis is simpler if we first categorize the available networks in different classes. There exist several ways to do this: first, we can simply distinguish real-world and synthetic (=(randomly) generated) networks. Even though network analysis algorithms generally target real-world networks, one should also use synthetic networks, e.g., to study the asymptotic scalability of an algorithm, since we can easily generate similar synthetic networks of different scales.

Classification of real-world networks generally follows the phenomena they are modeling. Common examples are social networks, hyperlink networks or citation networks, which also fall under the umbrella of *complex networks*. (As the name suggests, complex networks have highly non-trivial (complex) topological features. Most notably, such features include a small diameter (*small-world effect*) and a skewed degree distribution (many vertices with low degree and a few vertices with high degree)). Other examples of real-world networks are certain infrastructure networks such as road networks. If our algorithm targets real-world data, we want to carefully build a diverse collection of instances that is representative of the graph classes we are targeting.

Another interesting classification is one based on specific topological features of the input data such as the diameter, the clustering coefficient, the triangle count, etc. Classifying based on a certain property and reporting it should definitely be done when the property in question is relevant to the algorithm and may have an impact on its behavior. For instance, reporting the clustering coefficient could help with the interpretation of results concerning algorithms that perform triangle counting.

A source of instances especially suited for scalability experiments are synthetic graphs generated with graph generators [31]. An advantage is easier handling especially for large graphs, since no large files need to be transferred and stored. In addition, desired properties can often be specified in advance and in some models, a ground truth beneficial to test analysis algorithms is available [32]. Important drawbacks include a lack of realism, especially when choosing an unsuitable generative model.

KADABRA Example

In a real-world scenario, we want to show the improvement of the KADABRA algorithm with respect to the competition. Clearly, the graph class of highest relevance should be most prominent in our data collection. With this objective in mind, we show a collection for the KADABRA experiments in Table 1; this table also reports the diameter of each graph, along with its size and application class. We include the diameter because it is part of our hypotheses, i.e., the performance of KADABRA depends on the diameter of the input graph (see Section 4.1). In a different scenario, such as triangle counting or community detection, a more pertinent choice would be the average local clustering coefficient. Finally, we focus on complex networks, but include a minority of non-complex infrastructure networks. All of these instances were gathered from one of the aforementioned public repositories.

Table 1. Graphs (taken from KONECT) for our evaluation of the betweenness approximation algorithms.

Network Name	# of Nodes	# of Edges	Diameter	Class
moreno_blogs	1 224	16 715	8	Hyperlink
petster-hamster	2 426	16 631	10	Social
ego-facebook	2 888	2 981	9	Social
openflights	3 425	19 256	13	Infrastructure
opsahl-powergrid	4 941	6 594	46	Infrastructure
p2p-Gnutella08	6 301	20 777	9	Peer-to-peer
advogato	6 539	39 285	9	Social
wiki-Vote	7 115	100 762	7	Social
p2p-Gnutella05	8 846	31 839	9	Peer-to-peer
p2p-Gnutella04	10 876	39 994	10	Peer-to-peer
foldoc	13 356	91 471	8	Hyperlink
twin	14 274	20 573	25	Intl. Relations
cfinder-google	15 763	148 585	7	Hyperlink
ca-AstroPh	18 771	198 050	14	Coauthorship
ca-cit-HepTh	22 908	2 444 798	9	Coauthorship
subelj_cora	23 166	89 157	20	Citation
ego-twitter	23 370	32 831	15	Social
ego-gplus	23 628	39 194	8	Social
p2p-Gnutella24	26 518	65 369	11	Peer-to-peer
ca-cit-HepPh	28 093	3 148 447	9	Citation
cit-HepPh	34 546	420 877	14	Citation
facebook-wosn-wall	46 952	183 412	18	Social
edit-frwikibooks	47 905	139 141	8	Authorship
dblp-cite	49 789	49 759	2	Citation
loc-brightkite_edges	58 228	214 078	18	Social
edit-frwikinews	59 546	157 970	7	Authorship
dimacs9-BAY	321 270	397 415	837	Road
dimacs9-COL	435 666	521 200	1 255	Road
roadNet-PA	1 088 092	1 541 898	794	Road
roadNet-TX	1 379 917	1 921 660	1 064	Road

4.3. Scale of Experiments

Clearly, we cannot test all possible instances, even if they are small [33]. On the other hand, we want to draw significant conclusions about our new algorithm (and here not about a particular data set). This means that our experimental results should justify the later conclusions and allow to clearly distinguish between hypotheses. While Section 4.2 discusses *which* instances to select, we proceed with the question *"how many?"*. In addition, if the results are affected by randomness, how many repeated runs should we make for each?

Too few instances may be insufficient to support a conclusion. Plots then look sparse, hypothesis tests (Section 7.3.1) are inconclusive and inferred parameters are highly uncertain, which is reflected in unusably wide confidence and credible intervals (Sections 7.3.2 and 7.4). More instances are always better, but as a very rough rule of thumb reflecting community custom, we recommend at least 12–15 real-world instances for an experimental paper. When using synthetically generated graphs, use different generators and varying parameters to avoid a set of overly similar instances. If you want to support specific conclusions about algorithmic behavior within a subclass of instances, make sure to have that many instances *in that subclass*. Yet, the noisier the measurements and the stronger the differences of output between instances, the more experiments are needed for a meaningful (statistical) analysis. More formally, the uncertainty of many test statistics and inferred parameters (Most importantly the t-statistic used in confidence intervals and t-tests. In Bayesian statistics, the marginal posterior distribution of a parameter with a Gaussian likelihood also follows a t-distribution, at least when using a conjugate prior [34].) scales with $\sqrt{\frac{s^2}{n}}$, where n is the number of samples and s^2 is the sample variance. Generally speaking, if we want a result that is twice as precise, we need to quadruple the number of measurements.

The same applies for the number of repetitions for each instance. When plotting results affected by randomness, it is often recommended to average over the results of multiple runs. The expected deviation of such an average from the true mean The mean that would result given infinite samples.) is called the *standard error* and also scales with $\sqrt{\frac{s^2}{n}}$ [35]. For plotting purposes, we suggest that this standard error for the value of one plotted point (i.e., the size of the error bars) should be one order of magnitude smaller than the variance *between* plotted points. In the companion notebook *Plot-Smoother*, we give an example of code which calculates the number of repetitions required for a desired smoothness. Frameworks exist that automate the repeated measurements, for example Google Benchmark (https://github.com/google/benchmark).

In the context of binary hypothesis tests, the *statistical power* of a test is the probability that it correctly distinguishes between hypotheses, i.e., rejects a false null hypothesis in the classical approach. A larger sample size, a stronger effect or less variance in measurements all lead to a more powerful test. For further reading, see the extensive literature about statistical power [36,37].

All these calculations for the number of necessary experiments require an estimate of the variance of results—which might be unavailable before conducting the experiments. It might thus be necessary to start with a small initial set of instances to plan and prepare for the main experiments. Otherwise called *pilot study*, as opposed to the main experiments called *workhorse study*, see [38,39].

Guideline 1. More instances are better, use at least 12–15. To smooth out random fluctuations in running time and results, the number of required repetitions scales with the square of the desired precision.

4.4. Parameter Tuning

Experiments are not only a necessary ingredient to validate our hypothesis, but also a helpful discovery tool in a previous stage [10]. Some algorithm's behavior depends on tunable parameters. For example, the well-known hybrid implementation of quicksort switches between insertion sort and quicksort depending on whether the current array size is above or below a tunable threshold M [40]. An example from network analysis includes a plethora of approximation methods that calculate some centrality measure using sampling [22,23,41]. In such cases the number of selected samples may highly impact the performance of the algorithm. Experiments are thus used to determine the adequate number of samples that achieves a good approximation.

The KADABRA algorithm consists of two phases, in which bounds for a sampling process are first computed (Line 3 in Algorithm 1 of Section 3.2) and then used (when checking the stopping condition in Line 9). Tighter bounds are more expensive to compute but save time in the later phase; their quality is thus an important tradeoff. When evaluating a newly developed algorithm, there is the risk (or even temptation) to tune its parameters for optimal performance in the experiments while using general default parameters for all competitors. This may often happen with no ill intent—researchers usually know their own algorithm best. To ensure generalizability of results, we recommend to split the set of instances into a *tuning set* and an *evaluation set*. The tuning set is used to find good tuning parameters for all algorithms, while the main experiments are performed on the evaluation set. Only results from the evaluation set should be used to draw and support conclusions. Note that the tuning set can also be used to derive an initial variance estimate, as discussed in Section 4.3.

In each round of the KADABRA algorithm, a number of random vertex pairs is selected and the shortest path distance for each pair is randomly sampled. The number of pairs that are selected corresponds to the parameter c in the implementation (Line 12 in Algorithm 1) and affects the running time of the algorithm. Here, we run KADABRA in the tuning set by doing a binary search in order to find the best value for parameter c. The best average running time was found for parameter value 4375.0, as can be seen in Figure 1, and all the experiments in the evaluation used this value.

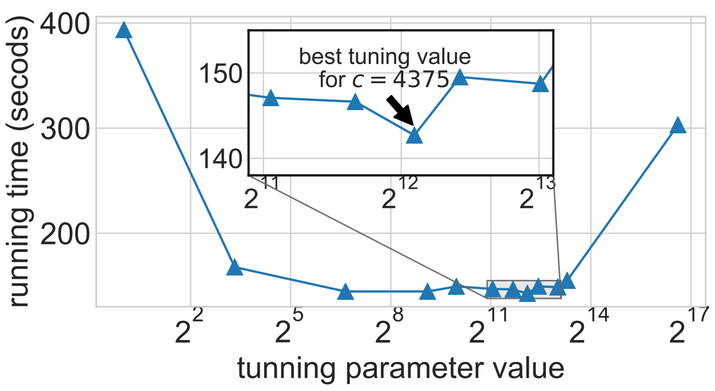


Figure 1. Tuning parameter search. Every point is the (arithmetic) mean running time for all the instances in the tuning set. Experiments on each instance are repeated 3 times for the respective value of the parameter. The value that gives the best results is picked; here it is 4375.0 as it provides the lowest running time of about 165.2 s.

How to Create the Tuning Set and the Evaluation Set

The tuning set should be structurally similar to the whole data set, so that parameter tuning yields good general performance and is representative for the evaluation set. (Due to symmetry, this also requires that the tuning set is structurally similar to the evaluation set). Tuning and evaluation sets should be disjoint. For large data sets, simple random sampling (i.e., simply picking instances from the data set uniformly at random and without replacement) yields such a representative split in expectation. Stratified sampling (i.e., partitioning the data set, applying simple random sampling to each partition and merging the result) [42] guarantees it.

In our example, note that our data set is already partitioned into different classes of networks (hyperlink, social, infrastructure, etc.), and that those classes are reasonably balanced (see Table 2). We can thus select a certain fraction of instances in each network class as tuning set. In case of computationally expensive tuning, it is advantageous to keep the tuning set small, for large data sets it can be much smaller than the evaluation set. In our example, we select a single instance per network class, as seen in Table 3.

These considerations are similar to the creation of training, test and validation sets in machine learning, which has spawned sophisticated methods like statistical learning theory for this division of datasets [43–45]. The intuitions do not match exactly, though: Our tuning step is to select high-level parameters for the compared algorithms, perhaps comparable to a combined training and testing step to select good learning parameters. While some sampling methods remain applicable, more sophisticated methods optimize for different objectives and are thus not considered here.

Table 2. Network class frequency of the data set in Table 1.

Class	Frequency
Social	26.67%
Citation	13.33%
Peer-to-peer	13.33%
Road	13.33%
Hyperlink	10.00%
Infrastructure	6.67%
Coauthorship	6.67%
Authorship	6.67%
Intl. relations	3.33%

Table 3. Instances are split into tuning and evaluation set.

Tuning Set		
Network Name	$\vert V\vert$	$\vert E\vert$
ego-facebook	2 888	2 981
advogato	6 539	39 285
foldoc	13 356	91 471
p2p-Gnutella24	26 518	65 369
dblp-cite	49 789	49 759
dimacs9-BAY	321 270	397 415
Evaluation Set		
Network Name	$\vert V\vert$	$\vert E\vert$
moreno_blogs	1 224	16 715
petster-hamster	2 426	16 631
openflights	3 425	19 256
opsahl-powergrid	4 941	6 594
p2p-Gnutella08	6 301	20 777
wiki-Vote	7 115	100 762
p2p-Gnutella05	8 846	31 839
p2p-Gnutella04	10 876	39 994
twin	14 274	20 573
cfinder-google	15 763	148 585
ca-AstroPh	18 771	198 050
ca-cit-HepTh	22 908	2 444 798
subelj_cora	23 166	89 157
ego-twitter	23 370	32 831
ego-gplus	23 628	39 194
ca-cit-HepPh	28 093	3 148 447
cit-HepPh	34 546	420 877
facebook-wosn-wall	46 952	183 412
edit-frwikibooks	47 905	139 141
loc-brightkite_edges	58 228	214 078
edit-frwikinews	59 546	157 970
dimacs9-COL	435 666	521 200

Guideline 2. When dealing with algorithms whose behavior depends on tunable parameters, use a *tuning set* to find good parameters, and use an *evaluation set* to draw your conclusions.

Guideline 3. In general, good performance in parameter tuning is achieved when the tuning set is structurally similar to the whole dataset. For large datasets, use random sampling techniques such as stratified sampling to create a representative tuning set.

4.5. Determining Your Competition

Competitor algorithms solve the same or a similar problem and are not dominated in the relevant metrics by other approaches in the literature. Since we focus on the design of the experimental pipeline, we consider only competitors that are deemed implementable. The best ones among them are considered the *state of the art* (SotA), here with respect to the criteria most relevant for our claims about KADABRA.

Other considerations are discussed next.

4.5.1. Unavailable Source Code

The source code of a competing algorithm may be unavailable and sometimes even the executable is not shared by its authors. If one can implement the competing algorithm with reasonable effort (to be weighted against the importance of the competitor), you should then do so. If this is not possible, a viable option is to compare experimental results with published data. For a fair comparison, the experimental settings should then be replicated as closely as possible. In order to avoid this scenario, we recommend open-source code; it offers better transparency and replicability (also see Section 5.2).

4.5.2. Solving a Different Problem

In some cases, the problem our algorithm solves is different from established problems and there is no previous algorithm solving it. Here, it is often still meaningful to compare against the SotA for the established problem. For example, if we consider the dynamic case of a static problem for the first time [46–48] or an approximation version for an exact problem [21,22,49] or provide the first parallel version of an algorithm [50]. Sometimes, this step requires to consider previous work that solves the same problem with an entirely different approach. This is easily forgotten; a recent paper on graph distance approximation [51], for example, compares against other embedding approaches only. Its case would have been much stronger, however, if the paper had also compared against a label-based algorithm [52] that efficiently gives exact results on graphs of similar size. Another example can be an optimization problem with a completely new constraint. While this may change optimal solutions dramatically compared to a previous formulation, a comparison to the SotA for the previous formulation may still be the best assessment. If the problem is so novel that no comparison is meaningful, however, there is no reason for an experimental comparison. (Nonetheless, experiments can still reveal empirical insights into the behavior of the new algorithm and can serve as basis for future comparisons).

4.5.3. Algorithmic and Implementation Improvements

Both the underlying algorithm and the efficiency of its implementation determine the running time of a program. When demonstrating an algorithmic improvement over a predecessor, it is thus necessary to compare against an *efficient implementation* of the state of the art. An example of someone taking this point seriously is the work of Penschuck [53]: Providing a new generation algorithm for hyperbolic random graphs, he first optimizes the implementations of his competitors (which includes a subset of the authors of this paper) to demonstrate which speedup was due to the improved algorithm alone.

4.5.4. Comparisons on Different Systems

Competitors may be designed to run on different systems. Recently, many algorithms take advantage of accelerators, mainly GPUs, and this trend has also affected the algorithmic network analysis community [54–56]. A direct comparison between CPU and GPU implementations is not necessarily meaningful due to different architectural characteristics and memory sizes. Yet, such a comparison may be the only choice if no other GPU implementation exists. In this case, one should compare against a multithreaded CPU implementation that should be tuned for high performance as well. Indirect comparisons can, and should, be done using system independent measures (Section 4.6).

4.5.5. KADABRA Example

The most relevant candidates to compare against KADABRA are the SotA algorithms RK [23] and ABRA [24]. The same algorithms are chosen as competitors in the original KADABRA paper. However, in our paper we focus only on a single comparison, the one between KADABRA and RK. This is done in order to highlight the main purpose of our work—to demonstrate the benefits of a thoughtful experimental design for meaningful comparisons of network-related algorithms. Reporting the actual results of such a comparison is of secondary importance. Furthermore, RK and ABRA exhibit similar behavior in the original KADABRA paper, with RK being overall slightly faster. Again, for the purpose of our paper, this is an adequate reason to choose RK over ABRA. (Of course, if we were to perform experiments for a new betweenness approximation algorithm, we would include all relevant solutions in the experiments).

Guideline 4. Broadening the competitor set and investing time to improve competing algorithms can (only) strengthen the experimental part of the paper.

4.6. Metrics and Performance Indicators

The most common metric for an algorithm's performance is its *running time*. Following [38], we use the term *performance indicator* to denote the quantities that are measured during a run of the algorithm. Hence, wall-clock time is a commonly used performance indicator for running time. For solution quality, the performance indicators are usually problem-specific; often, the gap between an algorithm's solution and the optimum is reported as a percentage (if known). In the following, we highlight situations that require more specific indicators. First, we discuss indicators for evaluating an algorithm's running time.

4.6.1. Running Time

CPU Time vs. Wall-Clock Time

For sequential algorithms, evaluations should prefer CPU time over wall-clock time. Indeed, this is the indicator for running time that we use in our KADABRA experiments. Compared to wall-clock time, CPU time is less influenced by external factors such as OS scheduling or other processes running on the same machine. The exception to this rule are algorithms that depend on those external factors such as external memory algorithms (where disregarding the time spent in system-level I/O routines would be unfair). In the same line of reasoning, evaluations of parallel algorithms would be based on wall-clock time as they depend on specifics of the OS scheduler.

Architecture-Specific Indicators

CPU and wall-clock times heavily depend on the microarchitecture of the CPU the algorithm is executed on. If comparability with data generated on similar CPUs from other vendors or microarchitectures is desired, other indicators should be taken into account. Such performance indicators can be accessed by utilizing the *performance monitoring* features of modern CPUs: these allow determining the number of executed instructions, the number of memory accesses (both are mostly independent of the CPU microarchitecture) or the number of cache misses (which depends on the cache size and cache strategy, but not necessarily the CPU model). If one is interested in how efficient algorithms are in exploiting the CPU architecture, the utilization of CPU components can be measured (example: the time spent on executing ALU instructions compared to the time that the ALU is stalled while waiting for results of previous instructions or memory fetches). Among some common open-source tools to access CPU performance counters are the `perf` profiler, `Oprofile` (http://oprofile.sourceforge.net/news/) (on linux only) and `CodeXL` (https://gpuopen.com/compute-product/codexl/).

System-Independent Indicators

It is desirable to compare different algorithms on the same system. However, in reality, this is not always possible, e.g., because systems required to run competitors are unavailable. For such cases, even if we do not run into such issues ourselves, we should also consider system-independent indicators. For example, this can be the speedup of a newly proposed algorithm against some base algorithm that can be tested on all systems. As an example, for betweenness centrality, some papers re-implement the Brandes algorithm [20] and compare their performance against it [24,46,57]. As this performance indicator is independent of the hardware, it can even be used to compare implementations on different systems, e.g., CPU versus GPU implementations. System-independent indicators also include algorithm-specific indicators, like the number of iterations of an algorithm or the number of edges visited. Those are particularly useful when similar algorithms are compared, e.g., if the algorithms are iterative and only differ in their choice of the stopping condition.

Aggregation and Algorithmic Speedup

Running time measurements are generally affected by fluctuations, so that the same experiment is repeated multiple times. To arrive at one value per instance, one usually computes the arithmetic mean over the experiments (unless the data are highly skewed). A comparison between the running times (or other metrics) of two algorithms A and B on different data sets may result in drastically different values, e.g., because the instances differ in size or complexity. For a concise evaluation, one is here also interested in aggregate values. In such a case it is recommended to aggregate over *ratios* of these metrics; regarding running time, this would mean to compute the *algorithmic speedup* (the *parallel speedup* of an algorithm A is instead the speedup of the parallel execution of A against its sequential execution, more precisely the ratio of the running time of the *fastest* sequential algorithm and the running time of the parallel algorithm. It can be used to analyze how efficiently an algorithm has been parallelized.) of A with respect to B. To achieve a fair comparison of the algorithmic aspects of A and B, the algorithmic speedup is often computed over their *sequential* executions [58]. In view of today's ubiquitous parallelism, this perspective may need to be reconsidered, though. To summarize multiple ratios, one can use the geometric mean [59]:

$$\mathrm{GM}(\text{speedup}) = \left(\prod_{i=1}^{\#\text{ of values}} \text{speedup on instance } i \right)^{\frac{1}{\#\text{ of values}}}$$

as it has the fundamental property that $\mathrm{GM}(\text{speedup}) = \frac{\mathrm{GM}(\text{running times of } A)}{\mathrm{GM}(\text{running times of } B)}$. Which mean is most appropriate for which aggregation is a matter of some discussion [60–62].

4.6.2. Solution Quality

Next, we discuss performance indicators for *solution quality*. Here, the correct measurements are naturally problem-specific; often, there is no single quality indicator but multiple orthogonal indicators are used.

Empirical vs. Worst-Case Solution Quality

As mentioned in the introduction, worst-case guarantees proven in theoretical models are rarely approached in real-world settings. For example, the accuracy of the ABRA algorithm for betweenness approximation has been observed to be always below the fixed absolute error, even in experiments where this was only guaranteed for 90% of all instances [24]. Thus, experimental comparisons should include also indicators for which theoretical guarantees are known.

Comparing Against (Unknown) Ground Truth

For many problems and instances beyond a certain size, *ground truth* (in the sense of the exact value of a centrality score or the true community structure of a network) is neither known nor feasible to compute. For betweenness centrality, however, AlGhamdi et al. [63] have computed exact scores for large graphs by investing into considerable supercomputing time. If such values are not available, lower and upper bounds on the true scores can sometimes be used to bound the quality of a solution. The latter is often done in the combinatorial optimization community, by comparing against bounds such as Held-Karp [64] on the traveling salesman problem (TSP). In other situations, the absence of ground truth or exact values, clearly requires the comparison to other algorithms in order to evaluate an algorithm's solution quality.

5. Guidelines for the Experimental Pipeline

Organizing and running all the required experiments can be a time-consuming activity, especially if not planned and carried out carefully. That is why we propose techniques and ideas to orchestrate this procedure efficiently. The experimental pipeline can be divided into four phases. In the first one, we finalize the algorithm's implementation as well as the scripts/code for the experiments themselves. It is important to use scripts or some external tool in order to automate the experimental pipeline. This also helps to reduce human errors and simplifies repeatability and replicability. Next, we submit the experiments for execution (even if the experiments are to be executed locally, we recommend to use some scheduling/batch system). In the third phase, the experiments run and create the output files. Finally, we parse the output files to gather the information about the relevant performance indicators.

5.1. Implementation Aspects

Techniques for implementing algorithms are beyond the scope for this paper; however, we give an overview of tooling that should be used for developing algorithmic code.

Source code should always be stored in version control systems (VCS); nowadays, the most commonly used VCS is Git [65]. For scientific experiments, a VCS should also be used to version scripts that drive experiments, valuable raw experimental data and evaluation scripts. Storing instance names and algorithm parameters of experiments in VCS is beneficial, e.g., when multiple iterations of experiments are done due to the AE cycle.

The usual principles for software testing (e.g., unit tests and assertions) should be applied to ensure that code behaves as expected. This is particularly important in growing projects where seemingly local changes can affect other project parts with which the developer is not very familiar. It is often advantageous to open-source code. (We acknowledge that open-sourcing code is not always possible, e.g., due to intellectual property or political reasons). The *Open Source Initiative* keeps a list (https://opensource.org/licenses/alphabetical) of approved open source licenses. An important difference is whether they require users to publish derived products under the same license. If code is open-sourced, we suggest well-known platforms like Github [66], Gitlab [67], or Bitbucket [68] to host it. An alternative is to use a VCS server within one's own organization, which reduces the dependence on commercial interests. In an academic context, a better accessibility can have the benefit of a higher scientific impact of the algorithms. For long-term archival storage, in turn, institutional repositories may be necessary.

Naturally, code should be well-structured and documented to encourage further scientific participation. Code documentation highly benefits from documentation generator tools such as Doxygen. (http://www.doxygen.nl.) *Profiling* is usually used to find bottlenecks and optimize implementations, e.g., using tools such as the `perf` profiler on Linux, Valgrind [69] or a commercial profiler such as VTune [70].

5.2. Repeatability, Replicability and Reproducibility

Terminology differs between venues; the Association of Computing Machinery defines *repeatability* as obtaining the same results when the same team repeats the experiments, *replicability* for a different team but the same programs and *reproducibility* for the case of a reimplementation by a different team. Our recommendations are mostly concerned with replicability.

In a perfect world scenario, the behavior of experiments is completely determined by their code version, command line arguments and configuration files. From that point of view, the ideal case for replicability, which is increasingly demanded by conferences and journals (for example, see the *Replicated Computational Results Initiative* of the *Journal on Experimental Algorithms*, http://jea.acm.org) in experimental algorithms, looks like this: A single executable program automatically downloads or generates the input files, compiles the programs, runs the experiments and recreates the plots and figures shown in the paper from the results.

Unfortunately, in reality some programs are non-deterministic and give different outputs for the same settings. If randomization is used, this problem is usually avoided by fixing an initial seed of a pseudo-random number generator. This seed is just another argument to the program and can be handled like all others. However, parallel programs might still cause genuine non-determinism in the output, e.g., if the computation depends on the order in which a large search space is explored [71,72] or on the order in which messages from other processors arrive. (As an example, some associative calculations are not associative when implemented with floating point numbers [73]. In such a case, the order of several, say, additions, matters). If these effects are of a magnitude that they affect the final result, these experiments need to be repeated sufficiently often to cover the distribution of outputs. A replication would then aim at showing that its achieved results are, while not identical, equivalent in practice. For a formal way to show such *practical equivalence*, see Section 7.4.1.

Implementations often depend on libraries or certain compiler versions. This might lead to complications in later replications when dependencies are no longer available or incompatible with modern systems. Providing a virtual machine image or container addresses this problem.

5.3. Running Experiments

Running experiments means to take care of many details: Instances need to be generated or downloaded, jobs need to be submitted to batch systems or executed locally, (note that the exact submission mechanism is beyond the scope of this paper, as it heavily depends on the batch system in question. Nevertheless, our guidelines and tooling suggestions can easily be adapted to all common batch system, such as Slurm (https://slurm.schedmd.com/) or PBS (https://www.pbspro.org/) running jobs need to be monitored, crashes need to be detected, crashing jobs need to be restarted without restarting all experiments, etc. To avoid human errors, improve reproducibility and accelerate those tasks, scripts and tooling should be employed.

To help with these recurring tasks, we provide as a supplement to this paper SimexPal, a command-line tool to automate the aforementioned tasks (among others). (SimexPal can be found at https://github.com/hu-macsy/simexpal). This tool allows the user to manage instances and experiments, launch jobs and monitor the status of those jobs. While our tool is not the only possible way to automate these tasks, we do hope that it improves over the state of writing custom scripts for each individual experiment. SimexPal is configured using a simple YAML [74] file and only requires a minimal amount of user-supplied code. To illustrate this concept, we give an example configuration in Figure 2. Here, `run` is the only piece of user-supplied code that needs to be written. `run` executes the algorithm and prints the output (e.g., running times) to `stdout`. Given such a configuration file, the graph instances can be downloaded using `simex instances download`. After that is done, jobs can be started using the command `simex experiments launch`. SimexPal takes care of not launching experiments twice and never overwrites existing output files. `simex experiments list` monitors the progress of all jobs. If a job crashes, `simex experiments purge` can be used to remove the corresponding output files. The next `launch` command will rerun that particular job.

Guideline 5. Use automation tools (e.g., simexpal or others) to manage experiments.

```
instances:
konect:
- 'advogato'
- 'ego-twitter'
- 'ego-facebook'
- 'ego-gplus'
# ... (more instances follow)
configurations:
- name: kadabra-1t
args: ['./run', '--threads=1', 'kadabra', '@INSTANCE@']
output: stdout
- name: kadabra-2t
args: ['./run', '--threads=2', 'kadabra', '@INSTANCE@']
output: stdout
# ... (more configurations follow)
- name: rk
args: ['./run', 'rk', '@INSTANCE@']
output: stdout
```

Figure 2. SimexPalconfiguration (`experiments.yml`) for KADABRA.

5.4. Structuring Output Files

Output files typically store three kinds of data: (i) *experimental results*, e.g., running times and indicators of solution quality, (ii) *metadata* that completely specify the parameters and the environment of the run, so that the run can be replicated (see Section 5.2), and (iii) *supplementary data*, e.g., the solution to the input problem that can be used to understand the algorithm's behavior and to verify its correctness. Care must be taken to ensure that output files are suitable for long-term archival storage (which is mandated by good scientific practices [75]). Furthermore, carefully designing output files helps to accelerate experiments by avoiding unnecessary runs that did not produce all relevant information (e.g., if the focus of experiments changes after exploration).

Choosing which experimental results to output is problem-specific but usually straightforward. For metadata, we recommend to include enough information to completely specify the executed algorithm version, its parameters and input instance, as well as the computing environment. This usually involves the following data: the *VCS commit hash* of the implementation and compiler(s) as well as all libraries, the implementation depends on name (or path) of the *input instance*, values of *parameters* of the algorithm (including *random seeds host name* of the machine and *current date and time*. (Experiments should never run uncommitted code. If there are any uncommitted changes, we suggest to print a comment to the output file to ensure that the experimental data in question does enter a paper). If the implementation's behavior can be controlled by a large number of parameters, it makes sense to print command line arguments as well as relevant environment variables and (excerpts from) configuration files to the output file. (Date and time help to identify the context of the experiments based on archived output data). Implementations that depend on hardware details (e.g., parallel algorithms or external-memory algorithms) want to log *CPU, GPU and/or memory configurations*, as well as *versions of relevant libraries* and drivers.

The relevance of different kinds of supplementary data is highly problem-dependent. Examples include (partial) solutions to the input problem, numbers of iterations that an algorithm performs, snapshots of the algorithm's state at key points during the execution or measurements of the time spent in different parts of the algorithm. Such supplementary data is useful to better understand an algorithm's behavior, to verify the correctness of the implementation or to increase confidence in the experimental results (i.e., that the running times or solution qualities reported in a paper are actually correct). If solutions are stored, automated correctness checks can be used to find and debug problems or to demonstrate that no such problems exist.

The *output format* itself should be chosen to be both *human readable* and *machine parsable*. Human readability is particularly important for long-term archival storage, as parsers for proprietary binary formats (and the knowledge of how to use them) can be lost over time. Machine parsability enables automated extraction of experimental results; this is preferable over manual extraction, which is inefficient and error-prone. Thus, we recommend structured data formats like YAML (or JSON [76]). Those formats can be easily understood by humans; furthermore, there is a large variety of libraries to process them in any commonly used programming language. If plain text files are used, we suggest to ensure that experimental results can be extracted by simple regular expressions or similar tools.

KADABRA Example

Let us apply these guidelines to our example of the KADABRA algorithm using SimexPal with the YAML file format. For each instance, we report KADABRA's running time, the values of the parameters ϵ and δ, the random seed that was used for the run, the number of iterations (i.e., samples) that the run required and the top-25 nodes of the resulting betweenness centrality ranking and their betweenness scores. The number 25 here is chosen arbitrarily, as a good balance between the amount of information that we store to verify the plausibility of the results and the amount of space consumed by the output. To fully identify the benchmark setting, we also report the hostname, our git commit hash, our random generator seed and the current date and time. Figure 3 gives an example how the resulting output file looks like.

Guideline 6. Use output file formats that are easy to read (by humans) and that can be parsed using existing libraries.

```
info:
commit: fef6c5ca
date: '2018-09-14T12:21:55.497368'
host: erle
iterations: 12598
parameters:
delta: 0.1
epsilon: 0.015
seed: 0
run_time: 1.6034371852874756
topk_nodes:
- 156
- 45
- 596
# ... (more nodes follow)
topk_scores:
- 0.0651690744562629
- 0.04643594221304969
- 0.0349261787585331
# ... (more scores follow)
```

Figure 3. Output of KADABRA example in YAML format.

5.5. Gathering Results

When the experiments are done, one has to verify that all runs were successful. Then, the output data has to be parsed to extract the desired experimental data for later evaluation. Again, we recommend the use of tools for parsing. In particular, SimexPal offers a Python package to collect output data. Figure 4 depicts a complete script that computes average running times for different algorithms on the same set of instances, using only seven lines of Python code. Note that SimexPal takes care of reading the output files and checking that all runs indeed succeeded (using the function `collect_successful_results()`).

```
import simexpal
import yaml
cfg = simexpal.config_for_dir('.')
res = cfg.collect_successful_results(yaml.load)
for algo in 'kadabra', 'rk':
rts = [r['run_time'] for r in res if r['algo'] == algo and r['threads'] == 1]
print(algo, sum(rts)/len(rts))
```

Figure 4. Script to collect KADABRA output.

In our example, we assume that the output files are formatted as YAML (thus we use the function `yaml.load()` to parse them). In case a custom format is used (e.g., when reading output from a competitor where the output format cannot be controlled), the user has to supply a small function to parse the output. Fortunately, this can usually be done using regular expressions (e.g., with Python's `regex` module).

It would also be a good time to aggregate data appropriately (unless this has been taken care of before, also see Section 4.6).

6. Visualizing Results

After the experiments have finished, the recorded data and results need to be explored and analyzed before they can finally be reported. Data visualization in the form of various different plots is a helpful tool both in the exploration phase and also in the final communication of the experimental results and the formal statistical analysis. The amount of data collected during the experiments is typically too large to report in its entirety and hence meaningful and faithful data aggregations are needed. For future reference and repeatability, it may make sense to include relevant raw data in tables in the appendix. However, raw data tables are rarely good for illustrating trends in the data. While descriptive summary statistics such as means, medians, quartiles, variances, standard deviations, or correlation coefficients, as well as results from statistical testing like p-values or confidence intervals provide a well-founded summary of the data, they do so, by design, only at a very coarse level. The same statistics can even originate from very different underlying data: a famous example is Anscombe's quartet [77], a collection of four sets of eleven points in $\mathbb{R}^2$ with very different point distributions, yet (almost) the same summary statistics such as mean, variance, or correlation (for example plots of the four point sets see https://commons.wikimedia.org/w/index.php?curid=9838454). It is a striking example of how important it can be not to rely on summary statistics alone, but to visualize experimental data graphically. A more recent example is the datasaurus (https://www.autodeskresearch.com/publications/samestats).

Here we discuss a selection of *plot types* for data visualization with focus on algorithm engineering experiments, together with guidelines when to use which type of plot depending on the properties of the data. For a more comprehensive introduction to data visualization, we refer the reader to some of the in-depth literature on the topic [78–80]. Furthermore, there are many powerful libraries and tools for generating data plots, e.g., R (https://www.r-project.org) and the ggplot2 package (https://ggplot2.tidyverse.org), gnuplot (http://www.gnuplot.info), matplotlib (https://matplotlib.org). Additionally, mathematical software such as MATLAB (https://www.mathworks.com/products/matlab.html) (or even spreadsheet tools) can generate various types of plots from your data. For more details about creating plots in one of these tools, we refer to the respective user manuals and various available tutorials.

When presenting data in two-dimensional plots, *marks* are the basic graphical elements or geometric primitives to show data. They can be points (zero-dimensional), lines (1D), or areas (2D). Each mark can be further refined and enriched by visual variables or *channels* such as their position, size, shape, color, orientation, texture, etc. to encode different aspects of the data. (The term *attribute* is also commonly used to describe the same concept. We use "channel" for compatibility with relevant literature, i.e., [81,82]). The most important difference between those channels is that some are more suitable to represent categorical data (e.g., graph properties, algorithms, data sources) by assigning different shapes or colors to them, whereas others are well suited for showing quantitative and ordered data (e.g., input sizes, timings, quantitative quality measures) by mapping them to a spatial position, size, or lightness. For instance, in Figure 5 we use blue circles as marks for one of the algorithms, while for the other we use orange crosses. Using different shapes makes sure that the plots are still readable if printed in grey-scale. Not all channels are equally expressive and effective in encoding information for the human visual system, so that one has to carefully select which aspects of the data to map to which channels, possibly also using redundancy. For more details see the textbooks [81,82].

As discussed in Section 4.6, the types of metrics from algorithmic experiments comprise two main aspects: running time data and solution quality data. Both types can consist of absolute or relative values. Typically, further attributes and parameters of the experimental data, of the algorithms, and of the input instances are relevant to include in a visualization to communicate the experimental findings. These can range from hardware-specific parameters, over the set of algorithms and possible algorithm-specific parameters, to instance-dependent parameters such as certain graph properties. Depending on the focus of the experiment, one needs to decide on the parameters to show in a plot and on how to map them to marks and channels. Typically, the most important metric to answer the guiding research question (e.g., running time or solution quality) is plotted along the y-axis. The x-axis, in turn, is used for the most relevant parameter describing the instances (e.g., instance size for a scalability evaluation or a graph parameter for evaluating its influence on the solution quality). Additional parameters of interest can then be represented by using distinctly colored or shaped marks for different experimental conditions such as the respective algorithm, an algorithmic parameter or properties of the used hardware; see for example that, in Figure 5 instances roadNet-PA (rdPA), roadNet-TX (rdTX) are plotted differently for KADABRA because RK did not finish within the allocated time frame of 7 h.

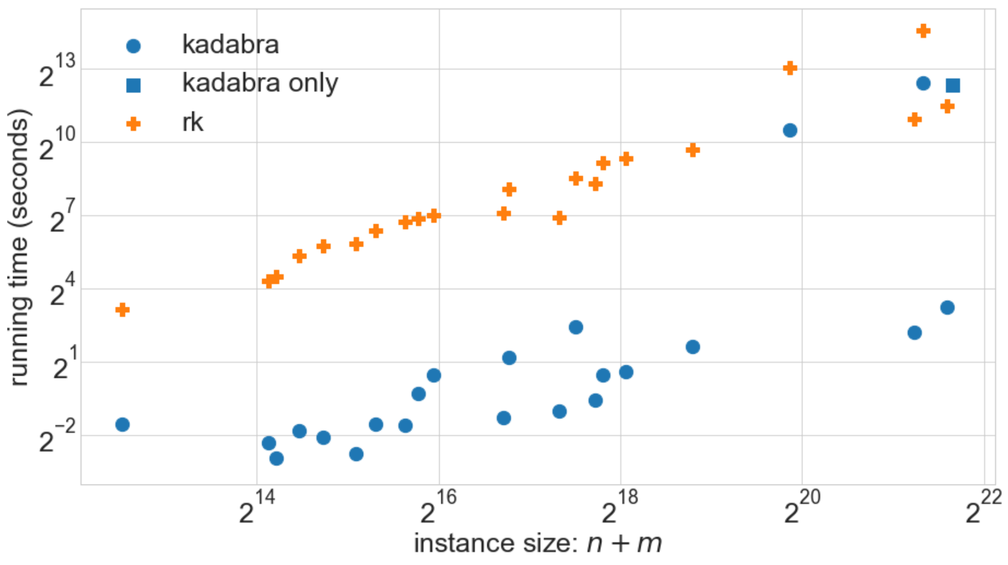


Figure 5. Scatter plot for running times of KADABRA for 4375.0 and RK for all instances in the evaluation set. Both axes are in log scale. Every point is the (arithmetic) mean running time for 5 repeated runs for each instance. The squares represent the run of KADABRA for the networks roadNet-PA (rdPA), roadNet-TX (rdTX) for which RK did not finish within the cutoff time of 7 h.

Before starting to plot the data, one needs to decide whether raw absolute data should be visualized (e.g., running times or objective function values) or whether the data should be shown relative to some baseline value, or be normalized prior to plotting. This decision typically depends on the specific algorithmic problem, the experimental setup and the underlying research questions to be answered. For instance, when the experiment is about a new algorithm for a particular problem, the algorithmic speedup or possible improvement in solution quality with respect to an earlier algorithm may be of interest; hence, running time ratios or quality ratios can be computed as a variable to plot—as shown in Figure 6 with respect to KADABRA and RK. Another possibility of data preprocessing is to normalize certain aspects of the experimental data before creating a plot. For example, to understand effects caused by properties of the hardware, such as cache sizes and other memory effects, one may normalize running time measurements by the algorithm's time complexity in terms of n and m, the number of vertices and edges of the input graph, and examine if the resulting computation times are constant or not. A wide range of meaningful data processing and analysis can be done before showing the resulting data in a visualization. While we just gave a few examples, the actual decision of what to show in a plot needs to be made by the designers of the experiment after carefully exploring all relevant aspects of the data from various perspectives. For the remainder of this section, we assume that all data values to be visualized have been selected.

A very fundamental plot is the *scatter plot*, which maps two variables of the data (e.g., size and running time) onto the x- and y-axis, see Figure 5. Every instance of the experiment produces its own point mark in the plot, by using its values of the two chosen variables as coordinates. Further variables of the data can be mapped to the remaining channels such as color, symbol shape, symbol size, or texture. If the number of instances is not too large, a scatter plot can give an accurate visualization of the characteristics and trends of the data. However, for large numbers of instances, overplotting can quickly lead to scatter plots that become hard to read.

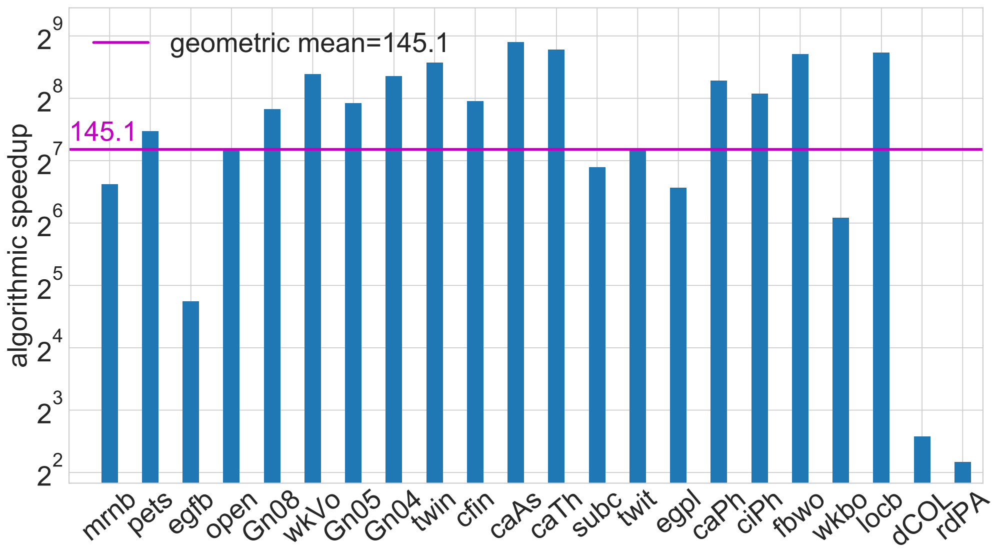


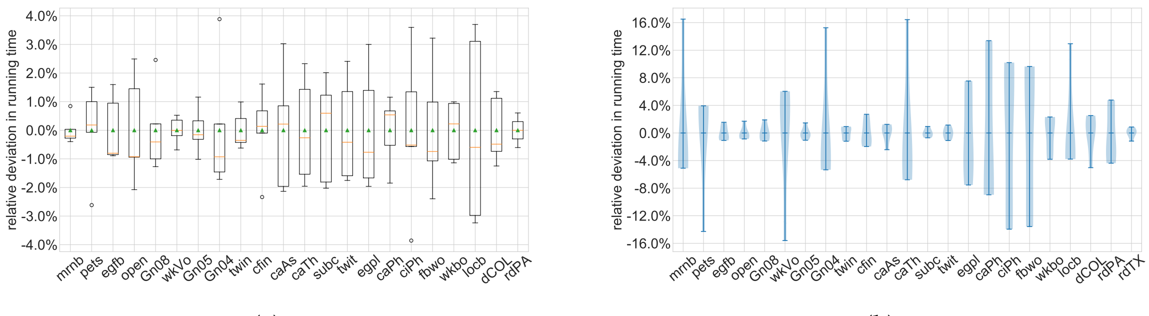
Figure 6. The running time speedup of KADABRA over RK for all the instances in the evaluation set for 4375.0. Every bar is the ratio of the arithmetic means for 5 repeated runs per instance. The smallest speedup is 6.0 for instance dimacs9-COL (dCOL) and the largest speedup is 478.7 for instance ca-AstroPh (caAs). The geometric mean for all speedups is around 159.3. Instances are sorted by their number of nodes and the y-axis is in log scale.

In such cases, the data needs to be aggregated before plotting, by grouping similar instances and showing summaries instead. The simplest way to show aggregated data, such as repeated runs of the same instances, is to plot a single point mark for each group, e.g., using the mean or median, and then optionally putting a vertical error bar on top of it showing one standard deviation or a particular confidence interval of the variable within each group. Such a plot can be well suited for showing how running times scale with the instance size. If the sample sizes in the experiment have been chosen to cover the domain well, one may amplify the salience of the trend in the data by linking the point marks by a line plot. However, this visually implies a linear interpolation between neighboring measurements and therefore should only be done if sufficiently many sample points are included and the plot is not misleading. Obviously, if categorical data are represented on the x-axis, one should never connect the point marks by a line plot.

At the same time, the scale of the two coordinate axes is also of interest. While a linear scale is the most natural and least confusing for human interpretation, some data sets contain values that may grow exponentially. On a linear scale, this results in the large values dominating the entire plot and the small values disappear in a very narrow band. In such cases, axes with a logarithmic scale can be used; however, this should always be made explicit in the axis labeling and the caption of the plot.

A more advanced summary plot is the *box plot* (or *box-and-whiskers plot*), where all repeated runs of the same instance or sets of instances with equal or similar size form a group, see Figure 7a. This group is represented as a single glyph showing simultaneously the median, the quartiles, the minimum and maximum values or another percentile, as well as possibly outliers as individual marks. If the x-axis shows an ordered variable such as instance size, one can still clearly see the scalability trend in the data as well as the variability within each homogeneous group of instances.

Violin plots take the idea of box plots even further and draw the contour of the density distribution of the variable mapped to the y-axis within each group, see Figure 7b. It is thus visually more informative than a simple box plot, but also more complex and thus possibly more difficult to read. When deciding for a particular type of plot, one has to explore the properties of the data and choose a plot type that is neither oversimplifying them nor more complex than needed.



(**a**) (**b**)

Figure 7. Plots on variance of running times. For each instance in the evaluation set, we perform 5 repeated runs and calculate the mean. Shown in the plot are, for each run, the distance to the mean, divided by the mean. This way we get the relative running time variance for every instance. Here, we use a box and a violin plot for illustration purposes; in a scientific paper one of the two should be chosen. Notice also that instances roadNet-PA (rdPA), roadNet-TX (rdTX) are missing for RK since it did not finish within the 7 h limit. (**a**) A box plot for running time deviation of RK. The orange line indicates the median. (**b**) A violin plot for running time deviation of KADABRA.

Data from algorithmic experiments may also often be aggregated by some attribute into groups of different sizes. In order to show the distribution of the instances into these groups, bar charts (for categorical data) or histograms (for continuous data) can be used to visualize the cardinality of each group. Such diagrams are often available in public repositories like KONECT (for example, one can find this information for moreno_blogs in http://konect.uni-koblenz.de/networks/moreno_blogs). For a complex network, for example, one may want to plot the degree distribution of its nodes with the degree (or bins with a specific degree range) on the x-axis and the number of vertices with a particular degree (or in a particular degree range) on the y-axis. Such a histogram can then quickly reveal how skewed the degree distribution is. Similarly, histograms can be useful to show solution quality ratios obtained by one or more algorithms by defining bins based on selected ranges of quality ratios. Such a plot quickly indicates to the reader what percentage of instances could be solved within a required quality range, e.g., at least 95% of the optimum.

A single plot can contain multiple experimental conditions simultaneously. For instance, when showing running time behavior and scalability of a new algorithm compared to previous approaches, a single plot with multiple trend lines in different colors or textures or with visually distinct mark shapes can be very useful to make comparisons among the competing algorithms. Clearly, a legend needs to specify the mapping between the data and the marks and channels in the plot. Here it is strongly advisable to use the same mapping if multiple plots are used that all belong together. But, as a final remark, bear in mind the size and resolution of the created figures. Avoid clutter and ensure that your conclusion remains clearly visible!

Guideline 7. Effectively visualizing the experimental findings is a problem-dependent task, but it is a necessary step towards communicating them to your audience.

7. Evaluating Results with Statistical Analysis

Even if a result looks obvious (it fulfills the "inter-ocular trauma test", as the saying goes, the evidence hitting you between the eyes), it can benefit from a statistical analysis to quantify it, especially if random components or heuristics are involved. The most common questions for a statistical analysis are:

- Do the experimental results support a given hypothesis, or is the measured difference possibly just random noise? This question is addressed by *hypothesis testing*.
- How large is a measured effect, i.e., which underlying real differences are plausible given the experimental measurements? This calls for *parameter estimation*.
- If we want to answer the first two questions satisfactorily, how many data points do we need? This is commonly called *power analysis*. As this issue affects the planning of experiments, we discussed it already in Section 4.3.

The two types of hypotheses we discussed in Section 4.1 roughly relate to hypothesis tests and parameter estimation. Papers proposing new algorithms mostly have hypotheses of the first type: The newly proposed algorithm is faster, yields solutions with better quality or is otherwise advantageous.

In many empirical sciences, the predominant notion has been *null hypothesis significance testing* (NHST), in which a statistic of the measured data is compared with the distribution of this statistic under the *null hypothesis*, an assumed scenario where all measured differences are due to chance. Previous works on statistical analysis of experimental algorithms, including the excellent overviews of McGeoch [38] and Coffin and Saltzmann [15], use this paradigm.

Due to some limitations of the NHST model, a shift towards *parameter estimation* [83–86] and also Bayesian methods [87–89] is taking place in the statistical community.

We aim at applying the current state of the art in statistical analysis to algorithm engineering, but since no firm consensus has been reached [85], we discuss both frequentist and Bayesian approaches. As an example for null hypothesis testing, we investigate whether the KADABRA and the RK algorithms give equivalent results for the same input graphs. While this *equivalence test* could also be performed using Bayesian inference, we use a null hypothesis test for illustration purposes.

It is easy to see from running time plots (e.g., Figure 5) that KADABRA is faster than RK. To quantify this speedup, we use Bayesian methods to infer plausible values for the algorithmic speedup and different scaling behavior. We further evaluate the influence of the graph diameter on the running time.

7.1. Statistical Model

A statistical model defines a family of probability distributions over experimental measurements. Many experimental running times have some degree of randomness: caching effects, network traffic and influence of other processes contribute to this, sometimes the tested algorithm is randomized itself. Even a deterministic implementation on deterministic hardware can only be tested on a finite set of input data, representing a random draw from the infinite set of possible inputs. A model encodes our *assumptions* about how these sources of randomness combine to yield the distribution of outputs we see. If, for example, the measurement error consists of many additive and independent parts, the central limit theorem justifies a normal distribution.

To enable useful inferences, the model should have at least one free *parameter* corresponding to a quantity of interest. Any model is necessarily an abstraction, as expressed in the aphorism "all models are wrong, but some are useful" [90].

7.1.1. Example

Suppose we want to investigate whether KADABRA scales better than RK on inputs of increasing size. (To reproduce this example and the following inferences, one may use the companion Jupyter notebook *Analysis-Example*, which is included in the statistics subdirectory of the public repository of this paper: https://github.com/hu-macsy/ae-tutorial-paper). Figure 5 shows running times of KADABRA and RK with respect to the instance size. These are spread out, implying either that the running time is highly variable, or that it depends on aspects other than the instance size.

In general, running times are modeled as functions of the input size, sometimes with additional properties of the input (or of the output, e.g., in the case of parametrized or output-sensitive algorithms). The running time of KADABRA, for example, possibly depends on the diameter of the input graph. Algorithms in network analysis often have polynomial running times where a reasonable upper bound for the leading exponent can be found. Thus, the running time can be modeled as such a polynomial, written as $a + bn + cn^2 + dn^3 + \ldots + \alpha \log n + \beta \log^2 n + \ldots$, with the unknown coefficients of the polynomial being the free parameters of the model.

However, a large number of free model parameters makes inference difficult. This includes the danger of overfitting, i.e., inferring parameter values that precisely fit the measured results but are unlikely to generalize.

To evaluate the scaling behavior, it is thus often more useful to focus on the largest exponent instead. Let $T_A(n)$ be the running time of the implementation of RK and $T_B(n)$ the running time of the implementation of KADABRA on inputs of size n, with unknown parameters $\alpha_A, \alpha_B, \beta_A$ and β_B. (For estimating asymptotic upper bounds, see the work of McGeoch et al. [33] on curve bounding).

$$T_A(n) = \alpha_A \cdot n^{\beta_A} \cdot \epsilon$$
$$T_B(n) = \alpha_B \cdot n^{\beta_B} \cdot \epsilon$$

The term ϵ explicitly models the error; it can be due to variability in inputs (some instances might be harder to process than others of the same size) and measurement noise (some runs suffer from interference or caching effects). Since harder inputs are not constrained to additive difficulty and longer runs have more opportunity to experience adverse hardware effects, we choose a multiplicative error term. (Summands with smaller exponents are also subsumed within the error term. If they have a large effect, an additive error might reflect this more accurately). Taking the logarithms of both sides makes the equations linear:

$$\log(T_A(n)) = \log(\alpha_A) + \beta_A \log(n) + \log(\epsilon) \tag{4}$$
$$\log(T_B(n)) = \log(\alpha_B) + \beta_B \log(n) + \log(\epsilon) \tag{5}$$

A commonly chosen distribution for additive errors is Gaussian, justified by the central limit theorem [91]. Since longer runs have more exposure to possibly adverse hardware or network effects we consider multiplicative error terms to be more likely and use a log-normal distribution. (In some cases, it might even make sense to use a hierarchical model with two different error terms: One for the input instances, the other one for the differences on the same input). Since the logarithm of the log-normal distribution is a normal (Gaussian) distribution, Equations (4) and (5) can be rewritten as normally distributed random variables:

$$\log(T_A(n)) \sim \mathcal{N}(\log(\alpha_A) + \beta_A \log(n),\ \sigma_\epsilon^2) \tag{6}$$
$$\log(T_B(n)) \sim \mathcal{N}(\log(\alpha_B) + \beta_B \log(n),\ \sigma_\epsilon^2) \tag{7}$$

This form shows more intuitively the idea that a statistical model is *a set of probability measures on experimental outcomes*. In this example, the set of probability measures modeling the performance of an algorithm are parametrized by the tuple $(\alpha, \beta, \sigma) \in \mathbb{R}^3$.

A problem remains if the input instances are very inhomogeneous. In that case, both A and B have a large estimated variance (σ^2). Even if the variability of performance *on the same instance* is small, any genuine performance difference might be wrongly attributed to the large *inter-instance* variance. This issue can be addressed with a combined model as recommended by Coffin and Saltzmann [15], in which the running time $T_A(x)$ of A is a function of the running time $T_B(x)$ of B *on the same instance* x:

$$\log(T_A(x)) \sim \mathcal{N}(\log(\alpha_{A/B}) + \beta_{A/B} \log(T_B(x)), \sigma^2) \tag{8}$$

For a more extensive overview of modeling experimental algorithms, see [38].

7.1.2. Model Fit

Several methods exist to infer parameters when fitting a statistical model to experimental data. The most well-known is arguably the maximum-likelihood fit, choosing parameters for a distribution that give the highest probability for the observed measurements. In the case of a linear model with a normally distributed error, this is equivalent to a least-squares fit [92]. Such a fit yields a single estimate for plausible parameter values.

Given random variations in the measurement, we are also interested in the *range* of plausible values, quantifying the uncertainty involved in measurement and instance selection. This is covered in Sections 7.3.2 and 7.4.

7.2. Formalizing Hypotheses

Previously (Section 4.1), we discussed two types of hypotheses. The first type is that a new algorithm is better in some aspect than the state of the art. The second type claims insight into how the behavior of an algorithm depends on settings and properties of the input. We now express these same hypotheses more formally, as statements about the parameters in a statistical model. The two types are then related to the statistical approaches of *hypothesis testing* and *parameter estimation*.

Considering the scaling model presented in Equation (8) and the question whether implementation A scales better than implementation B, the parameter in question is the exponent $\beta_{A/B}$. The hypothesis that A scales better than B is equivalent to $\log(\beta_{A/B}) < 0$; both scaling the same implies $\log(\beta_{A/B}) = 0$. Note that the first hypothesis does not imply a fully specified probability distribution on $\beta_{A/B}$, merely restricting it to the negative half-plane. The hypothesis of $\log(\beta_{A/B}) = 0$ does completely specify such a distribution (i.e., a point mass of probability 1 at 0), which is useful for later statistical inference.

7.3. Frequentist Methods

Frequentist statistics defines the probability of an experimental outcome as the *limit of its relative frequency* when the number of experiments trends towards infinity. This is usually denoted as the classical approach.

7.3.1. Null Hypothesis Significance Testing

As discussed above, *null hypothesis significance testing* evaluates a proposed hypothesis by contrasting it with a *null hypothesis*, which states that no true difference exists and the observed difference is due to chance. As an example application, we compare the approximation quality of KADABRA and RK. From theory, we would expect higher approximation errors from KADABRA, since it samples fewer paths. We investigate whether the measured empirical difference supports this theory, or could also be explained with random measurement noise, i.e., with the null hypothesis (denoted with H_0) of both algorithms having the same distribution of errors and just measuring higher errors from KADABRA by coincidence. Here it is an advantage that the proposed alternate hypothesis (i.e., the distributions are meaningfully different) does not need an explicit modeling of the output distribution, as the distribution of differences in our case does not follow an easily parameterizable distribution.

When deciding whether to reject a null hypothesis (and by implication, support the alternate hypothesis), it is possible to make one of two errors: (i) rejecting a null hypothesis, even though it is true (false positive), (ii) failing to reject a false null hypothesis (false negative). In such probabilistic decisions, the error rate deemed acceptable often depends on the associated costs for each type of error. For scientific research, Fisher suggested that a false positive rate of 5% is acceptable [93], and most fields follow that suggestion. This threshold is commonly denoted as α.

Controlling for the first kind of error, the p-value is defined as the probability that a *summary statistic* as extreme as the observed one would have occurred given the null hypothesis [94]. Please note that this is *not* the probability $P\left(H_0 | \text{observations}\right)$, i.e., the probability that the null hypothesis is true given the observations.

For practical purposes, a wide range of *statistical hypothesis tests* have been developed, which aggregate measurements to a summary statistic and often require certain conditions. For an overview of which of them are applicable in which situation, see the excellent textbook of Young and Smith [94].

KADABRA Example

In our example the paired results are of very different instances and clearly not normally distributed. We thus avoid the common t-test and use a Wilcoxon test of pairs [95] from the SciPy [96] stats module, yielding a p-value of $4 \cdot 10^{-5}$, see cell 14 of the Analysis-Example notebook in the statistics subdirectory. Since this is smaller than our threshold α of 0.05, one would thus say that this result allows us to reject the null hypothesis at the level of $4 \cdot 10^{-5}$. Such a difference is commonly called *statistically significant*. To decide whether it is actually significant in practice, we look at the magnitude of the difference: The error of KADABRA is about one order of magnitude higher for most instances, which we would indeed call significant.

Multiple Comparisons

The NHST approach guarantees that of all false hypotheses, the expected fraction that seem significant when tested is at most α. Often though, a publication tests more than one hypotheses. For methods to address this problem and adjust the probability that *any* null hypothesis in an experiment is falsely rejected (also called the *familywise error rate*), see Bonferroni [97] and Holm [98].

7.3.2. Confidence Intervals

One of the main criticism of NHST is that it ignores effect sizes; the magnitude of the p-value says nothing about the magnitude of the effect. More formally, *for every true effect with size $\epsilon > 0$ and every significance level α, there exists an n_0 so that all experiments containing $n \geq n_0$ measurements are likely to reject the null hypothesis at level α.* Following this, Coffin and Saltzmann [15] caution against overly large data sets—a recommendation which comes with its own set of problems.

The statistical response to the problem of small but meaningless p-values with large data sets is a shift away from hypothesis testing to *parameter estimation*. Instead of asking whether the difference between populations is over a threshold, the difference is quantified as a parameter in a statistical model, see also Section 7.2. As Kruschke et al. [88] put it, the null hypothesis test asks whether the null value of a parameter would be rejected at a given significance level. A *confidence interval* merely asks which other parameter values would not be rejected [88]. We refer to [99,100] for a formal definition and usage guidelines.

7.4. Bayesian Inference

Bayesian statistics defines the probability of an experimental outcome as the *uncertainty of knowledge about it*. Bayes's theorem gives a formal way to update probabilities on new observations. In its simplest form for discrete events and hypotheses, it can be given as:

$$P\left(H_1|\mathrm{A}\right) = \frac{P\left(\mathrm{A}|H_1\right) P\left(H_1\right)}{P\left(\mathrm{A}\right)} \tag{9}$$

When observing outcome A, the probability of hypothesis H_1 is proportional to the probability of A conditioned on H_1 multiplied by the *prior probability* $P\left(H_1\right)$. The conditional probability $P\left(A|H_1\right)$ of an outcome A given an hypothesis H_1 is also called the *likelihood* of H_1. The prior probability reflects the estimation before making observations, based on background knowledge.

Extended to continuous distributions, Bayes's rule allows to combine a statistical model with a set of measurements and a *prior probability distribution* over parameters to yield a *posterior probability distribution* over parameters. This posterior distribution reflects both the uncertainty in measurements and possible prior knowledge about parameters. A thorough treatment is given by Gelman et al. [101].

For our example model introduced in Section 7.1, we model the running times of implementation B as a function of the time of implementation A, as done in Equation (8):

$$\log(T_A(n)) \sim \mathcal{N}(\log(\alpha_{A/B}) + \beta_{A/B} \cdot \log(T_B(n)), \sigma^2_{A/B})$$

This defines the likelihood function as a Gaussian noise with variance σ^2. Since this variance is unknown, we keep it as a model parameter. As we have no specific prior information about plausible values of α and β, we define the following vague prior distributions:

$$\alpha_{A/B} \sim \mathrm{Lognormal}(0, 10),\ \beta_{A/B} \sim \mathcal{N}(1, 10),\ \sigma_{A/B} \sim \mathrm{InvGamma}(1, 1)$$

The first two distributions represent our initial—conservative—belief that the two implementations are equivalent in terms of scaling behavior and constants. We model the variance of the observation noise σ^2 as an inverse gamma distribution instead of a normal distribution, since a variance cannot be negative.

Figure 8 shows how to compute and trace the posterior distribution of these three parameters using SciPy [96] and PyMC3 [102]. Results are listed in Table 4. The interval of *Highest Probability Density* (HPD) is constructed to contain 95% of the probability mass of the respective posterior distribution. It is the Bayesian equivalent of the confidence interval (Section 7.3.2) and also called *credible interval*. The most probable values for $\alpha_{A/B}$ and $\beta_{A/B}$ are $e^{-5.27} \approx 0.0051$ and 1.04, respectively. Taking measurement uncertainty into account, the true values are within the intervals $[0.0008, 0.03]$ respective $[0.72, 1.36]$ with 95% probability. This shows that KADABRA is more than two orders of magnitude faster on average, but results about the relative scaling behavior are inconclusive. While the average for $\beta_{A/B}$ is 1.04 and suggests similar scaling, the interval $[0.72, 1.36]$ neither excludes the hypothesis that KADABRA scales better, nor the hypothesis that it scales worse.

```
import pymc3 as pm
from scipy import optimize
basic_model = pm.Model()
with basic_model:
logalpha = pm.Normal('alpha', mu=0, sd=10)
beta = pm.Normal('beta', mu=0, sd=10)
sigma = pm.InverseGamma('sigma', alpha=1,beta=1)
mu = logalpha + beta*logTimeRK
Y_obs = pm.Normal('Y_obs', mu=mu, sd=sigma, observed=logTimeKadabra)
with basic_model:
# approximate posterior distribution with 10000 samples
trace = pm.sample(10000)
pm.summary(trace)
```

Figure 8. Example listing for inference of parameters $\alpha_{A/B}$, $\beta_{A/B}$ and $\sigma_{A/B}$.

Table 4. Posterior distribution of parameter values.

	HPD 2.5	**Mean**	**HPD 97.5**
$\log \alpha_{A/B}$	−7.05	−5.27	−3.48
$\beta_{A/B}$	0.72	1.04	1.36
$\sigma_{A/B}$	0.85	1.19	1.59

7.4.1. Equivalence Testing

Computing the highest density interval can also be used for hypothesis testing. In Section 7.3.1 we discussed how to show that two distributions are different. Sometimes, though, we are interested in showing that they are sufficiently *similar*. An example would be wanting to show that two sampling algorithms give the same distribution of results. This is not easily possible within the NHST paradigm, as the two answers of a classical hypothesis test are "probably different" and "not sure".

This problem can be solved by calculating the posterior distribution of the parameter of interest and defining a *region of practical equivalence* (ROPE), which covers all parameter values that are effectively indistinguishable from 0. If x% of the posterior probability mass are in the region of practical equivalence, the inferred parameter is practically indistinguishable with probability x%. If x% of the probability mass are outside the ROPE, the parameter is meaningfully different with probability x%. If the intervals overlap, the observed data is insufficient to come to either conclusion. In our example, the scaling behavior of two algorithms is equivalent if the inferred exponent modeling their relative running times is more or less 1. We could define practical equivalence as $\pm 5\%$, resulting in a region of practical equivalence of $[0.95, 1.05]$. The interval $[0.72, 1.36]$ containing 95% of the probability mass for β is neither completely inside $[0.95, 1.05]$ nor completely outside it, implying that more experiments are needed to come to a conclusion. In Section 4.3, we discussed that the width of many confidence and credible intervals is inversely proportional to the square root of the number of samples used to compute it. If the scaling is indeed equivalent, we would thus expect to need at least $((1.36 - 0.72)/(1.05 - 0.95))^2 \approx 41$ times as many samples to confirm it.

7.4.2. Bayes Factor

Bayes factors are a way to compare the *relative fit* of several models and hypotheses to a given set of observations. To compute this relative fit of two hypotheses H_1, H_2 to an observation A, we apply the Bayes theorem (Equation (9)) to both and consider their ratios:

$$\frac{P(H_1|\mathrm{A})}{P(H_0|\mathrm{A})} = \frac{P(\mathrm{A}|H_1)}{P(\mathrm{A}|H_0)} \frac{P(H_1)}{P(H_0)}$$

Crucially, the ratio of prior probabilities, which is subjective, is a separate factor from the ratio of likelihoods, which is objective. This objective part, the ratio $P(\mathrm{A}|H_1)/P(\mathrm{A}|H_0)$, is called the *Bayes factor*. It models how much the *posterior odds ratio* $P(H_1|\mathrm{A})/P(H_0|\mathrm{A})$ differs from the *prior odds ratio* $P(H_1)/P(H_0)$ due to making observation A.

The first obvious difference to NHST is that calculating a Bayes Factor consists of comparing the fit of *both* hypotheses to the data, not only the null hypothesis. It thus requires that an alternate hypothesis is stated explicitly, including a probability distribution over observable outcomes. If the alternative hypothesis is meant to be vague, for example just that two distributions are different, an uninformative prior with a high variance should be used. However, specific hypotheses like "the new algorithm is at least 20% faster" can also be modeled explicitly.

This explicit modeling allows inference in both directions; using NHST, on the other hand, one can only ever infer that a null hypothesis is unlikely or that the data is insufficient to infer this. Using Bayes factors, it is possible to infer that H_1 is more probable than H_0, or that the observations are insufficient to support this statement, or that H_0 is more probable than H_1.

In the previous running time analysis, we hypothesized a better scaling behavior, which was not confirmed by the experimental measurements. However, a cursory complexity analysis of the KADABRA algorithm suggests that the diameter has an influence on the running time. Might also the relative scaling of KADABRA and RK depend on the diameter?

To answer this question, we compare the fit of two models: The first model is the same as discussed earlier (Equation (8)), it models the expected running time of KADABRA on instance x as $e^{\alpha} \cdot T_{\mathrm{RK}}(x)^{\beta}$, where $T_{\mathrm{RK}}(x)$ is the running time of RK on the same instance. The second model has the additional free parameter γ, controlling the interaction between the diameter and running times:

$$\log(T(x)) \sim \mathcal{N}(\log(\alpha) + \beta \log(T_{\mathrm{RK}}(x)) + \gamma \log(\mathrm{diam}_x),\ \sigma_{\epsilon}). \tag{10}$$

Comparing for example the errors of a least-squares fit of the two models would not give much insight, since including an additional free parameter in a model almost always results in a better fit. This does not have to mean that the new parameter captures something interesting.

Instead, we calculate the Bayes factor, for which we integrate over the prior distribution in each model (if favoring frequentist methods, one could compare whether the correlation of running times and diameter is significantly higher than expected by chance). Since this integral over the prior distribution also includes values of the new parameter which are not a good fit, models with too many additional parameters are automatically penalized. Our two models are similar, thus we can phrase them as a hierarchical model with the additional parameter controlled by a boolean random variable, see Figure 9.

```
import pymc3 as pm
from scipy import optimize
basic_model = pm.Model()
with basic_model:
pi = (0.5, 0.5)
selected_model = pm.Bernoulli('selected_model', p=pi[1])
alpha = pm.Normal('alpha', mu=0, sd=1)
beta = pm.Normal('beta', mu=0, sd=1)
gamma = pm.Normal('gamma', mu=0, sd=1)
sigma = pm.InverseGamma('sigma', alpha=1, beta=1)
mu = alpha + beta*logTimesRK + gamma*logDiameters*selected_model
Y_obs = pm.Normal('Y_obs', mu=mu, sd=sigma, observed=logTimeKadabra)
```

Figure 9. Listing to compute the Bayes factor, the relative likelihood of two models, between the model including the graph diameter and the model with just the graph sizes. The variable *selected_model* is an indicator for which model is selected.

The posterior for the indicator variable `selected_model` is 0.67, yielding a Bayes factor of $\approx$2 in support of including the diameter in the model. This is so inconclusive as to be barely worth mentioning [103]. A likely reason for this uncertain result is that our high-diameter graphs are also significantly larger and thus have long running times even independent from their high diameter.

Graph Generator Example

Some of the publications that could have been improved by applying these methods are our own. In von Looz et al. [104], we present an algorithm to efficiently sample random graphs from hyperbolic geometry. To demonstrate its correctness, we compared the distribution of generated graphs with the results of a reference implementation. At the time, we decided to plot the average properties of both generator outputs, which indeed look similar except for random fluctuations.

How can we approach this more rigorously? Showing that the two distributions are *exactly* equal is of course impossible, as it requires an infinite number of samples. We set a (somewhat arbitrary) threshold of 0.1σ instead: When the difference *between* the average output of the generators is at least one order of magnitude less than the average difference of two instances sampled from *the same* generator, we call them indistinguishable in practice. Due to the central limit theorem, we expect the measurements to be normally distributed. For two normally distributed random variables $A \sim$

$\mathcal{N}(\mu_A, \sigma_A)$ and $B \sim \mathcal{N}(\mu_B, \sigma_B)$, their difference is again normally distributed, with $A - B \sim \mathcal{N}(\mu_A - \mu_B, \sqrt{\sigma_A^2 + \sigma_B^2})$. Estimating this difference $\mu_A - \mu_B$ by taking n samples has a standard error of $\sqrt{\frac{\sigma_A^2+\sigma_B^2}{n}}$, see Section 4.3. Plausibly judging whether an estimated difference is below $0.1\sigma_A$ requires that the expected error of the estimate itself is at most $0.1\sigma_A$. As σ_A and σ_B are equivalent, this happens if $n \geq 200$.

A problem in this scenario is that the real quantities of interest are the probabilities over isomorphic graph classes, which do not lend themselves to understandable modeling. We instead consider the distributions of five network properties: the average degree, the diameter, the clustering coefficient, the degeneracy and the degree assortativity.

Observable differences in these properties will reflect underlying differences in the graphs' structure. To ensure our results are generally applicable, we explore different parameters for the graph density and degree distributions, 28 combinations in total. We thus generate at least 200 random graphs for each combination of parameters.

A problem remains that when testing 28 combinations of input parameters and 5 properties each, some of the 140 tests are likely to show a difference by chance—the same problem with multiple comparisons in the NHST (Section 7.3.1) approach (with random draws and such a high number of tests, one could easily show by accident that a distribution is meaningfully different *from itself*). We instead define two hypotheses H_0 and H_1 and compute a Bayes factor to decide between them. As the properties have different variances, we estimate the standard deviation σ separately for each property. Let H_0 then be the hypothesis that for all properties, the true difference between the average outputs of the generators is at most 10% of the estimated standard deviation σ. Let H_1 be the hypothesis that this difference is above 10%.

The notebook *Equivalence-BF-Hyperbolic-Generator* in the statistics subdirectory contains code to compute the Bayes Factor comparing H_0 and H_1. This analysis shows that hypothesis H_0 is overwhelmingly more likely, meaning that if there is a difference between the properties of the generated graphs, it is at least one order of magnitude smaller than the average difference between graphs from one generator. It also shows that for some parameters, both our and the reference implementation deviate from the requested degree distribution in the same way, indicating a possible weakness in the shared approximations used to derive internal parameters.

7.5. Recommendations

Different statistical methods fulfill different needs. For almost all objectives, both Bayesian and frequentist methods exist, see Table 5. In experimental algorithmics, most hypotheses can be expressed as statements constraining parameters in a statistical model, i.e., "the average speedup of A over B is at least 20%". Thus, in contrast to earlier statistical practice, we recommend to approach evaluation of hypotheses by parameter estimation and to only use the classical hypothesis tests when parameter estimation is not possible. The additional information gained by parameter estimates has a couple of advantages. For example, when using only hypothesis tests, small differences in large data sets can yield impressively small p-values and thus claims of *statistical significance* even when the difference is irrelevant in practice and the significance is statistical only [15]. Using confidence intervals (Section 7.3.2) or the posterior distribution in addition with a region of practical equivalence avoids this problem [88].

Table 5. Overview of different statistical methods. TOST stands for *Two One-Sided T-Tests* and AIC is the *Akaike Information Criterion* [105], a combined measure for model fit and model complexity.

		Frequentist	**Bayesian**
Estimation		**Confidence Interval**	**Posterior**
Hypothesis	Equivalence	TOST	Posterior + ROPE *or* BF
	Difference	NHST	
	Model Selection	AIC	Bayes Factor (BF)

Below is a rough guideline (also shown in Figure 10) outlining our method selection process. It favors Bayesian methods, since the python library PyMC3 [102] offering them fits well into our workflow.

1. Define a model that captures the parts of the measured results that interest you, see Section 7.1.
2. Using *confidence intervals* (Section 7.3.2) or *credible intervals* (Section 7.4), estimate plausible values for the model parameters, including their uncertainty. If this proves intractable and you are only interested in whether a measured difference is due to chance, use a significance test instead (Section 7.3.1).
3. If you want to show that two distributions (of outcomes of algorithms) are *similar*, use an equivalence test, in which you define a region of practical equivalence.
4. If you want to show that two distributions (of outcomes of algorithms) are *different*, you may also use an equivalence test or alternatively, a significance test.
5. If you want to compare how well different hypotheses explain the data, for example compare whether the diameter has an influence on relative scaling, compare the *relative fit* using a Bayes factor (Section 7.4.2). Bayes factors are also useful when investigating equivalence or difference of more complex models.

Needless to say, these are only recommendations.

Guideline 8. Statistical methods help a rigorous interpretation of experimental results. The classical approach of using *p*-values, while common, is often not the most appropriate method.

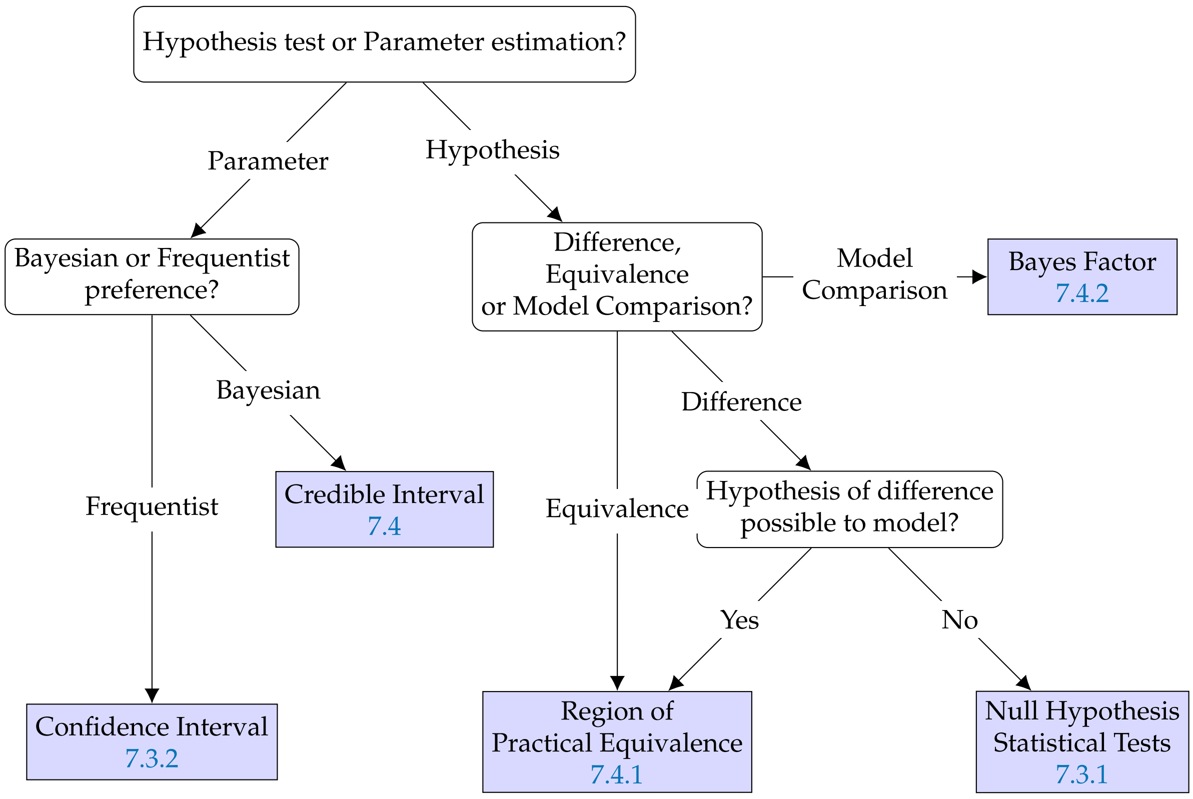

Hypothesis test or Parameter estimation?
Parameter
Hypothesis
Bayesian or Frequentist preference?
Difference, Equivalence or Model Comparison?
Model Comparison
Bayes Factor 7.4.2
Bayesian
Difference
Frequentist
Credible Interval 7.4
Equivalence
Hypothesis of difference possible to model?
Yes
No
Confidence Interval 7.3.2
Region of Practical Equivalence 7.4.1
Null Hypothesis Statistical Tests 7.3.1

Figure 10. Flowchart outlining our method selection.

8. Conclusions

Besides setting guidelines for experimental algorithmics, this paper provides a tool for simplifying the typical experimental workflow. We hope that both are useful for the envisioned target group—and beyond, of course. We may have omitted material some consider important. This happened on purpose to keep the exposition reasonably concise. To account for future demands, we could imagine an evolving "community version" of the paper, updated with the help of new co-authors in regular time intervals. That is why we invite the community to contribute comments, corrections and/or text. (The source files of this paper can be found at https://github.com/hu-macsy/ae-tutorial-paper. We encourage readers post suggestions via GitHub issues and welcome pull requests).

Let us conclude by reminding the reader: most of the guidelines in the paper are not scientific laws nor set in stone. Always apply common sense to adapt a guideline to your concrete situation! In addition, we cannot claim that we always followed the guidelines in the past. But this was all the more motivation for us to write this paper, to develop SimexPal and to have a standard to follow—we hope that the community shares this motivation.

Author Contributions: All credited authors contributed equally to this work.

Funding: The work by the authors from HU Berlin was partially supported by grant ME 3619/3-2 of the German Research Foundation (DFG) Priority Programme 1736 *Algorithms for Big Data* and by German Federal Ministry of Education and Research (BMBF) project WAVE (grant 01 | H15004B). M.N. was partially supported by the Austrian Science Fund (FWF) P 31119.

Acknowledgments: We thank Michael Hamann and Sebastian Schlag for their timely and thorough feedback on a working draft of the paper. We also thank Alexander Meier, Dimitri Ghouse and Patrick Lahr for valuable advice on statistics as well as Sven Mallach for pointers to the literature.

Conflicts of Interest: The authors declare no conflict of interest. The funding agents had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Bogdanov, A.; Trevisan, L. Average-Case Complexity. *Found. Trends® Theor. Comput. Sci.* **2006**, *2*, 1–106. [CrossRef]
2. Spielman, D.; Teng, S.H. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. In Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC'01), Crete, Greece, 6–8 July 2001; ACM: New York, NY, USA, 2001; pp. 296–305. [CrossRef]
3. Roughgarden, T. Beyond worst-case analysis. *Commun. ACM* **2019**, *62*, 88–96. [CrossRef]
4. Heule, M.J.H.; Järvisalo, M.J.; Suda, M. (Eds.) *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*; University of Helsinki: Helsinki, Finland, 2018.
5. Applegate, D.L.; Bixby, R.E.; Chvatal, V.; Cook, W.J. *The Traveling Salesman Problem: A Computational Study*; Princeton Series in Applied Mathematics; Princeton University Press: Princeton, NJ, USA, 2007.
6. Mehlhorn, K.; Sanders, P. *Algorithms and Data Structures: The Basic Toolbox*; Springer: Berlin, Germany, 2008.
7. Puglisi, S.J.; Smyth, W.F.; Turpin, A.H. A Taxonomy of Suffix Array Construction Algorithms. *ACM Comput. Surv.* **2007**, *39*. [CrossRef]
8. Johnson, D.S. A theoretician's guide to the experimental analysis of algorithms. In *DIMACS Workshop Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*; American Mathematical Society: Providence, RI, USA, 1999; pp. 215–250.
9. Muller-Hannemann, M.; Schirra, S. (Eds.) *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*; Springer: Berlin/Heidelberg, Germany, 2010.
10. Moret, B. Towards a discipline of experimental algorithmics. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*; American Mathematical Society: Providence, RI, USA, 2002; pp. 197–213.
11. Sanders, P. Algorithm Engineering—An Attempt at a Definition Using Sorting as an Example. In Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX), Austin, TX, USA, 16 January 2010; pp. 55–61.
12. Bast, H.; Delling, D.; Goldberg, A.; Müller-Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; Werneck, R.F. Route planning in transportation networks. In *Algorithm Engineering*; Springer: Berlin, Germany, 2016; pp. 19–80.
13. Brandes, U.; Robins, G.; McCranie, A.; Wasserman, S. What is network science? *Netw. Sci.* **2013**, *1*, 1–15. [CrossRef]
14. Newman, M. *Networks*; Oxford University Press: Oxford, UK, 2018.
15. Coffin, M.; Saltzman, M.J. Statistical Analysis of Computational Tests of Algorithms and Heuristics. *INFORMS J. Comput.* **2000**, *12*, 24–44. [CrossRef]
16. Borassi, M.; Natale, E. KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation. *LIPIcs-Leibniz Int. Proc. Inf.* **2016**, *57*, 20:1–20:18. [CrossRef]
17. Staudt, C.L.; Sazonovs, A.; Meyerhenke, H. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* **2016**, *4*, 508–530. [CrossRef]
18. Boldi, P.; Vigna, S. Axioms for centrality. *Internet Math.* **2014**, *10*, 222–262. [CrossRef]
19. Freeman, L.C. A set of measures of centrality based on betweenness. *Sociometry* **1977**, *40*, 35–41. [CrossRef]
20. Brandes, U. A faster algorithm for betweenness centrality. *J. Math. Sociol.* **2001**, *25*, 163–177. [CrossRef]
21. Bader, D.A.; Kintali, S.; Madduri, K.; Mihail, M. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*; Springer: Berlin, Germany, 2007; pp. 124–137.

22. Geisberger, R.; Sanders, P.; Schultes, D. Better Approximation of Betweenness Centrality. In Proceedings of the Meeting on Algorithm Engineering & Expermiments, Provincetown, MA, USA, 30 May–1 June 2008; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008; pp. 90–100.
23. Riondato, M.; Kornaropoulos, E.M. Fast approximation of betweenness centrality through sampling. *Data Min. Knowl. Discov.* **2016**, *30*, 438–475. [CrossRef]
24. Riondato, M.; Upfal, E. ABRA: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Trans. Knowl. Discov. Data* **2018**, *12*, 61. [CrossRef]
25. Kunegis, J. Konect: The koblenz network collection. In Proceedings of the 22nd International Conference on World Wide Web, Rio de Janeiro, Brazil, 13–17 May 2013; pp. 1343–1350.
26. Leskovec, J.; Krevl, A. SNAP Datasets: Stanford Large Network Dataset Collection. 2014. Available online: http://snap.stanford.edu/data (accessed on 10 March 2019).
27. Bader, D.; Meyerhenke, H.; Sanders, P.; Wagner, D. (Eds.) Contemporary Mathematics. In Proceedings of the 10th DIMACS Implementation Challenge, Atlanta, GA, USA, 13–14 February 2012; American Mathematical Society: Providence, RI, USA, 2012.
28. Davis, T.A.; Hu, Y. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* **2011**, *38*, 1. [CrossRef]
29. Boldi, P.; Vigna, S. The WebGraph Framework I: Compression Techniques. In Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), New York, NY, USA, 17–22 May 2004; ACM Press: New York, NY, USA, 2004; pp. 595–601.
30. Rossi, R.A.; Ahmed, N.K. An Interactive Data Repository with Visual Analytics. *SIGKDD Explor. Newsl.* **2016**, *17*, 37–41. [CrossRef]
31. Goldenberg, A.; Zheng, A.X.; Fienberg, S.E.; Airoldi, E.M. A survey of statistical network models. *Found. Trends® Mach. Learn.* **2010**, *2*, 129–233. [CrossRef]
32. Lancichinetti, A.; Fortunato, S.; Radicchi, F. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E* **2008**, *78*, 046110. [CrossRef]
33. McGeoch, C.C.; Sanders, P.; Fleischer, R.; Cohen, P.R.; Precup, D. Using Finite Experiments to Study Asymptotic Performance. In *Experimental Algorithmics*; Springer: Berlin, Germany, 2000; pp. 93–126.
34. Bernardo, J.M.; Smith, A.F. *Bayesian Theory*; John Wiley & Sons: Hoboken, NJ, USA, 2009; Volume 405.
35. Altman, D.G.; Bland, J.M. Standard deviations and standard errors. *BMJ* **2005**, *331*, 903. [CrossRef]
36. Ellis, P.D. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*; Cambridge University Press: Cambridge, UK, 2010.
37. Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*; Routledge: Abingdon-on-Thames, UK, 1988.
38. McGeoch, C.C. *A Guide to Experimental Algorithmics*, 1st ed.; Cambridge University Press: New York, NY, USA, 2012.
39. Rardin, R.L.; Uzsoy, R. Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial. *J. Heuristics* **2001**, *7*, 261–304. [CrossRef]
40. Sedgewick, R. Implementing quicksort programs. *Commun. ACM* **1978**, *21*, 847–857. [CrossRef]
41. Eppstein, D.; Wang, J. Fast Approximation of Centrality. In Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01), Washington, DC, USA, 7–9 January 2001; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2001; pp. 228–229.
42. James, G.; Witten, D.; Hastie, T.; Tibshirani, R. *An Introduction to Statistical Learning—With Applications in R*; Springer: Berlin, Germany, 2013.
43. Hinton, G.E. A practical guide to training restricted Boltzmann machines. In *Neural Networks: Tricks of the Trade*; Springer: Berlin, Germany, 2012; pp. 599–619.
44. Larochelle, H.; Erhan, D.; Courville, A.; Bergstra, J.; Bengio, Y. An empirical evaluation of deep architectures on problems with many factors of variation. In Proceedings of the 24th International Conference on Machine Learning, Corvalis, OR, USA, 20–24 June 2007; pp. 473–480.
45. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
46. Bergamini, E.; Meyerhenke, H.; Staudt, C.L. Approximating Betweenness Centrality in Large Evolving Networks. In Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX), San Diego, CA, USA, 5 January 2015; pp. 133–146. [CrossRef]

47. Green, O.; McColl, R.; Bader, D.A. A Fast Algorithm for Streaming Betweenness Centrality. In Proceedings of the 2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust, Amsterdam, The Netherlands, 3–5 September 2012; IEEE Computer Society: Washington, DC, USA, 2012; pp. 11–20. [CrossRef]
48. Kourtellis, N.; Morales, G.D.F.; Bonchi, F. Scalable Online Betweenness Centrality in Evolving Graphs. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 2494–2506. [CrossRef]
49. Brandes, U.; Pich, C. Centrality estimation in large networks. *Int. J. Bifurc. Chaos* **2007**, *17*, 2303–2318. [CrossRef]
50. Bader, D.; Madduri, K. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06), Columbus, OH, USA, 14–18 August 2006; pp. 539–550.
51. Rizi, F.S.; Schloetterer, J.; Granitzer, M. Shortest Path Distance Approximation Using Deep Learning Techniques. In Proceedings of the IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), Barcelona, Spain, 28–31 August 2018; pp. 1007–1014.
52. Akiba, T.; Iwata, Y.; Yoshida, Y. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 349–360.
53. Penschuck, M. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In Proceedings of the 16th International Symposium on Experimental Algorithms (SEA 2017), London, UK, 21–23 June 2017.
54. McLaughlin, A.; Bader, D.A. Scalable and High Performance Betweenness Centrality on the GPU. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 572–583. [CrossRef]
55. Sariyüce, A.E.; Kaya, K.; Saule, E.; Çatalyürek, U.V. Betweenness Centrality on GPUs and Heterogeneous Architectures. In Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, Houston, TX, USA, 16 March 2013; ACM: New York, NY, USA, 2013; pp. 76–85. [CrossRef]
56. Shi, Z.; Zhang, B. Fast network centrality analysis using GPUs. *BMC Bioinform.* **2011**, *12*, 149. [CrossRef]
57. Crescenzi, P.; D'Angelo, G.; Severini, L.; Velaj, Y. Greedily Improving Our Own Centrality in A Network. In *Experimental Algorithms*; Bampis, E., Ed.; Springer: Cham, Switzerland, 2015; pp. 43–55.
58. Hennessy, J.L.; Patterson, D.A. *Computer Architecture: A Quantitative Approach*; Elsevier: Amsterdam, The Netherlands, 2011.
59. Bixby, R.E. Solving real-world linear programs: A decade and more of progress. *Oper. Res.* **2002**, *50*, 3–15. [CrossRef]
60. Mitchell, D.W. 88.27 more on spreads and non-arithmetic means. *Math. Gaz.* **2004**, *88*, 142–144. [CrossRef]
61. Smith, J.E. Characterizing Computer Performance with a Single Number. *Commun. ACM* **1988**, *31*, 1202–1206. [CrossRef]
62. Fleming, P.J.; Wallace, J.J. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Commun. ACM* **1986**, *29*, 218–221. [CrossRef]
63. AlGhamdi, Z.; Jamour, F.; Skiadopoulos, S.; Kalnis, P. A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs. In Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, 27–29 June 2017; pp. 6:1–6:12. [CrossRef]
64. Held, M.; Karp, R.M. The Traveling-Salesman Problem and Minimum Spanning Trees. *Oper. Res.* **1970**, *18*, 1138–1162. [CrossRef]
65. Git. 2005. Available online: https://git-scm.com/ (accessed on 21 March 2019).
66. GitHub. 2007. Available online: https://github.com/ (accessed on 30 June 2019).
67. GitLab. 2011. Available online: https://gitlab.com/ (accessed on 12 May 2019).
68. Bitbucket. 2008. Available online: https://bitbucket.org/ (accessed on 12 May 2019).
69. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* **2007**, *42*, 89–100. [CrossRef]
70. Reinders, J. *VTune Performance Analyzer Essentials*; Intel Press: Santa Clara, CA, USA, 2005.
71. Hamadi, Y.; Wintersteiger, C.M. Seven Challenges in Parallel SAT Solving. In Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Toronto, ON, Canada, 22–26 July 2012.

72. Kimmig, R.; Meyerhenke, H.; Strash, D. Shared Memory Parallel Subgraph Enumeration. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA, 29 May–2 June 2017; pp. 519–529. [CrossRef]
73. Goldberg, D. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48. [CrossRef]
74. YAML. The YAML Project. 2002. Available online: https://yaml.org/ (accessed on 22 March 2019).
75. German Research Foundation (DFG). *DFG Guidelines on the Handling of Research Data*; German Research Foundation: Bonn, Germany, 2015.
76. JSON. 2001. Available online: https://www.json.org/index.html (accessed on 8 June 2019).
77. Anscombe, F.J. Graphs in Statistical Analysis. *Am Stat.* **1973**, *27*, 17–21. [CrossRef]
78. Healy, K. *Data Visualization: A Practical Introduction*; Princeton University Press: Princeton, NJ, USA, 2018.
79. Sanders, P. Presenting Data from Experiments in Algorithmics. In *Experimental Algorithmics*; Fleischer, R., Moret, B., Schmidt, E.M., Eds.; Springer: Berlin, Germany, 2002; Volume 2547, Chapter 9, pp. 181–196.
80. Tufte, E.R. *The Visual Display of Quantitative Information*; Graphics Press: Cheshire, CT, USA, 2001.
81. Munzner, T. *Visualization Analyis and Design*; CRC Press: Boca Raton, FL, USA, 2014.
82. Ware, C. *Information Visualization: Perception for Design*, 3rd ed.; Morgan Kaufmann: Burlington, MA, USA, 2012.
83. Anderson, D.R.; Burnham, K.P.; Thompson, W.L. Null hypothesis testing: Problems, prevalence, and an alternative. *J. Wildl. Manag.* **2000**, *64*, 912–923. [CrossRef]
84. Trafimow, D.; Marks, M. Editorial. *Basic Appl. Soc. Psychol.* **2015**, *37*, 1–2. [CrossRef]
85. Wasserstein, R.L.; Lazar, N.A. The ASA's statement on p-values: Context, process, and purpose. *Am. Stat.* **2016**, *70*, 129–133. [CrossRef]
86. Lash, T.L. The harm done to reproducibility by the culture of null hypothesis significance testing. *Am. J. Epidemiol.* **2017**, *186*, 627–635. [CrossRef]
87. Cumming, G. The New Statistics: Why and How. *Psychol. Sci.* **2014**, *25*, 7–29. [CrossRef]
88. Kruschke, J.K.; Liddell, T.M. The Bayesian New Statistics: Hypothesis testing, estimation, meta-analysis, and power analysis from a Bayesian perspective. *Psychon. Bull. Rev.* **2018**, *25*, 178–206. [CrossRef]
89. Murtaugh, P.A. In defense of P values. *Ecology* **2014**, *95*, 611–617. [CrossRef]
90. Box, G.E.P. Science and Statistics. *J. Am. Stat. Assoc.* **1976**, *71*, 791–799. [CrossRef]
91. Pólya, G. Über den zentralen Grenzwertsatz der Wahrscheinlichkeitsrechnung und das Momentenproblem. *Math. Z.* **1920**, *8*, 171–181. [CrossRef]
92. Charnes, A.; Frome, E.L.; Yu, P.L. The Equivalence of Generalized Least Squares and Maximum Likelihood Estimates in the Exponential Family. *J. Am. Stat. Assoc.* **1976**, *71*, 169–171. [CrossRef]
93. Fisher, R.A. Statistical methods for research workers. In *Breakthroughs in Statistics*; Springer: Berlin, Germany, 1992; pp. 66–70.
94. Young, G.A.; Smith, R.L. *Essentials of Statistical Inference*; Cambridge University Press: Cambridge, UK, 2005; Volume 16.
95. Wilcoxon, F. Individual comparisons by ranking methods. *Biomed. Bull.* **1945**, *1*, 80–83. [CrossRef]
96. Jones, E.; Oliphant, T.; Peterson, P. SciPy: Open Source Scientific Tools for Python. 2001. Available online: https://www.scipy.org (accessed on 14 June 2019).
97. Dunn, O.J. Multiple comparisons among means. *J. Am. Stat. Assoc.* **1961**, *56*, 52–64. [CrossRef]
98. Holm, S. A Simple Sequentially Rejective Multiple Test Procedure. *Scand. J. Stat.* **1979**, *6*, 65–70.
99. Smithson, M. *Confidence Intervals*; SAGE Publications, Inc.: Thousand Oaks, CA, USA, 2003.
100. Neyman, J. Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability. *Philos. Trans. R. Soc. Lond. A* **1937**, *236*, 333–380. [CrossRef]
101. Gelman, A.; Stern, H.S.; Carlin, J.B.; Dunson, D.B.; Vehtari, A.; Rubin, D.B. *Bayesian Data Analysis*; Chapman Press: Boca Raton, FL, USA, 2013.
102. Salvatier, J.; Wiecki, T.V.; Fonnesbeck, C. Probabilistic programming in Python using PyMC3. *PeerJ Comput. Sci.* **2016**, *2*, e55. [CrossRef]
103. Jeffreys, H. *The Theory of Probability*; Oxford University Press: Oxford, UK, 1998.

104. Von Looz, M.; Meyerhenke, H.; Prutkin, R. Generating random hyperbolic graphs in subquadratic time. In Proceedings of the International Symposium on Algorithms and Computation, Nagoya, Japan, 9–11 December 2015; pp. 467–478.
105. Akaike, H. A new look at the statistical model identification. *IEEE Trans. Autom. Control* **1974**, *19*, 716–723. [CrossRef]