SURVEY

Methodology of Algorithm Engineering

JAN MENDLING, Humboldt University of Berlin, Berlin, Germany

HENRIK LEOPOLD, Kühne Logistics University, Hamburg, Hamburg, Germany

HENNING MEYERHENKE, Karlsruhe Institute of Technology, Karlsruhe, Baden-Wurttemberg, Germany

BENOÎT DEPAIRE, Hasselt University, Hasselt, VLI, Belgium

Open Access Support provided by:

Karlsruhe Institute of Technology

Hasselt University

Humboldt University of Berlin

Kühne Logistics University

Published: 25 October 2025
Online AM: 22 September 2025
Accepted: 15 September 2025
Revised: 06 August 2025
Received: 30 October 2023

Methodology of Algorithm Engineering

JAN MENDLING, Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany, Vienna University of Economics and Business, Wien, Austria, and Weizenbaum Institute, Berlin, Germany
HENRIK LEOPOLD, Kühne Logistics University, Hamburg, Germany and Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
HENNING MEYERHENKE, Karlsruhe Institute of Technology, Karlsruhe, Germany
BENOÎT DEPAIRE, Hasselt University, Hasselt, Belgium

Research on algorithms has drastically increased in recent years. Various sub-disciplines of computer science investigate algorithms according to different objectives and standards. This plurality of the field has led to various methodological advances that have not yet been transferred to neighboring sub-disciplines. The central roadblock for a better knowledge exchange is the lack of a common methodological framework integrating the perspectives of these sub-disciplines. It is the objective of this article to develop such a research framework for algorithm engineering. Our framework builds on three areas discussed in the philosophy of science: ontology, epistemology and methodology. The framework helps us to identify and discuss various *validity concerns* relevant for any contribution on algorithms in various areas of computer science.

CCS Concepts: • **General and reference → Surveys and overviews**; **Evaluation**; **Validation**; **Experimentation**; **Empirical studies**; • **Theory of computation → Design and analysis of algorithms;**

Additional Key Words and Phrases: Algorithms; algorithm engineering; evaluation of algorithms; design and analysis of algorithms

ACM Reference Format:
Jan Mendling, Henrik Leopold, Henning Meyerhenke, and Benoît Depaire. 2025. Methodology of Algorithm Engineering. *ACM Comput. Surv.* 58, 4, Article 94 (October 2025), 38 pages. https://doi.org/10.1145/3769071

1 Introduction

The design, analysis, and evaluation of algorithms have been a major concern of computer science since its founding days. Algorithms constitute a specific class of technological rules [Bunge 1967]. An *algorithm* is a sequence of computational steps that transforms some input into some output [Cormen et al. 2009]. The number of steps, the effort they require, and the time they take have to be finite [Aho et al. 1974]. Research on algorithms builds on foundationals work by Dijkstra [1968] and Knuth [1974]. The corresponding subdiscipline of computer science is called *algorithmics*, of which

Authors' Contact Information: Jan Mendling (corresponding author), Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Berlin, Germany and Vienna University of Economics and Business, Wien, Wien, Austria andWeizenbaum Institute, Berlin, Berlin, Germany; e-mail: jan.mendling@hu-berlin.de; Henrik Leopold, Kühne Logistics University, Hamburg, HH, Germany, Hasso Plattner Institute and University of Potsdam, Potsdam, Brandenburg, Germany; e-mail: Henrik.Leopold@the-klu.org; Henning Meyerhenke, Karlsruhe Institute of Technology, Karlsruhe, BW, Germany; e-mail: meyerhenke@hu-berlin.de; Benoît Depaire, Hasselt University, Hasselt, Flanders, Belgium; e-mail: benoit.depaire@uhasselt.be.

ACM 0360-0300/2025/10-ART94
https://doi.org/10.1145/3769071

algorithm engineering [Sanders 2009] is a branch that embraces both theoretical and empirical contributions, thus, striving for a symbiosis [Mitzenmacher 2015; Ullman 2015].

Various efforts have been made to build an inventory of algorithms. Important works include books by Knuth [1968, 1969, 1973], textbooks by Aho et al. [1974]; Cormen et al. [2009]; Harel and Feldman [2004]; Sedgewick and Flajolet [2013], and review books by Ferragina [2023]; Lange [2020]. Black et al. [2020] lists more than 300 algorithms covering areas such as automata, combinatorics, cryptography, memory management, geometry, graphs, numerical computation, parallel processing, quantum computation, searching, sorting, trees, and verification methods. This list is far from complete though. Today, researching algorithms is not only the focus of algorithmics and algorithm engineering, but also of various other sub-disciplines of computer science and neighboring research fields. For 2024 only, Google Scholar lists more than 62,000 research papers with the term *algorithm* in the title, highlighting the importance and the richness of research on algorithms.

We observe that a review of the *theory and practice of algorithm engineering* is missing. Much of methodological innovations are developed within specific conference series such as Very Large Databases (VLDB), Neural Information Processing Systems (NeurIPS), European Symposium on Algorithms (ESA), or Visualization (VIS). Innovations are partly fostered internally without being transferred to or adapted by neighboring communities. Is this pluralism an indication that there is no consensus in the field of computer science? Kuhn [1962] calls such a consensus *normal science* to signify the incremental work in an established paradigm, and interprets it as a sign of maturity of a field. However, the requirements, e.g., for evaluating algorithmic contributions differ quite considerably across the above-mentioned conferences: papers published at VIS put more emphasis on user studies, papers published at VLDB focus on performance evaluation and formal correctness. While there are good reasons for these differences, the question is whether a common scientific framework can integrate different requirements, such that it might help other sub-disciplines of computer science to adopt methodological advances.

Against this background, we aim at explicating diverse perspectives on researching algorithms by the help of an integrated theoretical framework. To that end, we take foundational concepts discussed in the philosophy of science as a starting point. Our goal is to develop a theoretical framework along three dimensions: ontology, epistemology, and methodology. In essence, *ontology* is concerned with "what is". For algorithm engineering, this means we need to clarify what the phenomena are that we consider. *Epistemology* deals with the nature of knowledge about a specific phenomenon: "what can we know about algorithms". *Methodology* refers to the study of method: "how can we systematically enhance our knowledge" of specific algorithms. With our framework, we aim at providing an answer to these questions and facilitate learning across different domains of computer science. In this way, our framework might eventually strengthen the *cumulative tradition* [Keen 1980] in computer science.

We proceed as follows. Section 2 presents the ontological perspective of our framework. Section 3 describes its epistemological perspective. Section 4 discusses the methodological perspective. Section 5 distills recommendations for algorithm engineering. Section 6 concludes our article.

2 Ontology of Algorithm Engineering

Ontology is the study that is concerned with the structure of the real world [Bunge 1977; Wand and Weber 1990]. An ontological theory of engineering clarifies the structure of real-world phenomena that relate to engineering. In general, *engineering* is defined as the practice of organizing the design, construction, and operation of any artifact that transforms the world around us for meeting a recognized need [Rogers 1983; Vincenti 1990]. Staples [2014] emphasizes that engineering deals with artifacts (algorithms in our context), that artifacts are meant to meet specific requirements, and that engineering builds on theories explaining why specific artifacts meet certain requirements.

In turn, *algorithm engineering* has been defined as a discipline that focuses on the design, analysis, implementation, tuning, debugging, and experimental evaluation of algorithms [Demetrescu et al. 2004; Sanders 2009]. The salient feature of this definition is the term *discipline*. It indicates that the design of algorithms is not the end of algorithm engineering, but the phenomenon that is studied. This means that its end is the generation of scientific knowledge about algorithms. While an algorithm design is an ontological entity in the realm of algorithm engineering practice, mind that knowledge about it is already an epistemological entity in the realm of the algorithm engineering discipline. Also note that algorithms are often parts of larger *techniques* and *systems* [Munzner 2014]. These are more coarse granular entities, studied in disciplines such as information systems engineering [Castro et al. 2002; Sage 1992] and design science [Peffers et al. 2007; Wieringa 2014].

The *ontological perspective* of algorithm engineering refers to the entities associated with the practice of engineering algorithms. Here, we build upon the ontological model of Staples [2014]. This model identifies ontological entities in a precise and explicit way according to principles generally accepted in computer science [Aho et al. 1974] and operations research [Landry et al. 1983]. These ontological entities have specific relationships with each other.

(1) The first ontological entity is a *real-world problem*, which resides in a specific problem context and which hints at an algorithm as part of an envisioned solution.
(2) This real-world problem is in an abstraction relationship to an *algorithmic task*. This task conceptualizes the real-world problem and captures its essential assumptions and requirements.
(3) Algorithmic tasks are in a satisfaction relationship with *algorithm designs*. A design addresses the algorithmic task and incorporates design principles and design decisions. It can be characterized in terms of the performance guarantees it provides, e. g. derived from *algorithm analysis*.
(4) An algorithm design can be instantiated as one or many *algorithm implementations*. An implementation reflects implementation decisions and materializes the design. The execution of the algorithm implementation on data from the real-world problem generates *results*. The implementation provides a specific output to the given real-world problem, as well as an indication of the algorithm's empirical performance.

Figure 1 provides an overview of our ontological framework. We discuss its four ontological entities in turn. For illustration proposes, we use examples from different areas of computer science including, among others, sorting, shortest-path identification, and natural language processing.

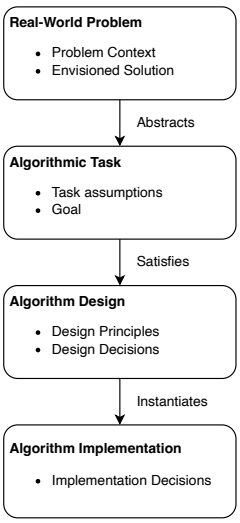

Fig. 1. A framework for algorithm engineering: The ontological perspective.

2.1 Real-World Problem

The motivation for engineering algorithms can be found in real-world problems. Real-world problems are concrete, they are associated with dynamic situations and complex systems. They are often underspecified and their boundaries are fuzzy so that Meyer et al. [2015] rather speak of a problem domain. Ackoff [1979] emphasizes this aspect by referring to real-world problems as *messes*. Real-world problems range from managerial problems to technical problems.

Example 2.1. The textbook "*The Art of Computer Programming Volume 3: Sorting and Searching*" by Knuth [1973] describes several technical real-world problems that motivate the definition and analysis of sorting algorithms. "*If several files have been sorted into the same*

order, it is possible to find all of the matching entries in one sequential pass through them, without backing up."

Example 2.2. The textbook "*Tools for Thinking: Modelling in Management Science*" by Pidd [2003, p.62] describes several real-world problems, among others, a supermarket chain. This "*[...] supermarket chain must constantly have a full product offering on its shelves, must provide good customer service, must attract customers, must maintain low stocks and must make a profit. As people strive to achieve these aims, this throws up a continuing stream of issues through time. Examples might be the extent to which EPOS [Electronic Point of Sale] systems should be installed and used.*"

To solve a particular real-world problem, we can use an existing or a new algorithm design. This, however, requires to first translate the real-world problem into an algorithmic task.

2.2 Algorithmic Task

Many real-world problems are deeply intertwined with the specifics of a problem domain. Modeling and analysis is required for structuring these problems, and various methods of management science and requirements engineering support such analysis [De Gea et al. 2012; Pidd 2003]. The result of problem-structuring is an *abstraction* of the original real-world problem. Also the original messiness is resolved towards a precise formulation of what we call an *algorithmic task*. Algorithmic tasks are also called *algorithmic problems*, *computational problems*, or just *problems* [Aho et al. 1974]. Algorithmic tasks abstract from specific real-world problems, they represent *classes of problems* [Aken 2004]. The more abstract an algorithmic task is formulated, the broader is the extension of real-world problems it subsumes. Over time, various algorithmic tasks have become standard tasks that are covered in textbooks, while new ones are continuously identified.

Example 2.3. Knuth [1973] describes several classical algorithmic tasks in his textbooks, among others sorting. He defines sorting as a specific task that considers as input a list of n records and an ordering relation $<$ over the keys of these n records. The goal of sorting is then to determine a permutation of the records that is consistent with the ordering relation [Knuth 1973, p. 5].

The example highlights that algorithmic tasks explicate *assumptions* regarding the input and the processing of the input. The *goal* for the algorithmic tasks specifies what the algorithm is meant to achieve in terms of output and performance. Also the supermarket chain might formulate its replenishment problem as an algorithmic task. Given the assumption that the input is a weighted graph without negative edges representing shops as vertices and roads as edges, the goal could be to find the shortest path in the graph covering all vertices in a minimum amount of processing time. Often, algorithmic tasks can be reformulated with different assumptions. Stronger assumptions can make tasks more specific. Also new algorithmic tasks are continuously introduced.

Example 2.4. Black et al. [2020] defines the task of finding the shortest path as the "*problem of finding the shortest path in a graph from one vertex to another. Shortest may be least number of edges, least total weight, etc.*" There is not just one algorithmic task for the shortest-path problem. Deo and Pang [1984] describe a classification of shortest-path tasks, distinguishing usual path length and generalized path length with altogether 12 different variants.

Example 2.5. Bevilacqua et al. [2021] review contributions on word sense disambiguation. In essence, word-sense disambiguation algorithms map a word to its most likely sense in the context of a sentence or text [Navigli 2009]. Bevilacqua et al. [2021] point to related algorithmic tasks that have been defined in recent research. For instance, the word-in-context task requires as input two contexts to provide a prediction whether the same target words are used with the same meaning.

It is desirable that tasks are fully specified. In such a case, all required information can be provided as input and an algorithm determines the output fully automatically [Sedlmair et al. 2012]. Sorting belongs to this category of tasks. Tasks can also be fuzzy in the sense that required input cannot be provided at the required level of precision. In this case, interactive techniques can be designed that integrate algorithmic components in such a way that a user can complement the output with background knowledge [Meyer et al. 2015]. Such fuzzy tasks are often addressed by research on computer visualization and natural language processing. For instance, word sense disambiguation is a fuzzy task since the semantics of natural language can hardly be fully specified. Ultimately, a key challenge is to specify the algorithmic task in such a way that an algorithm design is feasible.

2.3 Algorithm Design

Algorithm designs relate to algorithmic tasks. If appropriately developed, a design is in a *satisfaction* relationship with a task. While the algorithmic task essentially describes *what* is to be done and in which context (often by means of an input/output specification), the algorithm design specifies *how* it works, that is, how to obtain the desired output. The act of designing considers the task's assumptions and yields a design specification that meets the goal of the task [Wieringa 2014]. The algorithm design implies that certain performance guarantees hold.

Designing is guided by knowledge [Staples 2014] on why algorithm designs meet specific goals. Design principles inform the design of novel algorithms. In essence, a design principle describes a solution strategy [Gregor et al. 2020]. Specific solution strategies such as divide-and-conquer, dynamic programming, backtracking, or local search [Aho et al. 1974; Harel and Feldman 2004; Kleinberg and Tardos 2006] are available as a starting point for designing new algorithms.

Example 2.6. A **Vehicle Routing Problem** (**VRP**) relates to the identification of the optimal route from one or several depots to several geographically scattered customers subject to constraints. Laporte [1992] discusses exact algorithms and heuristics to solve the VRP. He identifies three solution strategies—direct tree search, dynamic programming, and integer linear programming—and discusses various designs for each solution strategy. He also identifies solution strategies for heuristic algorithms such as the nearest neighbor, insertion, and tour improvement procedure.

A *design decision* is a decision that has the potential to change the behavior and performance of the design. Some design decisions are explicit, others are implicit. In fact, every design is built on numerous implicit design decisions. For instance, when the algorithm is specified using Pascal-like pseudo code, this implies the assumption that the algorithm is likely to be implemented using a procedural programming language. This also entails assumptions about the machine executing the programming code [Sanders 2009]. From a scientific perspective, mainly the explicit decisions are of interest as they have been made intentionally and can be expected to have the highest impact on the design. Such a design decision may involve using a particular design principle (e.g., divide-and-conquer), paradigm (e.g., supervised learning or deep learning), or representation (e.g., word embeddings). Parameterization is an explicit design decision, as it delegates the actual decision to the end-user. A (hyper-)parameter is provided to control the actual behavior and performance.

Example 2.7. In their article "*Attention is all you need*," Vaswani et al. [2017] introduce the Transformer architecture. They explicitly discuss various design choices. Among others, they explain that building soley on the Attention mechanism is enough for the parallelization within training examples. They also discuss that the use of Attention may lead to "*reduced effective resolution*," which they counter by using Multi-Head Attention. Later in the article, they devote an entire section (*Why Self-Attention*) to demonstrate the impact of their design choices.

Design decisions and the usage of design principles have implications for performance. Algorithm analysis can clarify these implications and associate algorithm designs with *performance guarantees* [Sanders 2009; Sedgewick and Flajolet 2013]. Such guarantees can be formulated as theorems.

Example 2.8. Knuth [1973] provides an overview of sorting algorithms that use different solution strategies. The family of merge sort algorithms follows the design principle of divide-and-conquer. Its solution strategy is to decompose a problem into a hierarchy of partial problems in order to achieve an $\mathcal{O}(n \log n)$ run-time performance.

2.4 Algorithm Implementation

According to Colburn [2004], the ultimate goal of computer science is that programs can be efficiently implemented on physical systems. An algorithm design can be in an *instantiation* relationship with several algorithm implementations. The implementation program closes the circle: it can be used as a concrete solution to address the original real-world problem by executing it with concrete input data on a specific computer.

In practice, implementation rarely involves an exact instantiation [Lukyanenko and Parsons 2020]. Instead, it is shaped by concrete *implementation decisions* made to achieve specific goals based on a given rationale. For example, the choice of programming language affects whether an algorithm can be directly translated into code. At runtime, performance is further influenced by the execution environment, including hardware, memory hierarchy [Sanders 2004], caching strategies [Karedla et al. 1994], and compiler optimizations for parallel processing [Wolfe 1996]. Even under controlled conditions, differing implementation choices can result in performance variations by orders of magnitude.

Example 2.9. Implementation decisions may have a drastic impact on algorithm performance, as evidenced by a study of Kriegel et al. [2017]. The authors compare different implementations of the same algorithm designs. In their analysis of the reasons behind performance differences, they describe that certain implementations turned out be highly optimized. When re-implementing a k-means algorithm for a comparative evaluation, they "*chose the features with only eight dimensions [...] since k-means is known to work better in low dimensionality and we do not want to bias the analysis toward performance on high-dimensional datasets.*"

Executing an algorithm on real-world data yields two types of results: the actual *output* and insights into its *performance*. For instance, a web search algorithm returns a list of websites relevant to a query. Performance can be assessed along two dimensions: efficiency (e.g., runtime and memory usage) and effectiveness (i.e., output quality relative to the problem). Suitable performance metrics vary by context. For web search, for instance, common measures include precision, recall, and F1-score [Baeza-Yates et al. 1999]. Whether performance is deemed sufficient depends on the predefined goals of the task, e.g., outperforming a baseline like Google Search on specific metrics.

Example 2.10. Devlin et al. [2014] present a new neural network joint model for machine translation. Their core idea is to determine a source context window, i.e., the number of words from the source language that are relevant to find the correct word of the target language. To evaluate the *performance*, the authors conduct extensive experiments on Arabic-English and Chinese-English datasets. They show that their approach is superior in comparison to existing baselines both in terms of *effectiveness* and *efficiency*. For quantifying efficiency, they use the metrics *lookups/second* and *seconds/word*. They quantify *efficiency* using the *BLEU* score, an established measure of the similarity between machine-translated text and a reference translation.

3 Epistemology of Algorithm Engineering

The previous elaborations emphasize that the practice of algorithm engineering is concerned with finding an algorithm implementation for a given real-world problem. The science of engineering

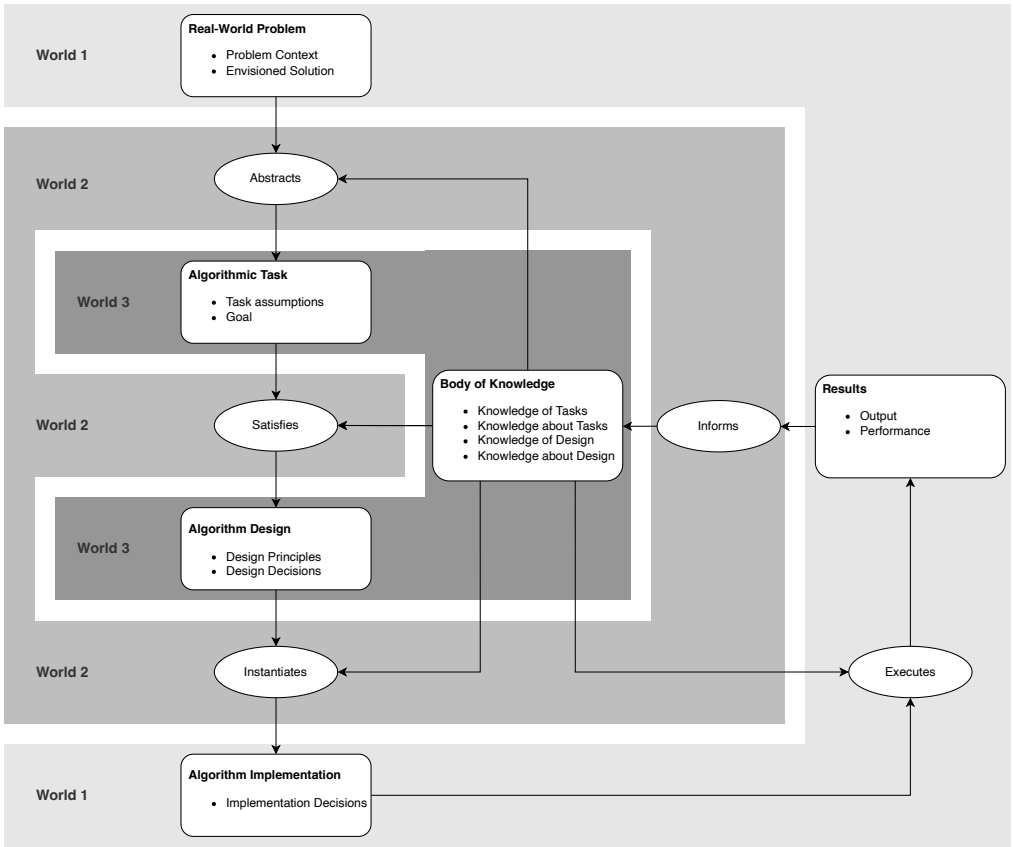

Fig. 2. A framework for algorithm engineering: the epistemological perspective.

takes a more abstract view. It is not about solving individual problems, but about "*general knowledge, linking an intervention or artifact with a desired output or performance in a certain field of application*" [Aken 2004]. Thus, algorithm engineering aims to expand the body of knowledge on algorithm design. The *epistemological* perspective of our framework addresses questions of what we can know about an algorithm. We use Popper's three worlds as a conceptual model to clarify the relationship between knowledge and algorithms. Note that we are not the first ones to build on Popper's model. It has been used by Naur [1985] and Staples [2014]. We describe that the body of knowledge contains knowledge *of* and *about* tasks as well as knowledge *of* and *about* designs.

3.1 Popper's Three Worlds

The body of knowledge of algorithm engineering relates to different spheres that Popper [1979] describes in his epistemological model of three worlds. In essence, he distinguishes World 1 of physical entities, World 2 of subjective mental states, and World 3 of objective knowledge. World 1, often referred to as the real world, is assumed to exist independently from an observer and to behave in a regular way. The goal of science is to formulate objective, empirical knowledge in World 3 that refers to World 1. Such scientific knowledge is created by humans, whose individual mental states define World 2, mediating between the other two worlds.

Figure 2 extends our ontological perspective on algorithm engineering and shows how it relates to Popper's three worlds. Our discussion builds on arguments by Naur [1985] and Staples [2014].

First, both real-world problems and algorithm implementations are entities that exist in the physical world independently from an observer; they belong to World 1. Second, the three actions abstract, design, and implement build on human cognition and mental models. For this reason, they reside in World 2. Third, the algorithmic tasks, algorithm design, and the body of knowledge are part of World 3 as these are meant to be objectively described. Regarding the body of knowledge, we distinguish between knowledge related to tasks and knowledge related to designs. For both types we further differentiate between *knowledge of* and *knowledge about*. In the context of tasks, *knowledge of* relates to *what is* a specific task, while *knowledge about* relates to *what are properties* we know about these tasks. In a similar way, knowledge of design relates to *what is* a specific algorithm design, while *knowledge about* relates to *what are properties* we know about this algorithm design.

Popper's three world view unveils an important implication for algorithm engineering that is easily overlooked: Any scientific knowledge resides in World 3. The same observation applies to algorithm engineering and its body of knowledge. Mind that our understanding of a specific real-world problem or a specific algorithm implementation is ontologically different: both reside in World 1. World 1 is what matters from a practical perspective, whereas World 3 is what we are interested in from a scientific perspective. In other words, from a practical perspective we are interested in real-world problems and algorithm implementations, whereas from a scientific perspective we focus on knowledge of and about algorithmic tasks and designs. We now discuss in turn what knowledge of and about tasks as well as knowledge of and about algorithmic designs can be formulated.

3.2 Knowledge of Tasks

Real-world problems reside in World 1 and are often *wicked problems* [Rittel and Webber 1973]. This wickedness implies that such problems are unique, having no definitive formulation, no explicit stopping rule, no ultimate test of solution, and affording numerous alternative explanations. Their wicked nature is the reason why algorithmic tasks are not connected with a problem via a simple isomorphic mapping. According to Rittel and Webber [1973], they have to be tamed by constructing an abstraction. We refer to the outcome of this abstraction process as *knowledge of tasks*. Knowledge of tasks, therefore, captures what tasks exist and how they are structured. As pointed out earlier, keep in mind that our notion of an algorithmic task is also referred to as *algorithmic problem*, *computational problem*, or just *problem* in the literature [Aho et al. 1974]. These are tasks (residing in World 3) and should, however, not be confused with our notion of a real-world problem (residing in World 1). Our notion of algorithmic tasks and World 3 problems are semantically equivalent. To avoid confusion, we will use the term *(algorithmic) task* throughout this article. For established tasks, such as the *traveling salesperson problem*, we will, however, adopt the original term.

Depending on the context, tasks and their structure might be described mathematically, formally, semi-formally, or also informally.

Example 3.1. The *traveling salesperson problem* can be described mathematically, modelled as an optimization problem. Hromkovič [2013, p.104] defines it as follows:

- *Input*: a weighted complete graph (G, c) where $G = (V, E)$ and $c : E \mapsto \mathbb{N}$. Let $V = \{v_1, \dots, v_n\}$ for some $n \in \mathbb{N} - \{0\}$.
- *Constraints*: For every input instance (G, c), $\mathcal{M}(G, c) = \{\langle v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1}\rangle \mid (i_1, i_2, \dots, i_n$ is a permutation of $(1, 2, \dots, n)\}$, i.e., the set of all Hamiltonian cycles of G.
- *Costs*: For every Hamiltonian cycle $H = \langle v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1}\rangle \in \mathcal{M}(G, c)$, the cost is induced by the cost function $cost(\langle v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_{i_1}\rangle, (G, c)) = \sum_{j=1}^{n} c(\{v_{i_j}, v_{i_{(j \bmod n)+1}}\})$.
- *Goal*: Minimum.

Example 3.2. Van der Aalst [2011] formally introduces the task of *process discovery* in the context of process mining. Informally, this task can be described as identifying a function that constructs a process model from an event log (i.e., a set of execution sequences) such that the model is representative for the behavior observed in the event log. Formally, he first clarifies the notion of an event log: A simple event log L is a multi-set of traces over a set of activities $\mathcal{A}$, i.e., $L \in \mathcal{B}(\mathcal{A}^*)$. Building on this definition, he then formally describes the process discovery task as identifying a function γ that maps an event log $L \in \mathcal{B}(\mathcal{A}^*)$ onto a Petri net N with a marking M, i.e., $\gamma(L) = (N, M)$. It is important to note that such formal task descriptions often contain text. As a result, the degree of formality can vary and the transition to fully informal task descriptions (see example below) is often fluid.

Example 3.3. Russakovsky et al. [2015] list several *image recognition* tasks in relation to ImageNet. They use an informal, textual description to specify the *object detection* task. The goal of an algorithm addressing this task is to produce bounding boxes at the right position and scale for all instances of a given object category. The effectiveness of algorithms is evaluated by recall (the share of correctly detected target object instances) and precision (the share of spurious detections).

3.3 Knowledge about Tasks

Knowledge about tasks relates to insights about properties of specific algorithmic tasks. The analysis of tasks can yield formal knowledge and empirical knowledge.

Formal knowledge has been established for a wide range of tasks. These tasks are often referred to as computational problems. A key concern in the analysis of tasks is the question whether a problem is tractable. An *intractable problem* can be solved in theory, but any algorithm in practice would take too much time or space on inputs of non-trivial size [Hopcroft et al. 2000].

Complexity classes have been identified for specific tasks. A hierarchy of these classes ranges from non-deterministic with logarithmic memory space **NL** to exponential space **EXPSPACE**. Many classical problems have been identified as members of the class that can be solved in polynomial time on a non-deterministic Turing machine **(NP)** [Garey and Johnson 1979]. An important subclass is the set of **NP-complete** problems; loosely speaking, its members are at least as difficult to solve as any other problem in **NP**. A more fine-grained analysis allows to reason about fixed-parameter tractable problems **(FPT)**, which can be solved in time $f(k) \cdot |(x,k)|^{O(1)}$ for some computable function f, where k is some parameter of the input [Cygan et al. 2015].

Empirical knowledge is of practical interest in particular for problems that have been identified as intractable. Key to this knowledge is the identification of patterns in typical input data that can be exploited for the design of algorithms. Improvements on many algorithmic tasks have been inspired by the availability of *benchmark datasets*. These datasets are publicly shared datasets of realistic input of a specific algorithmic task.

Example 3.4. A classical algorithmic task that is known to belong to the class of **NP-complete** problems is the traveling salesperson problem (see Example 3.1). It assumes a list of cities and distances between them. In its optimization version, the question is *what is the shortest route that visits all cities and returns to the origin?* (Its decision version asks if such a route of a specified length or less exists.) Formal knowledge on this task has been established among others by Papadimitriou [1977], who proves that the Euclidean travelling salesperson problem is **NP-complete**. Major breakthroughs on TSP have been inspired by the availability of realistic problem instances and corresponding benchmark datasets [Sanders 2009], such as the 8th DIMACS Implementation Challenge or the World TSP Tour. By help of these datasets, empirical knowledge about properties of the input data could be established and exploited, for instance by considering power-law distributions [Ouaarab et al. 2014]. In many domains of algorithm engineering, datasets such as those

for TSP [Drori et al. 2020] are continuously extended to challenge improvements and innovation of algorithms and knowledge about tasks.

Example 3.5. The understanding of algorithmic tasks in the field of image recognition is driven by large-scale annotated image datasets such as ImageNet [Russakovsky et al. 2015]. ImageNet contains several million images that have been manually annotated with names of objects. Research on ImageNet has not only led to major advancements of algorithms for image recognition, but also of techniques for systematically annotating a large amount of images using crowdsourcing.

3.4 Knowledge of Design

Knowledge of design refers to what an algorithm design is, how it works, and how it addresses an algorithmic task. Such knowledge of design is *prescriptive*, as enables the implementation of concrete algorithms by humans. Gregor and Hevner [2013] refer to it as Λ knowledge. The body of knowledge that relates to design largely consists of a repertoire of *established algorithm designs* that have demonstrated their efficiency and effectiveness in prior studies, i.e., where formal or empirical knowledge indicates that an algorithm design meets the goals defined for the algorithmic task. Established algorithm designs range from concrete designs that address specific tasks to general procedures that are applicable for a wide range of tasks. Regardless of the scope, algorithm designs may build on generic *design principles*, which describe general ideas for finding or constructing a solution for a given algorithmic task. Among others, Harel and Feldman [2004] as well as Kleinberg and Tardos [2006] describe several examples of such principles including divide-and-conquer. To illustrate the broad range of algorithm designs, consider the following four examples.

Example 3.6. Grund et al. [2021] introduce an approach that identifies earlier versions of a considered source code method in the file history. Its central artifact is a specifically designed algorithm that addresses this specific task in the context of software engineering and source code management.

Example 3.7. Gévay et al. [2021] introduce a novel distributed dataflow system called *Mitos* that helps to efficiently coordinate the distributed execution of control flow in a user-friendly fashion. The proposed system consists of several different components that are also separately introduced. Some parts are presented at the algorithmic level, while others are discussed conceptually. As for the scope, the system can be used for a range of analysis tasks that benefit from distributed execution.

Example 3.8. Bentley [1980] introduces an approach that targets problems dealing with collections of objects in a multidimensional space. The author positions his approach as a generic algorithmic design as it can be used to give best-known solutions to several problems including range searching, closest pair, and nearest neighbor. For a more general setting, Bentley [1999] presents a set of established and frequently used algorithm design techniques for solving algorithmic tasks in practice.

Example 3.9. In a collection on design techniques, Ferragina [2023] selects algorithmic tasks that admit "*surprisingly elegant [design] solutions that can be described in a few lines of code*." The topics covered are numerous and include random sampling, list ranking, sorting, string search, and compression. The variety of topics is also reflected by the variety of algorithm design principles presented.

Knowledge of what a *concrete algorithm design* is can be explicated in different ways. At least three different levels can be distinguished based on their proximity to the implementation. First, the algorithm's design can be described using pseudocode. Pseudocode is more abstract and more compact

than code and yet typically facilitates the re-implementation of the design in various programming languages. Tool support is available for generating pseudocode for actual implementations [Oda et al. 2015]. Second, diagrams such as flow charts can be used to describe the algorithm design. Such diagrams provide benefits in terms of cognitive effectiveness [Malinova Mandelburger and Mendling 2021] and they have been found to be superior in terms of comprehension in comparison to pseudocode [Scanlan 1989]. Third, algorithms can be described using formal specifications, e.g., by mathematical formulae. It is worth noting that not every type of specification is equally useful for any type of design. While specific algorithms can be well described using pseudocode or formal specifications, this does not apply to larger systems consisting of several algorithmic subcomponents. Below, we discuss a number of examples to illustrate how different specifications are used.

Example 3.10. Many algorithm engineering papers and textbooks use pseudocode (e.g., Ferragina [2023]; Sanders et al. [2019]). Grund et al. [2021] also use pseudocode to describe how the proposed approach identifies earlier versions of a considered source code method in the file history.

Example 3.11. Chen et al. [2020] use a flowchart for clarifying the data processing pipeline of their system. This system builds on a machine-learning-based approach that helps detecting vulnerabilities in open-source libraries.

Example 3.12. Zhou et al. [2021] use mathematical equations as formal specifications to describe their method for long sequence time series forecasting. They also use this specification as a basis to prove relevant properties of the proposed method.

3.5 Knowledge about Design

Knowledge about design relates to insights about properties and characteristics of the respective algorithm design. The relationship between algorithmic tasks and designs is often explicated as specific *hypotheses*. In their deductive-nomological model, Hempel and Oppenheim [1948] describe a hypothesis as a law bound to a set of conditions. This follows the structure of an *explanans* explaining an *explanandum*, a specific observation. For algorithm engineering, the explanandum is often the extent to which an algorithm *satisfies* the requirements defined by the algorithmic task, the explanans refers to the design principles realized by the algorithm design and the assumptions made by the algorithmic task [Hall and Rapanotti 2017; Simon 1969; Staples 2014; Wieringa 2014]. The algorithmic task and the algorithm design directly correspond to our ontological framework described earlier in Figure 1. Knowledge about design can be substantiated in a formal and in an empirical way.

Example 3.13. To gain formal knowledge on the performance of an algorithm design, Ferragina [2023] uses (among others) the external-memory (EM) model for analysis. This model focuses on the I/O bottleneck in computations by assuming a two-level memory hierarchy with a fast (yet small) internal memory and a large (yet slow) external memory. Data is retrieved from the external memory in chunks, which mimics block transfers in real systems. In this way, as a *golden rule* [Ferragina 2023, Ch. 1], algorithm designs are steered towards exploiting locality.

Example 3.14. Karlin et al. [2021] introduce a new approximation algorithm for the travelling salesperson problem. They use theorems and proofs to show that, given a constant ϵ, their algorithm delivers a solution that has at most $\frac{3}{2} - \epsilon$ times the cost of the optimal solution.

Example 3.15. Gévay et al. [2021] use a plot to demonstrate how the execution time of their proposed system *Mitos* changes with increasing size of the input.

There are different *types* of knowledge about design. Following work by Santner et al. [2003] on the design and analysis of computer experiments, we distinguish four general knowledge types that

are often investigated in the field of algorithm engineering: performance, sensitivity, uncertainty, and explanatory knowledge.

Performance knowledge focuses on the algorithm's performance in terms of the extent that the algorithm *meets its task requirements*. Performance knowledge can be formulated as a question of satisfaction (Does the algorithm design *satisfy* the task requirements?) or as a question of degree (To which extent does the algorithm design satisfy the task requirements *better* than other designs?) [Hall and Rapanotti 2017; Staples 2014; Wieringa 2014]. Often, knowledge claims on performance refer to general criteria [Larsen et al. 2025] or specific measurements. Research focused on performance knowledge often relies on a competition-based evaluation or formal proofs against state-of-the-art algorithms.

Example 3.16. Gévay et al. [2021] introduce the system *Mitos* for task analysis. The key feature of the proposed system is that it is considerably faster than the state of the art.

Example 3.17. Küçük et al. [2021] introduce an approach for statistical fault detection in source code. The key feature of the proposed approach is that it performs considerably better than the state of the art with respect to fault localization.

Sensitivity knowledge focuses on how robust the performance of an algorithm is in face of changes of *internal* design decisions. These design decisions are often parameterized and the goal is to evaluate the robustness of the algorithm's performance against sub-optimal parameter settings.

Example 3.18. Chen et al. [2020] propose a machine-learning approach for detecting vulnerabilities in open-source libraries. To this end, they build on word embeddings from word2vec. As word2vec relies on a number of parameters (e.g., window size and vector size), they systematically explore how the performance of the approach is affected by changing parameters for different datasets.

Uncertainty knowledge considers the performance of an algorithm as a function of the task assumptions, which describe the relevant aspects of the *external* environment of an algorithm. Larsen et al. [2025] use the term context claims to refer to such knowledge. The goal of uncertainty analysis is to assess how the expected performance of an algorithm varies across problem instances. Alternatively, the goal could also be to investigate the precision of the expected performance.

Example 3.19. Küçük et al. [2021] propose an approach for fault localization in source code. In the evaluation experiments, the authors do not only show that the approach is performing better than the state of the art, they also investigate the impact of a specific dataset characteristic called *covariate imbalance*. They analyze the empirical relationship between covariate imbalance and the chosen performance metric and find that increasing covariate imbalance indeed leads to higher fault localization costs. They also show, however, that the presented technique is less affected by a covariate imbalance than the state of the art.

Explanatory knowledge provides insights into the *mechanisms* of how task assumptions and design decisions interact and influence the algorithm's performance. Key for establishing such knowledge is the comparative evaluation of different configurations of design, for instance, by including and excluding certain functionality, and then studying the effect on results.

Example 3.20. He et al. [2016] present a residual learning framework for image recognition to ease the training of networks that are substantially deeper than the state of the art. In their experiments, the authors analyze edge cases, such as a network with more than 1000 layers. They find that the performance of such extremely deep networks is not as good as the performance of networks with around 100 layers. They conclude that this is caused by overfitting and that networks with more than 1000 layers are unnecessarily large.

Research papers differ in the extent to which they emphasize *knowledge of* and *knowledge about* design. There are papers that focus on knowledge of design. A paper proposing a new algorithm is a typical example of this kind. Mind, however, that such papers often include a larger evaluation part that essentially provides knowledge about design, in which properties of the algorithm are analyzed to justify its underlying hypotheses. There are also papers offering knowledge about design only. Survey papers, papers on new theoretical insights about existing algorithms, or benchmarking studies belong to this category. These studies often do not propose new algorithms, but compare or analyze previously published ones.

Example 3.21. Wang et al. [2021] address the problem of cardinality estimation in the context of query optimization. More specifically, they set out to answer the question whether learned models for cardinality estimation are ready for use in practice. To answer this question, the authors conduct a comprehensive study with five learned methods and eight traditional methods for cardinality estimation and four real-world datasets. The study consists of three different parts and provides deep insights into the performance of the studied methods in different environments and when learned methods fail to deliver correct results.

Example 3.22. Arthur et al. [2011] provide theoretical evidence on why the very popular k-means clustering algorithm from the 1950s is usually very fast in practice – despite its provably exponential worst-case running time. They show, based on smoothed analysis, that the smoothed number of iterations is in fact bounded by a polynomial in the input size and the (inverse of) the standard deviation of a Gaussian perturbation of the data.

In the end, it is important to recall that algorithmic designs as mathematical abstractions and algorithm implementations as physical programs are epistemologically different [Colburn 2004]. Contributions to the formal verification debate in the 1970s and 1980s argued that program verification was neither practically [De Millo et al. 1979] nor philosophically feasible [Fetzer 1988]. Abstractions of computer languages, procedures and data have become so firmly established in computer science that many agree with Hoare [1985] who posits the equivalence of computer programs and mathematical expressions as much as programming languages and mathematical theory. However, the correspondence between physical systems executing algorithms and mathematical specifications is not self evidence and has to be firmly established: "An axiom system may just happen to describe physical reality, but that is for experimentation in science to decide" [Colburn 2004]. For that reason, both formal and empirical knowledge are equally important for advancing algorithm engineering.

4 Methodologies of Algorithm Engineering

We emphasized above that the general goal of algorithm engineering is to extend the body of knowledge of algorithm designs, algorithmic tasks, and corresponding knowledge. Up to this point, however, we have focused on *what* we can know. In this section, we elaborate on the question of *how* to systematically extend the body of knowledge. We identify four categories of such extensions:

(1) New or better knowledge *of tasks*.
(2) New or better knowledge *of designs*;
(3) New or better formal knowledge *about tasks and designs*;
(4) New or better empirical knowledge *about tasks and designs*.

Many papers present extensions of the mode knowledge that are *improvements* of one or the other kind [Gregor and Hevner 2013]. For instance, an improvement could be a new sorting algorithm that performs better than existing ones, a new theorem that identifies narrower bounds than earlier published ones, or more specific empirical results obtained by an experiment. Some papers

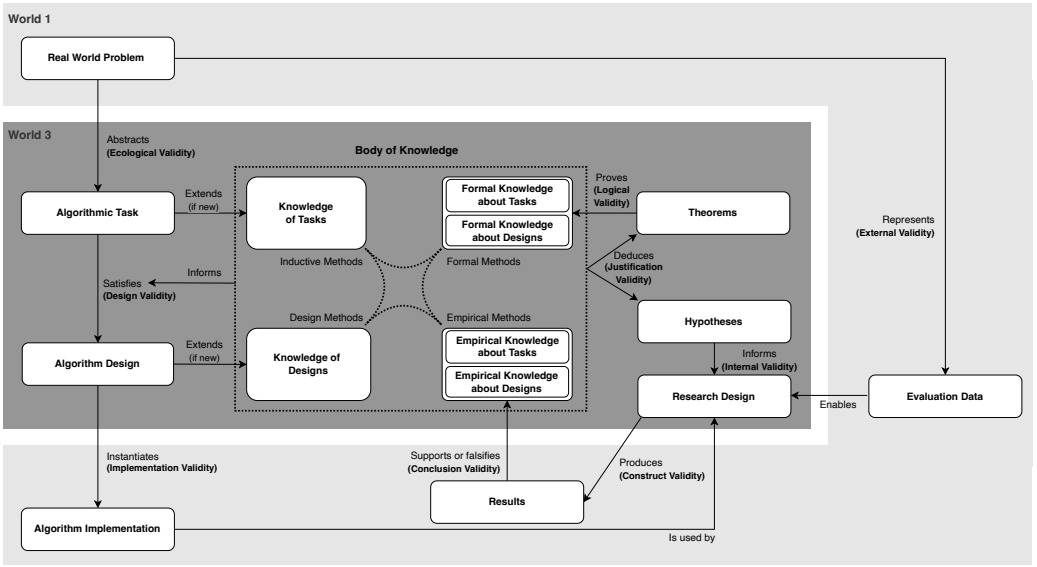

Fig. 3. A framework for algorithm engineering: the methodological perspective.

develop knowledge on how existing solutions can be transferred to problems that have not yet been discussed. Gregor and Hevner [2013] call this *exaptation*. Such knowledge extensions might be an algorithm that solves a newly defined algorithmic task, a theorem on an algorithm for which no theorems had been proven, or a first empirical study on an algorithm.

Figure 3 summarizes the methodological perspective of our framework. There are two key observations. First, additions to the body of knowledge are at the center of iterative processes [Sanders 2009; Wieringa 2014]. New algorithmic tasks and algorithm designs extend the corresponding knowledge categories; new theorems and corresponding proofs establish formal knowledge about algorithms; and new empirical results inform empirical knowledge about algorithms. Second, each knowledge type is associated with corresponding research methods: inductive methods, design methods, formal methods,[1] and empirical methods. Validity concerns are of specific importance for methods. Table 1 summarizes the key validity concerns for algorithm engineering. In the following, we describe each method in turn and highlight how these concerns inform and shape their application.

4.1 Methods for Generating Knowledge of Algorithmic Tasks

We already highlighted that the wicked nature of real-world problems is the reason why algorithmic tasks are not a simple isomorphic mapping of the problem and, therefore, need to be constructed. Various methods have been developed and used in operations research [Mingers and Rosenhead 2004; Pidd 2003], requirements engineering [Bourque et al. 2014], and visual analytics [Munzner 2014] for creating a specification of a real-world problem. Essentially, any research method that operates inductively on qualitative empirical data such as case study methods, focus group designs, interview studies, or think-aloud techniques [Recker 2021] can be applied.

Example 4.1. Wu et al. [2022] develop techniques for interactive pattern mining. They follow the design study methodology by Sedlmair et al. [2012] for analyzing the real-world problem domain of

[1]Here and in the following, *formal methods* refers to a broad set of rigorous theoretical techniques that allow to derive formal knowledge about algorithmic tasks and/or algorithm designs.

Table 1. Validity Concerns in Algorithm Engineering

Validity concern	Explanation	Affected entities
Ecological validity	Extent to which an algorithmic task or setup reflects real-world conditions and problem contexts (based on Ashcraft and Radvansky [2010]; Holleman et al. [2020]).	Algorithmic task
Design validity	Degree to which the internal structure and logic of an algorithm design is coherent, justified, and explainable [Larsen et al. 2020].	Algorithm design
Implementation validity	Extent to which an algorithm implementation faithfully instantiates the intended design and behaves as expected (based on Lukyanenko et al. [2015]; Lukyanenko and Parsons [2020]).	Algorithm implementation
External validity	Degree to which results generalize across datasets of interest (based on Cook et al. [2002]).	Empirical results
Justification validity	Degree how convincingly a hypothesis or theorem is supported by a deductive argument (based on [Lannin 2005]).	Theorems, hypotheses
Logical validity	Degree to which the syllogisms used in a proof preserve truth (based on Aristotle, reflected in Durand-Guerrier [2008]).	Proofs
Internal validity	Extent to which observed effects can be attributed to the treatment rather than to confounding factors [Wohlin et al. 2012].	Research design
Construct validity	Degree to which the measure of a construct accurately measures the intended property [O'Leary-Kelly and Vokurka 1998].	Measurement
Conclusion validity	Degree to which the results can reasonably be regarded as revealing the hypothesized connection [Cook and Campbell 1979; García-Pérez 2012].	Empirical results

racket sports. They interview domain experts, formalize their analysis tasks, and involve them in the design and evaluation of a prototype. In this way, they identify five analysis tasks and corresponding constraints. The resulting task descriptions inform the design of the author's interactive tactics mining algorithm. Input to this algorithm are raw sequence data of racket matches.

Models are often used for specification in general and for algorithmic tasks more specifically. In this context, a model is defined as (i) a mapping from some original (ii) by means of an abstraction operation (iii) in order to serve a specific purpose [Kühne 2006; Mendling 2008]. Various methods can be used for constructing models that structure problems. For an overview see Mingers and Rosenhead [2004].

Example 4.2. Algorithms play an important role in supporting supply chain operations. Biswas and Narahari [2004] provide an extensive analysis of supply chain tasks with the ambition to devise algorithmic decision support. They use different UML models: class diagrams for clarifying (a) the relationship between strategies and policies and (b) the objects of a supply chain, as well as activity diagrams for specifying (a) the steps of addressing the location problem and (b) the behaviour of the dealer object. Beyond that they use informal models to represent concept taxonomies and their proposed system architecture.

Examples of real-world problem instances also play an important role for describing tasks.

Example 4.3. Agarwal et al. [2018] analyze the problem of path planning amid a set of obstacles in a two-dimensional plane. Their work is motivated by robot motion planning used e.g., for surgeries.

They formalize this real-world problem to a task of constructing an approximate minimal-cost path algorithm. They illustrate the task using several examples of Voronoi diagrams.

The concern of *ecological validity* is of particular importance for the specification of tasks. It refers to what extent the task characteristics relate to the real world such that results will generalize to real-world problems [Ashcraft and Radvansky 2010; Holleman et al. 2020]. This is in essence a concern that touches upon the relationship between a class of real-world problems and an algorithmic task. Validating tasks is difficult, because the real-world problem might not yet be fully understood (unclear reference), the problem might not have yet occurred in the real world (hypothetical reference), or the change of the problem is difficult to predict (future reference) [Pidd 2003]. Problem structuring methods [Mingers and Rosenhead 2004] as well as validation and testing approaches from operations research [Pidd 2003], requirements engineering [Bourque et al. 2014] or design studies [Munzner 2014] can be used for assuring validity.

4.2 Methods for Generating Knowledge of Algorithm Designs

Algorithm designs are neither arbitrary nor self-evident. We call those research methods that support the generation of engineering knowledge of algorithm designs *design methods*. The outputs of these methods are essentially new algorithm designs. We distinguish deductive, inductive, abductive, and analogy engineering methods.

New algorithms can be developed by *deduction*. In essence, deduction implies an operation of specialization. The starting points of deduction can be existing algorithmic tasks and designs as well as more general design principles, such as divide-and-conquer.

Example 4.4. Ganapathi and Chowdhury [2022] present divide-and-conquer for parallelizing classical iterative sorting algorithms, including a recursive algorithm that combines quicksort and bubble sort. "*The aim is to sort the entire array* $A[0 \dots n-1]$. *The function BubbleSort*$(A[l \dots h])$ *sorts the subarray* $A[l \dots h]$. *The initial invocation to the algorithm is BubbleSort(*$A[0 \dots n-1]$*). The function in turn calls the Partition function. The Partition function brings the smallest* $n/2$ *elements to the left half and the largest* $n/2$ *elements to the right half of array A. Once the array A is partitioned, then BubbleSort is recursively called onto the left and right halves in parallel to sort the two halves. After the two halves are sorted recursively, the entire array* $A[0 \dots n-1]$ *will be sorted. When a subproblem reaches the base case, it is sorted using the standard iterative bubble sort logic.*" In this way, the authors deductively construct a new algorithm based on (1) the divide-and-conquer design principle also employed within quicksort and (2) the original bubble sort algorithm.

New algorithms can be developed by *induction*. In general, induction means generalizing from a set of specific problem instances. According to Manber [1988], particularly the analogy to mathematical induction is a promising methodological reference for algorithm design. Mathematical induction is a method for proving that a statement $P(n)$ is true for all $n \in \mathbb{N}$. The proof by induction works by showing that (1) $P(n)$ holds for a base case such as $n = 0$ or $n = 1$ and (2) $P(n)$ holds for $n + 1$. Manber [1988] argues that this method can be transferred to algorithm design by solving an arbitrary instance of the problem at hand by assuming that the same problem has already been solved for a smaller size.

Example 4.5. The application of inductive algorithm design can be illustrated using sorting algorithms [Manber 1988]: "*[...] given a sequence of* $n > 1$ *numbers to sort (it is trivial to sort one number), we can assume that we already know how to sort* $n-1$ *numbers. Then we can either sort the first* $n-1$ *numbers and insert the nth number in its correct position (which leads to an algorithm called insertion sort), or start by putting the nth number in its final position and then sort the rest (which is called selection sort). We need only to address the operation on the nth number.*" The proof

technique *loop invariant* can be seen as a direct translation from mathematical induction to proving the correctness of iterative algorithms such as insertion sort, see Cormen et al. [2009, Ch. 2].

It is worth pointing out that induction can also be used as a general bottom-up approach. Given a problem, the idea is to start by first solving a specific instance of this problem. By extending the obtained solution, one might then be able to solve the original, larger problem.

Example 4.6. To illustrate an inductive approach, consider the algorithmic task of *schema matching*. In essence, schema matching is the problem of generating correspondences between elements of two database schemas, e.g., for the purpose of data integration [Rahm and Bernstein 2001]. Given two database schemas and a respective ground truth (i.e., the correct correspondences as identified by a human), the inductive approach is to analyze a few of the correspondences from the ground truth and then devise a possible algorithmic strategy to identify them automatically, e.g., via simple string matching. By step-wise increasing the set of considered instances, a better solution can be developed.

New algorithms can also be devised by *abduction* [Tomiyama et al. 2003]. In essence, abduction is driven by anomalies [Sætre and Van de Ven 2021]. In the context of algorithms, such anomalies might be related to specific circumstances in which an algorithm does not meet its expectations. The abductive process takes these anomalies as a starting point for detailed diagnosis, which provides hunches for generating plausible modifications and extensions. These can then be evaluated to which extent they meet expectations.

Example 4.7. An example from process mining of abductive algorithm design is the α-algorithm for generating Petri net models from a set of execution sequences, so-called event logs. The original α-algorithm by Van der Aalst et al. [2004] is able to rediscover a Petri net from its set of execution sequences if this Petri net is structured and does not contain short loops (i.e., it does not produce execution sequences such as *aa* or *abab*). These limitations are addressed by later extensions. de Medeiros et al. [2004] propose the α+algorithm. It offers a solution to the short-loop problem by preprocessing the input sequences. Wen et al. [2007] work on the limitation that the original algorithm assumes that sequences are generated from a free-choice Petri net. Their α++algorithm is able to deal with non-local dependencies thanks to a modification of behavioural relations used by the algorithm.

New algorithms can also be developed by means of *analogy* or *metaphor* [Sörensen 2015]. This is the case when specific observations are generalized and detached from its context. Many algorithms stem from such generalizations. Most prominent are metaheuristics and optimization algorithms inspired by the phenomena discussed in the natural sciences such as simulated annealing [Kirkpatrick et al. 1983], particle swarm optimization [Kennedy and Eberhart 1995], or backpropagation for artificial neural networks [Werbos 1994]. Note that here we look at how these original algorithms have been formulated for the first time and not how they are applied to specific problems later. That latter case would be a deductive usage of existing algorithms.

Example 4.8. Genetic algorithms are inspired by the process of natural selection [Forrest 1996]. They build on the concepts of mutation, crossover, and selection to iteratively generate high-quality solutions to optimization and search problems. Among others, genetic algorithms have been used for image processing, scheduling problems, and parameter optimization in machine learning.

Some comments are warranted. First, the inspiration base for new algorithms is not restricted to the body of knowledge of algorithm engineering and other sciences. Feyerabend [1975] provides arguments why any sort of inspiration, even if factually wrong, can be fertile. Brown et al. [2005] have emphasized the value of artistic contributions that exhibit its merits in terms of subjective elegance or beauty. Algorithm engineering like any other act of human problem solving builds on creativity, for which dozens of mechanisms including and beyond the ones mentioned above exist.

For an extensive list see Simonton [2022, p.295]. Also serendipity is important to mention in this context as accidental discovery of things not sought for. It is estimated that one quarter of scientific discoveries are serendipitous [Thagard 2012]. Second, even if sought systematically, solutions in form of algorithms do not directly derive from algorithmic tasks. Design studies have shown that an understanding of the problem leads to solution ideas, which in turn provide a better problem understanding, and so forth. Dorst and Cross [2001] refer to this as a co-evolution of problem and solution space. Similar observations have been made for system development by Guindon [1990] and innovation management by von Hippel and von Krogh [2016]. These works describe design as an opportunistic thought process. This means that creativity and serendipity leave a white space for algorithm engineering that has not yet been fully conquered by systematic methods.

No matter which engineering method is used, a key concern for developing engineering knowledge of algorithms in this context is *design validity*. Adapting the definition by Larsen et al. [2020], we define this validity as the degree to which the internal structure of an algorithm is consistent, transparent, and explainable. The importance of explanation has been recently acknowledged by research on explainable artificial intelligence and decision making [Saeed and Omlin 2023]. Already Naur [1985] stated that a computer program can only be modified systematically if the developer has a theory about how the program works. In this way, design validity enables reproduction, critical assessment, and reuse in an incremental scientific process.

4.3 Methods for Generating Formal Knowledge about Tasks and Designs

Properties of algorithmic tasks and the outcomes of specific algorithms can be investigated using formal analysis methods and algorithm theory. In essence, such analysis is a branch of theoretical computer science. For that reason, analytical arguments are typically presented as theorems, for which their logical validity is demonstrated by corresponding proofs.

Analytical statements can be made about the complexity of algorithmic tasks and about the outcomes of specific algorithms. We associate outcomes with the efficiency of its computation and the effectiveness of the output. The *efficiency* of computation is typically assessed by means of asymptotic analysis. To this end, asymptotic formulas are used to describe the magnitude of computational steps and memory space required. These are expressed according to Landau [1909] as $\mathcal{O}(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$. In essence, $\mathcal{O}(\cdot)$ indicates the upper bound of the growth rate of a function that describes the efficiency, while $\Omega(\cdot)$ defines a lower bound, and $\Theta(\cdot)$ a bound above and below some strictly positive function. The symbols $o(\cdot)$ and $\omega(\cdot)$, in turn, refer to upper and lower bounds, respectively, that are asymptotically not tight [Cormen et al. 2009, Ch. 3.1]. Identifying such efficiency bounds is a key research objective of algorithm theory.

Example 4.9. Bodlaender et al. [2015] investigate algorithms for connectivity tasks[2] such as traveling salesperson, the number of Hamiltonian cycles, and the number of Steiner trees of a given graph $G = (V, E)$. They identify time complexity bounds for these problems that depend on the pathwidth and the treewidth of the input graphs. They formulate theorems and prove that algorithms with run-time complexity exist that challenge previously known bounds. Their bounds for these NP-hard problems are exponential in the treewidth tw (or in the pathwidth), but polynomial in the input size, i.e., $c^{\text{tw}} \cdot |V|^{\mathcal{O}(1)}$ for constants c that differ depending on the actual algorithmic task.

The *effectiveness* of output is typically assessed in terms of correctness and completeness. Note that for certain notions of output effectiveness like approximating human judgment or human

[2]We adopt terminology of our framework here. Algorithm theory typically uses the term *problem* for what we call here *algorithmic task*.

recognition of visual objects, the criteria of correctness and completeness do not objectively apply, such that effectiveness can be better understood as a notion of accuracy or solution quality. Many algorithms allow to tradeoff accuracy with efficiency.

Example 4.10. Fan et al. [2010] present an algorithm that performs graph pattern matching in cubic time if graph patterns are restricted to specific connectivity properties. They prove the correctness by showing that it terminates, that the result is indeed a match, and that this match is maximum. Also a proof for the cubic time complexity is provided.

Formal knowledge about algorithm designs is established by the help of mathematical proofs. A mathematical conjecture for which a proof was found is called a *theorem* [Velleman 2019]. Proofs make use of equivalences and inference rules for propositional or higher-order logic and for arithmetic.

Example 4.11. Gauch [2003, p.165-170] summarizes useful equivalences and inference rules for propositional logic. Equivalences include double negation, conditional exchange, commutativity, distribution, associativity, contraposition, DeMorgan's rules, exportation, redundancy, and biconditional. Gauch [2003, p.169-170] also states useful inference rules including modus ponens, modus tollens, simplification, conjunction, disjunctive syllogism, hypothetical syllogism, addition, and constructive dilemma. These rules build on axioms of identity, excluded middle, and noncontradiction.

Different methods for constructing proofs have been described including the construction of truth tables and deduction [Gauch 2003]. Methodological support is provided by Velleman [2019]. He distinguishes different types of *goals* that have to be proved and corresponding proof strategies.

(1) To prove $P \rightarrow Q$, assume P is true and then prove Q.
(2) To prove $P \rightarrow Q$, assume Q is false and then prove P is false.
(3) To prove $\neg P$, try to reformulate the goal and use another strategy.
(4) To prove $\neg P$, assume P is true and reach a contradiction.
(5) To prove $\forall x\, P(x)$, assume that x is arbitrary and prove $P(x)$.
(6) To prove $\exists x\, P(x)$, choose specific x and prove $P(x)$.
(7) To prove $P \wedge Q$, prove P and Q separately.
(8) To prove $P \leftrightarrow Q$, prove $P \rightarrow Q$ and $Q \rightarrow P$ separately.
(9) To prove $P \vee Q$, distinguish cases and prove either P or Q.
(10) To prove $P \vee Q$, assume P is false and prove Q.
(11) To prove $\exists! x P(x)$ (uniqueness), prove that $\forall y, z : P(y) \wedge P(z) \rightarrow y = z$.
(12) To prove $\exists! x P(x)$ (uniqueness), prove that $\exists x : P(x) \wedge P(y) \rightarrow x = y$.

Example 4.12. Lokshtanov et al. [2018] investigate lower bounds of run-time efficiency of algorithms operating on graphs with bounded treewidth. They present several theorems and lemmata with corresponding proofs. Their Theorem 1 has an implication as a goal and is proved using Velleman's Strategy (1) by construction. Their Lemma 11 includes three conjectures. For the first and the second, Strategy (9) is applied distinguishing two cases. For the third, Strategy (2) is used for constructing a contradiction. Lemma 15 is proved by help of Strategy (6). An arbitrary assignment is chosen for which the conclusion is proved. Many figures illustrate the idea of the proofs and concepts.

The key concern of proofs for theorems is logical validity. We define *logical validity* based on Aristotle as the degree to which the syllogisms used in a proof preserve truth [Durand-Guerrier 2008, p.374]. Crafting proofs is often a manual exercise performed by researchers. Tools can be used to support this exercise. Techniques from model checking and verification have been implemented

in *theorem provers* such as the ones by Detlefs et al. [2005]; Paulson [1994]; Schulz [2002]. Still, as Lakatos [1976] emphasizes, the eventual confidence in the logical validity of a proof is hardly a uni-directional deductive exercise, but rather a social process within the research community.

4.4 Methods for Generating Empirical Knowledge about Algorithm Designs

Empirical knowledge about algorithm designs refers to algorithmic tasks, designs, and their relationship. Such empirical knowledge can be methodologically established by *developing hypotheses* and testing them. New hypotheses can be derived inductively or deductively [Recker 2021]. The importance of theories as a knowledge context [Wieringa 2014] has been stressed in various empirical branches of computer science such as software engineering [Johnson et al. 2012; Ralph 2019; Wohlin et al. 2015] and computer visualization [Sedlmair et al. 2012].

4.4.1 Develop Hypotheses. The *inductive development of hypotheses* takes World 1 and its entities as a starting point. This includes the algorithmic problem, available implementations and data, as well as prior outcomes. Popper and Eccles [1977] states that *in order to learn more about World 1, [the researcher] must theorize.* Such *theorizing* is subjective to thoughts and observations about World 1 by the researcher in World 2 yielding hypotheses as World 3 entities. The *deductive development of hypotheses* takes World 3 and its different knowledge entities as a starting point. A reference for such deduction can be formal knowledge about designs and tasks, which may have been formally proved in earlier publications. Also empirical knowledge about designs and tasks published in prior research can inform the formulation of new hypotheses. Such knowledge can also stem from other disciplines.

Example 4.13. Sedgewick [1978] discusses the implementation of quicksort. Prior *formal* knowledge states that the average-case run-time performance increases with input size n according to $\mathcal{O}(n \log n)$. Sedgewick posits that the run time of a practical quicksort implementation can be improved – compared with the standard algorithm design found in textbooks. He relies on prior *empirical* (and to some extent also *formal*) knowledge on issues concerning recursion depth, small subarrays, worst-case input distributions, and different partitions.

Example 4.14. The graph drawing discipline develops algorithms that produce effective and aesthetic layouts for graphs. Ware et al. [2002] propose a set of metrics of a graph layout including the number of crossings, path bendiness, or shortest path length. Their experimental study shows that these metrics are correlated with cognitive effectiveness of the graph layout for human participants. The justification such metrics builds on cognitive theories [Malinova Mandelburger and Mendling 2021]. Metrics of graph aesthetics help to compare and improve algorithms [Gibson et al. 2013].

It is important to note that hypotheses are not always articulated in an explicit manner. This, however, does not mean that no hypotheses exist. If, for instance, the performance of a novel design is compared against the state-of-the-art design without stating a clear hypothesis, there still exists the implicit hypothesis that the novel design is better with respect to a relevant performance dimension.

The key concern for hypothesis and theorem formulation is *justification validity*. It refers to how *convincingly* a hypothesis or theorem is supported by a deductive argument [Lannin 2005]. This might be a matter of deduction from prior knowledge and, if applicable, its underlying theoretical argument. In certain settings, the backing of a theoretical justification is essential for establishing an explanation for the expected effect as stated by the hypothesis. It should again be noted that not all hypotheses in algorithm engineering are or can be justified by theory. Often, nonetheless, there is a rationale why an association, e.g., between a design and performance, should be present.

4.4.2 Derive Research Design. The term *research design* refers to the plan used to examine a research question of interest [Marczyk et al. 2010]. Various types of design can be distinguished according to how explicitly they define and justify hypotheses and control for alternative explanations. An *exploratory design* imposes no control and does not rely on justified hypotheses. It is appropriate in areas where algorithmic tasks and algorithm designs are not yet well understood.

A *correlational design* identifies hypotheses and corresponding measures. It does not control for alternative explanations and does not make use of explicit treatments. It can provide evidence for empirical connections, but does not offer causal insights. Such connections, for instance between the different data characteristics and performance, can be assessed using correlation and regression. Correlational studies have been used in empirical software engineering in the early 1980s, e.g., Basili et al. [1983], but were later largely replaced by experiments for better control of confounding factors.

An *experimental design* manipulates one or more experimental factors with two or more levels in a controlled way. Often, the algorithm design is considered as this factor with several competing algorithms representing the levels. Also properties of the input data can be varied. The effects of the factor can be evaluated using statistical tests of mean comparison [Demšar 2006].

Example 4.15. In the field of process mining, Augusto et al. [2018] conducted an exploratory study comparing the different algorithms for generating process models from event logs. These models are evaluated using accuracy and complexity measures, showing relative strengths and weaknesses of different algorithms. A follow-up correlational study by Augusto et al. [2022] investigates how properties of the input data are connected with properties of the generated process models using correlation and regression analysis. Experimental studies in this area are scarce. Experiments require the explicit manipulation by generating artificial event log data with controlled properties. An example of such an experimental study is Janssenswillen and Depaire [2019] who investigate the relation between process discovery algorithms and the fitness-precision tradeoff in an experimental setting.

A central concern of the research design is *internal validity*. A research design is internally valid if its manipulation is causally responsible for an observed effect [Wohlin et al. 2012, p.102,106]. Various factors that relate to the real-world problem, the algorithm itself, and the execution environment can affect validity [Barr et al. 1995]. Exploratory and correlational designs are weak in terms of internal validity. Experimental designs build on randomization and blocking to eliminate the effect of potentially confounding factors [Wohlin et al. 2012]. Randomization is a means to statistically eliminate the effect of confounding factors by randomly assigning units to treatment groups. Blocking eliminates the effect of confounding factors by keeping them at a constant level. For instance, consider a new algorithm is claimed to perform better than a recent alternative. Then, both can be run repeatedly on the same machine to avoid confounding effects. Furthermore, both can be implemented in the same programming language by the same developer to block these factors [Kriegel et al. 2017].

4.4.3 Build Implementation for Instrumentation. The *instrumentation* of the research design instantiates the research design by running an implementation of the algorithm with evaluation data as input to produce performance measurements of the computational process and of the generated output. An *implementation* of the algorithm design is used for the instrumentation. Any implementation uses a specific programming language on a specific operating system running on a specific microprocessor family, which all affect the performance of the algorithm.

Example 4.16. Kriegel et al. [2017] demonstrate that implementations of the same algorithm design can vary in performance by orders of magnitude. Run time of different DBScan

implementations was found to differ by four orders of magnitude. Even different versions of the same frameworks like ELKI and WEKA yielded substantial performance differences.

Several publications give practical advice and clarify which matters to consider for implementing the infrastructure for instrumentation. Various practical concerns are relevant such as software testing, handling errors, tracking provenance, and creating packages, as much as matters of team work and appropriate use of version control and command-line tools [Irving et al. 2021].

Example 4.17. Angriman et al. [2019] present guidelines on how to build the experimental pipeline addressing the following concerns. Principles of software testing such as unit tests should be considered. Implementation code should be managed using version control systems and shared as open source on platforms like Github. Experiments should be fully automated for reproducibility using scripts or tools such as SimexPal. Output files should be separated into experimental results, metadata, and supplementary data, in both a human readable, but also machine processable way.

A key concern here is *implementation validity* (or instantiation validity [Lukyanenko et al. 2015; Lukyanenko and Parsons 2020]). One threat for implementation validity is that the design generally leaves space for alternative implementation options. In this way, implementation decisions can turn out to be confounding factors [Lukyanenko et al. 2015]. Another threat is a potentially unfaithful implementation. Testing cannot fully assure the conformance with the design specification. Optimization, debugging and publishing code help to establish the desired confidence [Kriegel et al. 2017]. Formal verification with corresponding tools can assure that an implementation meets its design specification [D'Silva et al. 2008].

4.4.4 Choose Evaluation Data for Instrumentation. Algorithms run on input such that we can investigate their performance. This input for the instrumentation is called *evaluation data*. Different types of evaluation data can be distinguished along two dimensions: whether the data is publicly available or not and whether it stems from real-world or artificially generated problem instances.

In many research communities, *benchmark datasets* are made publicly available in order to encourage the performance evaluation of competing algorithms. The utilization of benchmark datasets affects the validity of research findings for several reasons. On the one hand, using benchmark data acts as a blocking mechanism and eliminates hidden confounding effects that could arise from using disparate datasets. On the other hand, the prolonged use of the same benchmark data could seduce researchers to tailor their designs to the characteristics of a specific dataset. This, in turn, increases the risk of overfitting the specific benchmark data and hurting the *external validity* of the research findings [Sim et al. 2003; Tichy 1998].

Benchmark datasets are available for various algorithmic tasks, such as part-of-speech tagging [Paul and Baker 1992], image recognition [Russakovsky et al. 2015], ontology matching [Algergawy et al. 2019], process mining [van Dongen et al. 2013], VRPs [Defryn et al. 2016], network analysis [Kunegis 2013; Leskovec and Sosic 2016], and various combinatorial optimization problems such as graph partitioning and graph clustering [Bader et al. 2018]. Benchmark datasets not only facilitate comparative evaluation. They also play an important role for exploring how specific features of realistic input data can be exploited [Sanders 2009].

Example 4.18. Different versions of the ImageNet dataset have been used for several classification challenges including the tasks of image classification, single-object localization, and object detection. Russakovsky et al. [2015] provide an overview of the results for the years 2012-2014 with error rates and corresponding confidence intervals for many published algorithms.

Benchmarking data might not be available for specific algorithmic tasks or at least not at the required level of richness. In this case, the use of private data is appropriate, even if that data cannot be shared.

Example 4.19. Alsger et al. [2016] develop an algorithm to estimate origin–destination trails based on smart card fare data. The authors use a unique smart card fare dataset obtained from the Australian public transport operator TransLink. This dataset provides rich details including boarding and alighting times and locations for each passenger. Using this dataset, the accuracy of the algorithm's capability to estimate origin–destination trails is evaluated.

Datasets, no matter if public or private, can be sampled from a real-world problem instance or artificially generated. The TransLink dataset is also an example of a *real-world* dataset. Real-world datasets are valuable for capturing the variability and complexity of a real-world problem, or at least the relevant and realistic range of to-be-expected input [Sanders 2009]. *Artificial* (computer-simulated or synthetic) datasets [Santner et al. 2003] can be used once relevant input data features are understood. Simulation affords the systematic variation of typical input ranges, which is instrumental for generating knowledge into how input data features translate into algorithmic performance variations. Simulated data is also useful when empirical data does not exist, is insufficient, imprecise [Hofmann 2013], or too large to store or share [Penschuck et al. 2020].

Example 4.20. Feature selection algorithms identify relevant features in high-dimensional data. Bolón-Canedo et al. [2013] review feature selection algorithms using artificial data for two reasons: "*1. Controlled experiments can be developed by systematically varying chosen experimental conditions, like adding more irrelevant features or noise in the input. This fact facilitates to draw more useful conclusions and to test the strengths and weaknesses of the existing algorithms. 2. The main advantage of artificial scenarios is the knowledge of the set of optimal features that must be selected; thus, the degree of closeness to any of these solutions can be assessed in a confident way.*"

The choice for real-world or artificially generated datasets has implications for external validity. *External validity* is the degree of how well we can generalize results across populations of interest [Cook et al. 2002, p.467], i.e., datasets of interest in our case. One aspect of external validity relates to *statistical generalizability*. This is the question whether the results obtained from a study can be generalized to the larger population from which the data was extracted [Winer 1999]. Ideally, the data should be a random sample from the population of interest, as it simplifies the task of achieving statistical generalizability. This approach is quite straightforward when dealing with artificial data. However, when dealing with real-world data, random sampling is often not feasible. In such cases, Cook et al. [2002] recommend employing purposive sampling to increase heterogeneity among the instances. In line with this argument, Kriegel et al. [2017] recommends to vary features related to the size and complexity of the input data.

Example 4.21. van der Kouwe et al. [2019] identify frequent benchmarking flaws in systems security evaluations, such as cherry-picking workloads, using unrealistic threat models, or omitting baseline comparisons. These issues pose a threat primarily to external validity, as the evaluation results often fail to generalize beyond the constrained conditions under which they were obtained. Although the benchmarks may be internally consistent, they do not reliably represent real-world environments and, thus, limit the applicability of the conclusions drawn.

Another aspect of external validity pertains to *realism*. This is the question to which extent the findings can be applied to a more natural or real-world setting. Real-world data, often referred to as *organic data*, are generally assumed to possess such realism. Nevertheless, as long as there are unknown discrepancies between the data generation processes across different populations, potential threats to external validity also exist for real-world data [Xu et al. 2020]. Achieving realism poses a greater challenge when working with artificial data. Experimental settings using artificial data often tend to simplify reality in order to unveil causal patterns, which may limit their applicability to more natural settings [Wieringa and Daneva 2015]. The primary challenge

when dealing with artificial data in this regard lies in the endeavor to study features of real-world problems and ensure that these features are adequately reflected in the artificial data [Lhermitte et al. 2011].

Example 4.22. Bolón-Canedo et al. [2013] conduct an extensive study using artificial datasets. In order to strengthen the external validity of their analysis, they extend their study to additionally include two real-world datasets. In this way, they demonstrate that conclusions of their study with artificial datasets also extend to real scenarios.

4.4.5 Measurement and Instrumentation. Hypotheses on algorithms have to be operationalized to *measures of performance*. Larsen et al. [2025] use the term *criterion*. These measures capture the efficiency of the algorithm execution and the effectiveness of the generated output. The measurement of *efficiency* is classically based on the number of steps an algorithm performs or run-time duration [Knuth 1974]. The *effectiveness* of the output can be measured in various ways based on the distance between the desired and actually generated output or an assessment of its usefulness.

Example 4.23. Graph drawing algorithms address the real-world problem of visualizing an abstract graph in an efficient and effective manner. Efficiency is often based on the run time and measured using, for instance, the CPU time in seconds [Hachul and Jünger 2005]. Effectiveness is often assessed according to how well the algorithm's output can assist humans solving real-world problems, for instance, as measured by the speed and accuracy of human task performance [Purchase 1998].

A key concern of measurement is *construct validity*. In essence, it points to the question whether a measure of a construct sufficiently measures the intended property [O'Leary-Kelly and Vokurka 1998]. More specifically, the measures have to be a valid and reliable operationalization of the intended concept [Fenton and Pfleeger 1996]. For run-time *efficiency measurement*, Kriegel et al. [2017] call for caution with wall-clock time, since it can be dominated by implementation details. *Effectiveness measurement* comes with more challenges and relates to properties of what is called gold standard, ground truth, reference data, or test data [Kondermann 2013; Toft et al. 2005; Zendel et al. 2017]. A gold standard is not the data that is processed by an algorithm, but the reference against which the generated output is compared.

If an objectively correct or optimal gold standard exists, the effectiveness of the solution can be assessed by the help of simple distance measures or measures such as precision, recall, mean square error, or area under the curve [Hossin and Sulaiman 2015; Saito and Rehmsmeier 2015]. A gold standard can be constructed by means of highly accurate devices, by simulating, or by generating synthetic data [Kondermann 2013]. If a gold standard is obtained by annotation of human judgement, the same measures can be applied, but have to be interpreted differently. Unknown human errors or biases likely exist depending on the type of task [Kondermann 2013]. Research on ImageNet has, initially as a by-product, yielded extensive insights into sophisticated large-scale annotation procedures using crowdsourcing [Russakovsky et al. 2015]. If a gold standard is absent, human judgement can be used to assess the usefulness of the output. This assessment can be based on psychometric measurement using questionnaire items, for instance, of the technology acceptance model [Venkatesh and Davis 2000]. Psychometric measurement items can also be newly developed using guidelines such as the ones proposed by Petter et al. [2007] and summarized by Recker [2021, p.99].

Example 4.24. Process drift detection is the analytical task of automatically identifying those points in time where the behaviour of a process changes. Algorithms for such drift detection take event sequences as input. Maaradji et al. [2017] evaluate their algorithm in terms of wall-clock time efficiency and effectiveness in terms of how precision and recall develop relative to time window size. Yeshchenko et al. [2021] evaluate their drift detection algorithm for efficiency using

wall-clock time and effectiveness based on user assessment of usefulness of the generated output. This assessment uses items of the technology acceptance model [Venkatesh and Davis 2000].

Research designs involving human participants have been critically appraised from an ethical standpoint. Awareness of ethical issues sprang from experiments that could cause medical and psychological harm for participants [Rutstein 1969; Slater et al. 2006]. Classical designs such as the Milgram experiments on obedience are distant to typical algorithm engineering research; however, ethical concerns are relevant, for instance, in studies where algorithmic output is evaluated using man-made gold standards.

Example 4.25. Miceli et al. [2020] conduct a participatory study on data annotation of images by humans in two annotation companies, one in Buenos Aires, Argentina, and one in Sofia, Bulgaria. Their findings highlight intertwined issues with biases, misunderstanding, and unawareness on the one hand and personal vulnerability of the individual workers on the other hand. As a consequence, Kazimzade and Miceli [2020] recommend the development of ethical standards for data annotation in terms of transparency, education of annotators, and corresponding regulation.

4.4.6 Draw Conclusions. The *comparison* of the expected with the observed results leads either to a rejection or a failure to reject the null hypothesis. In case of an *unexpected* failure to reject the null hypothesis, it is challenging to identify the cause of the deviation. The reason for failure can be a mismatch between real-world problem and implementation in World 1, human misinterpretations in World 2, or failures across worlds between algorithm task and real-world problem, algorithm design and task, or algorithm implementation and design [Staples 2015]. Methods for root cause analysis [Doggett 2005] can be used to single out what did work and what not. Upon this basis, there are two ways of responding to such unexpected results: a revision of the algorithm task, design, or implementation (*design revision*) or a revision of the hypotheses, research design, instrumentation, or measurement (*knowledge revision*) [Staples 2015]. In this way, scientific knowledge creation manifests itself as an iterative process that is directed towards aligning the algorithmic task and design, and any corresponding knowledge.

Example 4.26. Pittke et al. [2015] present algorithms to detect homonyms and synonyms in text labels of process model collections. Their evaluation analyzes efficiency in terms of processing time and effectiveness using precision and recall. Furthermore, they present examples to illustrate the capabilities of their approach. The examples highlight that resolving one case of a homonym or synonym potentially creates issues with other terms of the process model collection. This unexpected insight points to the opportunity of a design revision in future research.

In order to gain a profound understanding of the mechanisms that explain a certain result, it has been recommended to use a combination of macro- and micro-benchmarks.

Example 4.27. Traeger et al. [2008] describe various considerations for a careful evaluation design for file system and storage benchmarking. Specifically, the authors propose using at least one macro-benchmark, capturing a full application workload, with a diverse set of targeted micro-benchmarks, designed to analyze specific file system behaviors, such as small writes or metadata operations. This design supports internal validity, as the micro-benchmarks isolate and evaluate individual performance aspects with minimal confounding. At the same time, the inclusion of the macro-benchmark contributes to external validity by ensuring that overall system behavior is assessed under realistic, application-driven conditions. The complementary use of both types of benchmarks allows for fine-grained analysis while maintaining relevance to real-world workloads.

Also the *expected* rejection of the null hypothesis requires some further reflection. The key concern here is *conclusion validity*, i.e., to which degree the results can reasonably be regarded as

revealing the hypothesized connection [Cook and Campbell 1979; García-Pérez 2012]. Conclusion validity strongly emphasizes statistical analysis, but also covers qualitative considerations [Cozby 2007]. Conclusions can be drawn from statistical tests when their assumptions hold and the required significance levels are obtained. Additional qualitative analysis of outliers or data points that exhibit anomalies can help to further support conclusions [Hernández-Orallo 2017] or to investigate reasons of failure. Often, reservations on conclusions from empirical research are reported as *limitations* or *threats to validity*.

Example 4.28. Teymourian et al. [2020] present a clustering algorithm for the modularization of large-scale software systems. The authors conduct extensive experiments to show that their solution can compete with existing algorithms in terms of both modularization quality as well as run time. At the end of the paper, the authors have included a dedicated section entitled *Threats to Validity*. They use this section to mainly discuss concerns of external and internal validity. Among others, they discuss that the selected application might not be representative of software systems in general and that employing different evaluation metrics may lead to different results. Besides external and internal validity, the authors also discuss concerns, such as conclusion validity. For instance, they point out that the employed ground truth might not match the ground truth created by humans.

Researchers have to take a step back and stick to the facts when drawing conclusions. Results, no matter if expected or unexpected, may be subject to errors or fraudulent manipulation. Imperative for understanding is that all results can be explained [Small et al. 1997]. Gauch [2003] states that "*most fallacious reasoning in science results from accidental blunders. Unexamined presuppositions, bad data, and invalid logic lead to wrong conclusions, despite the best of intentions. To be honest, however, it must be admitted that scientists are only human, so occasionally errors result not from failure of competence but from failure of will.* [...] *Sometimes reason is usurped by desire, so the goal becomes not to embrace reality, but to evade reality. Logic is enlisted in this dirty, insincere business of rationalizing.*"

Reproducibility is therefore a key concern for establishing transparency and trust in scientific results. In essence, reproducibility requires that the algorithm can be implemented and evaluated independently by a different team [Angriman et al. 2019]. Weaker notions include replicability (a different research team conducts a study using the same source code and experimental setup) and repeatability (the same research team conducts a study using the same source code and experimental setup) [Plesser 2018]. To meet reproducibility as a requirement, evaluation data and implementations should be made available and citable [Stodden et al. 2016]. Different initiatives, such as making research data and research software meet the principles of findability, accessibility, interoperability, and reusability (FAIR), address this concern [Barker et al. 2022; Wilkinson et al. 2016], not only to reproduce results, but also to reuse research material to help gaining new insights.

Example 4.29. The Proceedings of the VLDB Endowment encourages authors to share their work in a reproducible way. Corresponding submissions are invited to include the following material:[3] a prototype system implementation, either the process to generate the input data or the actual data itself, the executable code as instrumentation to produce the raw result data, and the scripts to transform the raw data into the graphs included in the paper. The article by Tziavelis et al. [2020] on optimal algorithms for ranked enumeration of answers to full conjunctive queries is one of the accepted VLDB papers that has been evaluated to meet these reproducibility criteria.

5 Implications for Algorithm Engineering

This section discusses the implications of our framework for algorithm engineering. The framework highlights that research papers in this area require ontological clarity, epistemological precision,

[3] https://vldb.org/pvldb/reproducibility/, accessed 7 August 2023.

Table 2. Questions on Ontological Clarity

Ontological Clarity of Tasks
What is the task, and how does the task relate to real-world problems?
Which assumptions are made regarding input data, algorithmic processing, and the desired output?
Which performance requirements are relevant?
To which extent can the task be fully specified? How are users involved if the task is fuzzy?
Ontological Clarity of Designs
How does the algorithm work, and upon which design principles is the algorithm designed?
Which design decisions were made based on which assumptions?
Based on which considerations have parameter values been set?
If applicable, which implementation decisions have been made?
Which results does the implementation provide?

and methodological validity. These three pillars provide the foundation for a scientific contribution. In this context, a *contribution* is an extension or an improvement of the body of knowledge regarding *knowledge of* tasks and design, *knowledge about* tasks and design, and *knowledge how* to conduct research in terms of methodology. The accumulation of scientific contributions in the body of knowledge represents scientific progress.

In the following, we discuss which questions help capturing to which extent a contribution has ontological clarity, epistemological precision, and methodological validity.

5.1 Ontological Clarity

In essence, a contribution has *ontological clarity* if it is clear to which algorithmic task and algorithm design it relates, as much as corresponding real-world problem, implementations, and results. Table 2 summarizes questions that authors should consider answering in their papers with respect to *tasks*. The audience of the paper must be able to understand what the exact task is and to which real-world problems it relates. Some tasks are canonical and well understood in specific research areas. The more specific and newly identified a particular task is, the more detailed elaboration on its characteristics is required, for instance using models and examples. Clarity in this matter provides a basis for judging the appropriateness of assumptions in input data and algorithmic processing, as much as on the desired output and performance requirements. If the task cannot be fully specified or solved algorithmically, it should be clarified how users can interactively direct the computation and work with the output.

Table 2 also summarizes questions that authors should consider in their papers with respect to *designs*. The audience of the article must be able to understand how the algorithm works and how this addresses the requirements imposed by the task. Clarifying the design principles upon which the design rests, how design decisions were taken, and how parameter values were set, helps readers to judge plausibility. If the contribution relates to an implementation and corresponding experiments, relevant additional implementation decisions should be explained and made transparent.

5.2 Epistemological Precision

A contribution has *epistemological precision* if it precisely describes what is not known, how that lack of knowledge constitutes a research problem, and how it extends the body of knowledge in this respect. Table 3 summarizes related questions on *tasks*. The task can be described in different ways. The anticipated contribution defines the degree of precision to which the task has to be specified. If possible, descriptions from prior research should be used or appropriately adapted.

Table 3. Questions on Epistemological Precision

Epistemological Precision of Tasks
Why is the task described formally, semi-formally, informally, or else?
What prior research has been published on this task?
What knowledge of this specific task has been published?
Which new or improved knowledge of this task is presented?
Which formal knowledge about this task including theorems has been published?
Which new or improved theorems about this task are presented?
Which empirical knowledge including hypotheses about this task has been published?
Which new or improved hypotheses about this task are presented?
Epistemological Precision of Designs
What knowledge of a specific design has been published?
Which new or improved knowledge of a design is presented?
Why is the design described using pseudocode, flow charts, formal specification, or else?
Which formal knowledge about a design including theorems has been published?
Which new or improved theorems about a design are presented?
Which empirical knowledge including hypotheses about a design has been published?
Which new or improved hypotheses about a design are presented?
Do theorems and hypotheses relate to performance, sensitivity, uncertainty, and explanatory knowledge with corresponding causal factors?

Prior knowledge of a task defines the backdrop against which corresponding knowledge about the task can be identified. Understanding the boundaries of the body of knowledge requires a comprehensive review of the relevant literature, including formal and empirical contributions. Mind that not only publications on the exact task are relevant, but also more general classes of tasks and related tasks inform our understanding. The systematic review of the literature is the basis for summarizing what is known and the new or improved knowledge that a article provides.

Table 3 also summarizes related questions on *designs*. Similar concerns apply as for tasks. The anticipated contribution defines the degree of precision to which a design has to be specified. If possible, descriptions from prior research should be reused. Prior knowledge of a design defines the boundaries of relevant knowledge about it. Mind that this knowledge about designs is relative to the specific tasks under consideration. A comprehensive review of the relevant literature on related designs is required. If possible, insights into performance should be complemented with analysis of sensitivity, uncertainty, and explanatory factors.

5.3 Methodological Validity

A reader of a research article needs to understand the boundaries of the knowledge that is being created. A contribution has *methodological validity* if the latest research methods and methodological concerns are appropriately considered. Overall, we distinguish nine concerns. Table 4 summarizes corresponding questions.

The central question regarding validity is which validity concerns have to be considered. First, this consideration depends upon the *ontological focus of the contribution*.

— If a contribution relates to algorithmic tasks, then *ecological validity* is relevant.
— If a contribution presents an algorithm design, then *design validity* is important.
— If a contribution builds, among others, on an algorithm implementation, then *implementation validity* has to be considered.

Table 4. Questions on Methodological Validity

Methodological Validity of Tasks
Which methods are used to generate knowledge of tasks? If there are relevant threats to *ecological validity*, how have they been mitigated? To what extent can the study findings be generalized to real-world problems?
Methodological Validity of Design
Which methods are used to generate knowledge of designs? Why have deductive, inductive, abductive, analogy methods been used, or else? How did the co-evolution of problem and design space unfold? If there are relevant threats to *design validity*, how have they been mitigated? To which extent is the internal structure of an algorithm consistent, transparent and explainable?
To which asymptotic function class is the efficiency related? Which formal guarantees regarding correctness, completeness, and termination can be made? If there are relevant concerns regarding *logical validity*, how have they been addressed? What is the trade-off between efficiency and effectiveness? Which proof strategies are used to prove theorems and lemmata?
Why are hypotheses developed deductively or inductively, or else? If there are relevant threats to *justification validity*, how have they been mitigated? Do hypotheses have a theoretical justification or else?
Why is an exploratory, correlational, or experimental research design used? If there are relevant threats to *internal validity*, how have they been mitigated? What are potentially confounding factors and how are they controlled?
Which implementation method was chosen and why? How was the implementation tested? If there are relevant threats to *implementation validity*, how have they been mitigated?
Which evaluation data is chosen and why? Why is benchmark data or private data, real-world data or artificial used? If there are relevant threats to *external validity*, how have they been mitigated? Are feature ranges understood and realistic? How are relevant factors untangled and isolated? How is overfitting avoided?
How are appropriate measures selected or constructed? If there are relevant threats to *construct validity*, how have they been mitigated? Are the measures valid and reliable operationalizations of the intended concept? Is effectiveness measured by help of an objective gold standard, a gold standard obtained from human annotation, or human judgements in absence of a gold standard?
How can unexpected results be explained? What are arguments for a design revision or a knowledge revision? If there are relevant threats to *conclusion validity*, how have they been mitigated? Do assumptions of statistical tests hold? What can we learn from outliers or anomalous data points? Are the results reproducible according to established standards? Are data and research software shared upon FAIR principles?

Second, it is relevant to which *type of knowledge* a contribution refers.

— No matter if a contribution provides knowledge of or about tasks or design, *methodological transparency* is required. Based on which considerations and methods is this knowledge established?

- If a contribution offers new formal knowledge, then both *justification validity* of establishing theorems and *logical validity* of corresponding proofs has to be considered.
- If a contribution formulates new empirical knowledge, then *justification validity* of the hypotheses and *internal validity* of the research design have to be considered. Furthermore, the instrumentation of the research design requires the *implementation validity* of the implementation and the *external validity* of the evaluation data. *Construct validity* is relevant for the results produced and *conclusion validity* for drawing appropriate conclusions.

Third, there are different strategies to address validity concerns. The weakest consideration is to create transparency about limitations, for instance in a subsection on threats to validity. A better approach is to conduct additional analyses to investigate to which extent a certain aspect of validity could be affected by factors that were not addressed by the research design. If possible, an even better strategy is when validity concerns are anticipated and considered in the research design, for instance by help of manipulation checks, systematic sampling of evaluation data, or other means of quality control. All this helps to assure that the right conclusions can be drawn from the research at hand.

Fourth, it has to be noted that a contribution can never be perfect with respect to all validity concerns. In particular, the tradeoff between internal and external validity has often been discussed in the literature. To a certain extent, this tradeoff is inherited from the choice of a specific research design and research method [Recker 2021]. One way to address this is to integrate multiple different research designs into a research strategy [Roe and Just 2009]. Another perspective on imperfections is to judge them based on what is known on a particular research problem. If hardly any knowledge is available on an emerging research problem, the bar for methodological rigor might be lower than for well-researched problems. However, the bar is then still high to explicitly discuss threats to validity.

In the end, it is important to emphasize that *knowledge of* without *knowledge about* is insufficient. The central knowledge concern of algorithm engineering is the satisfaction relationship between algorithmic tasks and algorithm designs. This implies that validity concerns are relevant for any article that proposes a new algorithm.

5.4 Ethical Considerations along the Algorithm Engineering Framework

Ethics plays a pivotal role in algorithm engineering since algorithms increasingly impact various facets of society. Recognizing this, many authors discuss numerous dimensions of ethics in the context of designing and using algorithms including trust, fairness, transparency, and bias mitigation [Barocas et al. 2023; Martin 2019; Spiekermann et al. 2022]. If appropriately designed, algorithms can contribute to human well-being by positively influencing user behaviour and preventing negative outcomes for humans. Some fundamental dilemmas apply, such as exemplified by the moral machine experiment [Awad et al. 2018], which algorithms often approach from a utilitarian or a duty ethics angle. Preventing ethical issues is a key concern of approaches such as IEEE 7000 that cover ethical value requirements [Spiekermann 2021]. While discussing ethics in more detail is beyond the scope of this article, we would like to emphasize its importance in the context of algorithm engineering and point out that our framework can help identify where ethics may come into play.

The *ontological perspective* shows that ethical concerns are by no means limited to the design of algorithms. As the starting point is a real-world problem, one can already ask whether this real-world problem and the corresponding tasks are ethical. One prominent and widely discussed example is the task of creating so-called deepfakes, i.e., face-swapping technologies that enable the creation of fake images or videos that appear to be real [Meskys et al. 2020]. While there are several real-world problems that can ethically justify the deepfake creation task (e.g., bringing back dead actors), others are highly problematic (e.g., fake speeches of politicians or deep fake pornography).

The *epistemological perspective* points to the issue of creating knowledge about the design that provides insights into its ethical implications. Strictly speaking, such knowledge can be interpreted as a specific category of performance knowledge and to which extent corresponding requirements are satisfied. A difficulty of such an assessment is that dilemmas apply and ethical norms differ between individuals and cultural contexts [Awad et al. 2018].

The *methodological perspective* particularly emphasizes the role of evaluation data. It has been widely discussed that data, especially when used for training of AI models, can lead to different types of biases and discrimination [Ntoutsi et al. 2020]. Our framework further points at the relationship between evaluation data and the empirical knowledge we can generate about a design: data that is biased or not representative will lead to biased and non-representative knowledge. An aspect that might be less obvious is that the generation of evaluation data might be associated with ethical concerns as well. Kshetri [2021] discusses the issues associated with large data being manually labeled via crowdsourcing microtasks, often performed by individuals residing in developing countries who do not receive fair payment.

We recognize that the aspects discussed above are a few out of many ethical concerns. We, however, believe that our framework can structure a discussion on ethics in algorithm engineering.

6 Conclusion

In this article, we presented a research framework for algorithm engineering. Our framework builds on three areas discussed in the philosophy of science: ontology, epistemology and methodology. In essence, *ontology* describes algorithm engineering as being concerned with algorithmic problems, algorithmic tasks, algorithm designs and algorithm implementations. *Epistemology* describes the body of knowledge of algorithm engineering as a collection of prescriptive and descriptive knowledge, residing in World 3 of Popper's Three Worlds model. *Methodology* refers to the steps how we can systematically enhance our knowledge of specific algorithms. In this context, we identified nine validity concerns and discussed how researchers can respond to falsification.

Our framework has important methodological implications for researching algorithms in various areas of computer science. First, it offers a theoretical foundation for algorithm engineering grounded in the philosophy of science. This foundation clarifies how algorithms can be studied and which knowledge can be obtained. Second, our framework provides guidance for the methodological evaluation of newly proposed algorithms. Authors can use considerations that are related to the different validity concerns as a starting point. Third, this methodological grounding offers a framework for conducting research on algorithms in various areas of computer science in a way that supports an incremental research path. The concepts tied together in our framework might eventually help to foster knowledge transfer across sub-disciplines of computer science on how to research algorithms.

References

Russell L. Ackoff. 1979. The future of operational research is past. *J. of the Operational Research Society* 30, 2 (1979), 93–104.

Pankaj K. Agarwal, Kyle Fox, and Oren Salzman. 2018. An efficient algorithm for computing high-quality paths amid polygonal obstacles. *ACM Transactions on Algorithms (TALG)* 14, 4 (2018), 1–21.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. The design and analysis of computer algorithms. *Addison-Wesley Series in Computer Science and Information Processing, Reading, MA: Addison-Wesley, 1974* (1974).

Joan E. van Aken. 2004. Management research based on the paradigm of the design sciences: The quest for field-tested and grounded technological rules. *Journal of Management Studies* 41, 2 (2004), 219–246.

Alsayed Algergawy, Daniel Faria, Alfio Ferrara, Irini Fundulaki, Ian Harrow, Sven Hertling, Ernesto Jiménez-Ruiz, Naouel Karam, Abderrahmane Khiat, Patrick Lambrix, et al. 2019. Results of the ontology alignment evaluation initiative 2019. In *CEUR Workshop Proceedings*, Vol. 2536. RWTH, 46–85.

Azalden Alsger, Behrang Assemi, Mahmoud Mesbah, and Luis Ferreira. 2016. Validating and improving public transport origin–destination estimation algorithm using smart card fare data. *Transportation Research Part C: Emerging Technologies* 68, July (2016), 490–506.

Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. 2019. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms* 12, 7 (2019), 127.

David Arthur, Bodo Manthey, and Heiko Röglin. 2011. Smoothed analysis of the K-means method. *J. ACM* 58, 5 (2011), 1–31.

Mark H. Ashcraft and Gabriel A. Radvansky. 2010. *Cognition*. Pearson Education India.

Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio Maria Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. 2018. Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2018), 686–705.

Adriano Augusto, Jan Mendling, Maxim Vidgof, and Bastian Wurm. 2022. The connection between process complexity of event sequences and models discovered by process mining. *Information Sciences* 598, June (2022), 196–215.

Edmond Awad, Sohan Dsouza, Richard Kim, Jonathan Schulz, Joseph Henrich, Azim Shariff, Jean-François Bonnefon, and Iyad Rahwan. 2018. The moral machine experiment. *Nature* 563, 7729 (2018), 59–64.

David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. 2018. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining, 2nd Ed.*

Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. 1999. *Modern Information Retrieval*. Vol. 463. ACM Press New York.

Michelle Barker et al. 2022. Introducing the FAIR principles for research software. *Scientific Data* 9, 1 (2022), 622.

S. Barocas, M. Hardt, and A. Narayanan. 2023. *Fairness and Machine Learning: Limitations and Opportunities*. MIT Press.

Richard S. Barr, Bruce L. Golden, James P. Kelly, Mauricio G. C. Resende, and William R. Stewart. 1995. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics* 1, 1 (1995), 9–32.

Victor R. Basili, Richard W. Selby, and T. Phillips. 1983. Metric analysis and data validation across Fortran projects. *IEEE Transactions on Software Engineering* 9, 6 (1983), 652–663.

Jon Bentley. 1999. *Programming Pearls* (2nd ed.). Addison-Wesley Professional.

Jon Louis Bentley. 1980. Multidimensional divide-and-conquer. *Commun. ACM* 23, 4 (1980), 214–229.

Michele Bevilacqua, Tommaso Pasini, Alessandro Raganato, Roberto Navigli, et al. 2021. Recent trends in word sense disambiguation: A survey. In *International Joint Conference on Artificial Intelligence, IJCAI-21*.

Shantanu Biswas and Y. Narahari. 2004. Object oriented modeling and decision support for supply chains. *European Journal of Operational Research* 153, 3 (2004), 704–726.

Paul E. Black et al. 2020. *Dictionary of Algorithms and Data Structures*. Technical Report.

Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. 2015. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Information and Computation* 243, August (2015), 86–111.

Verónica Bolón-Canedo, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. 2013. A review of feature selection methods on synthetic data. *Knowledge and Information Systems* 34, 3 (2013), 483–519.

Pierre Bourque, Richard E. Fairley, et al. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

Aaron B. Brown, Anupam Chanda, Rik Farrow, Alexandra Fedorova, Petros Maniatis, and Michael L. Scott. 2005. The many faces of systems research-and how to evaluate them. In *HotOS*.

Mario Bunge. 1967. *Scientific Research II: The Search for Truth*. Springer-Verlag.

Mario Bunge. 1977. *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*. Vol. 3. Springer.

Jaelson Castro, Manuel Kolp, and John Mylopoulos. 2002. Towards requirements-driven information systems engineering: The tropos project. *Information Systems* 27, 6 (2002), 365–389.

Yang Chen, Andrew E. Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and D. Lo. 2020. A machine learning approach for vulnerability curation. *International Conference on Mining Software Repositories* (2020), 32–42.

Timothy Colburn. 2004. Methodology of computer science. In *The Blackwell Guide to the Philosophy of Computing and Information*, Luciano Floridi (Ed.). (2004), 318–326.

Thomas D. Cook and Donald T. Campbell. 1979. The design and conduct of true experiments and quasi-experiments in field settings. In *Reproduced in Part in Research in Organizations: Issues and Controversies*. Goodyear Publishing Company.

Thomas D. Cook, Donald Thomas Campbell, and William Shadish. 2002. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Vol. 1195. Houghton Mifflin Boston, MA.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press.

Paul C. Cozby. 2007. *Methods in Behavioral Research*. McGraw-Hill.

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Parameterized Algorithms*. Vol. 4. Springer.

Juan M. Carrillo De Gea, Joaquín Nicolás, José L. Fernández Alemán, Ambrosio Toval, Christof Ebert, and Aurora Vizcaíno. 2012. Requirements engineering tools: Capabilities, survey and assessment. *Information and Software Technology* 54, 10 (2012), 1142–1157.

Ana Karla A. de Medeiros, Boudewijn F. van Dongen, Wil M. P. van der Aalst, and AJMM Weijters. 2004. Process mining for ubiquitous mobile systems: An overview and a concrete algorithm. In *Int. Workshop on Ubiquitous Mobile Information and Collaboration Systems*. Springer, 151–165.

Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social processes and proofs of theorems and programs. *Commun. ACM* 22, 5 (1979), 271–280.

Christof Defryn, Kenneth Sörensen, and Trijntje Cornelissens. 2016. The selective vehicle routing problem in a collaborative environment. *European Journal of Operational Research* 250, 2 (2016), 400–411. DOI: https://doi.org/10.1016/j.ejor.2015.09.059

Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. 2004. Algorithm engineering. In *Current Trends in Theoretical Computer Science*. World Scientific, 83–104.

Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.

Narsingh Deo and Chi-Yin Pang. 1984. Shortest-path algorithms: Taxonomy and annotation. *Networks* 14, 2 (1984), 275–323.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics*. 1370–1380.

E. W. Dijkstra. 1968. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics* 8, 3 (1968), 174–186.

A. Mark Doggett. 2005. Root cause analysis: A framework for tool selection. *Quality Management Journal* 12, 4 (2005), 34–45.

Kees Dorst and Nigel Cross. 2001. Creativity in the design process: Co-evolution of problem–solution. *Design Studies* 22, 5 (2001), 425–437.

Iddo Drori, Brandon J. Kates, William R. Sickinger, Anant Girish Kharkar, Brenda Dietrich, Avi Shporer, and Madeleine Udell. 2020. GalaxyTSP: A new billion-node benchmark for TSP. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*.

Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178.

Viviane Durand-Guerrier. 2008. Truth versus validity in mathematical proof. *ZDM Mathematics Education* 40, 3 (2008), 373–384.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph pattern matching: From intractable to polynomial time. *Proceedings of VLDB* 3, 1-2 (2010), 264–275.

Norman E. Fenton and Shari L. Pfleeger. 1996. *Software Metrics—A Practical and Rigorous Approach (2. ed.)*. Int. Thomson.

Paolo Ferragina. 2023. *Pearls of Algorithm Engineering*. Cambridge University Press.

James H. Fetzer. 1988. Program verification: The very idea. *Commun. ACM* 31, 9 (1988), 1048–1063.

Paul Feyerabend. 1975. *Against Method: Outline of an Anarchistic Theory of Knowledge*. New Left Books.

Stephanie Forrest. 1996. Genetic algorithms. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 77–80.

Pramod Ganapathi and Rezaul Chowdhury. 2022. Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort. *Comput. J.* 65, 10 (2022), 2709–2719.

Miguel A. García-Pérez. 2012. Statistical conclusion validity: Some common threats and simple remedies. *Frontiers in Psychology* 3 (2012), 325.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of np-Completeness*.

Hugh G. Gauch. 2003. *Scientific Method in Practice*. Cambridge University Press.

Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *Int. Conf. on Data Engineering*. IEEE, 1428–1439.

Helen Gibson, Joe Faith, and Paul Vickers. 2013. A survey of two-dimensional graph layout techniques for information visualisation. *Information Visualization* 12, 3-4 (2013), 324–357.

Shirley Gregor, Leona Chandra Kruse, and Stefan Seidel. 2020. Research perspectives: The anatomy of a design principle. *Journal of the Association for Information Systems* 21, 6 (2020), 2.

Shirley Gregor and Alan R. Hevner. 2013. Positioning and presenting design science research for maximum impact. *MIS Quarterly* 37, 2 (2013), 337–355.

Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing method-level source code histories. In *International Conference on Software Engineering (ICSE)*. IEEE, 1510–1522.

Raymonde Guindon. 1990. Designing the design process: Exploiting opportunistic thoughts. *Human–Computer Interaction* 5, 2-3 (1990), 305–344.

Stefan Hachul and Michael Jünger. 2005. An experimental comparison of fast algorithms for drawing general large graphs. In *International Symposium on Graph Drawing*. Springer, 235–250.

Jon G. Hall and Lucia Rapanotti. 2017. A design theory for software engineering. *Information and Software Technology* 87, July (2017), 46–61.

David Harel and Yishai A. Feldman. 2004. *Algorithmics: The Spirit of Computing*. Pearson Education.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

Carl G. Hempel and Paul Oppenheim. 1948. Studies in the logic of explanation. *Philosophy of Science* 15, 2 (1948), 135–175.

José Hernández-Orallo. 2017. Evaluation in artificial intelligence: From task-oriented to ability-oriented measurement. *Artificial Intelligence Review* 48, 3 (2017), 397–447.

Charles Antony Richard Hoare. 1985. The mathematics of programming. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1–18.

Marko Hofmann. 2013. Ontologies in modeling and simulation: An epistemological perspective. In *Ontology, Epistemology, and Teleology for Modeling and Simulation*. Springer, 59–87.

Gijs A. Holleman, Ignace T. C. Hooge, Chantal Kemner, and Roy S. Hessels. 2020. The 'Real-World Approach' and Its Problems: A Critique of the Term Ecological Validity. *Frontiers in Psychology* 11 (2020).

John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc.

Mohammad Hossin and MN Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process* 5, 2 (2015), 1.

Juraj Hromkovič. 2013. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer Science & Business Media.

Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson. 2021. *Research Software Engineering with Python: Building Software that Makes Research Possible*. CRC Press.

Gert Janssenswillen and Benoît Depaire. 2019. Towards confirmatory process discovery: Making assertions about the underlying system. *Business & Information Systems Engineering* 61, 6 (2019), 713–728.

Pontus Johnson, Mathias Ekstedt, and Ivar Jacobson. 2012. Where's the theory for software engineering? *IEEE Software* 29, 5 (2012), 96–96.

Ramakrishna Karedla, J. Spencer Love, and Bradley G Wherry. 1994. Caching strategies to improve disk system performance. *Computer* 27, 3 (1994), 38–46.

Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. 2021. A (slightly) improved approximation algorithm for metric TSP. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 32–45.

Gunay Kazimzade and Milagros Miceli. 2020. Biased priorities, biased outcomes: Three recommendations for ethics-oriented data annotation practices. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*. 71–71.

Peter G. W. Keen. 1980. MIS research: Reference disciplines and a cumulative tradition. *Proceedings of the International Conference on Information Systems*. 9–18.

James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, Vol. 4. IEEE, 1942–1948.

Scott Kirkpatrick, C. Daniel Gelatt Jr, and Mario P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.

Jon Kleinberg and Eva Tardos. 2006. *Algorithm Design*. Pearson Education India.

Donald E. Knuth. 1968. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley.

Donald E. Knuth. 1969. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley.

Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

Donald E. Knuth. 1974. Computer programming as an art. *Commun. ACM* 17, 12 (1974), 667–673.

Daniel Kondermann. 2013. Ground truth design principles: an overview. In *Proceedings of the International Workshop on Video and Image Ground Truth in Computer Vision Applications*. 1–4.

Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. 2017. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowledge and Information Systems* 52, 2 (2017), 341–378.

Nir Kshetri. 2021. Data labeling for the artificial intelligence industry: Economic impacts in developing countries. *IT Professional* 23, 2 (2021), 96–99.

Yiğit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *International Conference on Software Engineering*. IEEE, 649–660.

Thomas S. Kuhn. 1962. *The Structure of Scientific Revolutions*. 50th Anniversary Edition, University of Chicago Press (2012).

Thomas Kühne. 2006. Matters of (meta-) modeling. *Software & Systems Modeling* 5, 4 (2006), 369–385.

Jérôme Kunegis. 2013. KONECT: The Koblenz network collection. In *22nd Int. World Wide Web Conference*. 1343–1350.

Imre Lakatos. 2015/1976. *Proofs and refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

Edmund Landau. 1909. *Handbuch der Lehre von der Verteilung der Primzahlen, 2 Bände*. Teubner.

Maurice Landry, Jean-Louis Malouin, and Muhittin Oral. 1983. Model validation in operations research. *European Journal of Operational Research* 14, 3 (1983), 207–220.

Kenneth Lange. 2020. *Algorithms from THE BOOK*. Society for Industrial and Applied Mathematics, Philadelphia, PA. DOI: https://doi.org/10.1137/1.9781611976175 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976175

John K. Lannin. 2005. Generalization and justification: The challenge of introducing algebraic reasoning through patterning activities. *Mathematical Thinking and learning* 7, 3 (2005), 231–258.

Gilbert Laporte. 1992. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59, 3 (1992), 345–358.

Kai R. Larsen, Roman Lukyanenko, Roland M. Mueller, Veda C. Storey, Debra VanderMeer, Jeffrey Parsons, and Dirk S. Hovorka. 2020. Validity in design science research. In *Int. Conf. on Design Science Research in Information Systems and Technology*. Springer, 272–282.

Kai R. Larsen, Roman Lukyanenko, Roland M. Mueller, Veda C. Storey, Jeffrey Parsons, Debra vanderMeer, and Dirk S. Hovorka. 2025. Validity in design science. *MIS Quarterly* (2025). DOI : https://doi.org/10.25300/MISQ/2024/18064

Jure Leskovec and Rok Sosic. 2016. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1:1–1:20. DOI: https://doi.org/10.1145/2898361

Stef Lhermitte, Jan Verbesselt, Willem W. Verstraeten, and Pol Coppin. 2011. A comparison of time series similarity measures for classification and change detection of ecosystem dynamics. *Remote Sensing of Environment* 115, 12 (2011), 3129–3152.

Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. 2018. Known algorithms on graphs of bounded treewidth are probably optimal. *ACM Transactions on Algorithms (TALG)* 14, 2 (2018), 1–30.

Roman Lukyanenko, Joerg Evermann, and Jeffrey Parsons. 2015. Guidelines for establishing instantiation validity in IT artifacts: A survey of IS research. In *Int. Conf. on Design Science Research in Information Systems*. Springer, 430–438.

Roman Lukyanenko and Jeffrey Parsons. 2020. Design theory indeterminacy: What is it, how can it be reduced, and why did the polar bear drown? *Journal of the Association for Information Systems* 21, 5 (2020).

Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. 2017. Detecting sudden and gradual drifts in business processes from execution traces. *IEEE Transactions on Knowledge and Data Engineering* 29, 10 (2017), 2140–2154.

Monika Malinova Mandelburger and Jan Mendling. 2021. Cognitive diagram understanding and task performance in systems analysis and design. *MIS Quarterly* 45, 4 (2021), 2101–2157.

Udi Manber. 1988. Using induction to design algorithms. *Commun. ACM* 31, 11 (1988), 1300–1313.

Geoffrey R. Marczyk, David DeMatteo, and David Festinger. 2010. Essentials of research design and methodology. Vol. 2. John Wiley & Sons.

Kirsten Martin. 2019. Designing ethical algorithms. *MIS Quarterly Executive* 18, 2 (2019). Available at: https://aisel.aisnet.org/misqe/vol18/iss2/5

Jan Mendling. 2008. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness*. Vol. 6. Springer Science & Business Media.

Edvinas Meskys, Julija Kalpokiene, Paulius Jurcys, and Aidas Liaudanskas. 2020. Regulating deep fakes: Legal and ethical considerations. *Journal of Intellectual Property Law & Practice* 15, 1 (2020), 24–31.

Miriah Meyer, Michael Sedlmair, P. Samuel Quinan, and Tamara Munzner. 2015. The nested blocks and guidelines model. *Information Visualization* 14, 3 (2015), 234–249.

Milagros Miceli, Martin Schuessler, and Tianling Yang. 2020. Between subjectivity and imposition: Power dynamics in data annotation for computer vision. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–25.

John Mingers and Jonathan Rosenhead. 2004. Problem structuring methods in action. *European Journal of Operational Research* 152, 3 (2004), 530–554.

Michael Mitzenmacher. 2015. Theory without experiments: Have we gone too far? *Commun. ACM* 58, 9 (2015), 40–42.

Tamara Munzner. 2014. *Visualization Analysis and Design*. CRC Press.

Peter Naur. 1985. Programming as theory building. *Microprocessing and Microprogramming* 15, 5 (1985), 253–261.

Roberto Navigli. 2009. Word sense disambiguation: A survey. *ACM Computing Surveys (CSUR)* 41, 2 (2009), 10.

Eirini Ntoutsi, Pavlos Fafalios, Ujwal Gadiraju, Vasileios Iosifidis, Wolfgang Nejdl, Maria-Esther Vidal, Salvatore Ruggieri, Franco Turini, Symeon Papadopoulos, Emmanouil Krasanakis, et al. 2020. Bias in data-driven artificial intelligence systems–An introductory survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 3 (2020), 1–14.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

Scott W. O'Leary-Kelly and Robert J. Vokurka. 1998. The empirical assessment of construct validity. *Journal of Operations Management* 16, 4 (1998), 387–405.

Aziz Ouaarab, Belaïd Ahiod, and Xin-She Yang. 2014. Discrete cuckoo search algorithm for the travelling salesman problem. *Neural Computing and Applications* 24, 7 (2014), 1659–1669.

Christos H. Papadimitriou. 1977. The euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science* 4, 3 (1977), 237–244.

Douglas B. Paul and Janet M. Baker. 1992. The design for the wall street journal-based CSR corpus. In *Proceedings of the Workshop on Speech and Natural Language*. Association for Computational Linguistics, 357–362.

Lawrence C. Paulson. 1994. *Isabelle: A Generic Theorem Prover*. Springer.

Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. 2007. A design science research methodology for information systems research. *Journal of Management Information Systems* 24, 3 (2007), 45–77.

Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. 2020. Recent advances in scalable network generation. (2020). Retrieved from https://arxiv.org/abs/2003.00736

Stacie Petter, Detmar Straub, and Arun Rai. 2007. Specifying formative constructs in information systems research. *MIS Quarterly* (2007), 623–656.

Michael Pidd. 2003. *Tools for Thinking: Modelling in Management Science*. Wiley.

Fabian Pittke, Henrik Leopold, and Jan Mendling. 2015. Automatic detection and resolution of lexical ambiguity in process models. *IEEE Transactions on Software Engineering* 41, 6 (2015), 526–544.

Hans E. Plesser. 2018. Reproducibility vs. replicability: A brief history of a confused terminology. *Frontiers in Neuroinformatics* 11 (2018), 76.

Karl Popper. 1979. *Three Worlds*. Ann Arbor,: University of Michigan.

Karl R. Popper and John C. Eccles. 1977. The Worlds 1, 2 and 3. In *The Self and its Brain*. Springer, 36–50.

Helen C. Purchase. 1998. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages & Computing* 9, 6 (1998), 647–657.

Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal* 10, 4 (2001), 334–350.

Paul Ralph. 2019. Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Trans. Software Eng.* 45, 7 (2019), 712–735. DOI: https://doi.org/10.1109/TSE.2018.2796554

Jan Recker. 2021. *Scientific Research in Information Systems: A Beginner's Guide* (2nd ed.). Springer Science & Business Media.

Horst W. J. Rittel and Melvin M. Webber. 1973. Dilemmas in a general theory of planning. *Policy Sciences* 4, 2 (1973), 155–169.

Brian E. Roe and David R. Just. 2009. Internal and external validity in economics research: Tradeoffs between experiments, field experiments, natural experiments, and field data. *American Journal of Agricultural Economics* 91, 5 (2009), 1266–1271.

Gordon Frederick Crichton Rogers. 1983. *The Nature of Engineering: A Philosophy of Technology*. Macmillan International Higher Education.

Olga Russakovsky et al. 2015. Imagenet large scale visual recognition challenge. *Int. J. of Computer Vision* 115, 3 (2015), 211–252.

David D. Rutstein. 1969. The ethical design of human experiments. *Daedalus* 98, 2 (1969), 523–541.

Waddah Saeed and Christian Omlin. 2023. Explainable AI (XAI): A systematic meta-survey of current challenges and future opportunities. *Knowledge-Based Systems* 263, March (2023), 110273.

Alf Steiner Sætre and Andrew Van de Ven. 2021. Generating theory by abduction. *Academy of Management Review* 46, 4 (2021), 684–701.

Andrew P. Sage. 1992. *Systems Engineering*. Vol. 6. John Wiley & Sons.

Takaya Saito and Marc Rehmsmeier. 2015. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PloS One* 10, 3 (2015), e0118432.

Peter Sanders. 2004. Algorithms for Memory Hierarchies (Column: Algorithmics). *Bull. EATCS* 83 (2004), 67–85.

Peter Sanders. 2009. Algorithm engineering–an attempt at a definition. In *Efficient Algorithms*. Springer, 321–340.

Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. 2019. *Sequential and Parallel Algorithms and Data Structures—The Basic Toolbox*. Springer. DOI: https://doi.org/10.1007/978-3-030-25209-0

Thomas J. Santner, Brian J. Williams, William Notz, and Brain J. Williams. 2003. *The Design and Analysis of Computer Experiments*.

David A. Scanlan. 1989. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software* 6, 5 (1989), 28–36.

Stephan Schulz. 2002. E–a brainiac theorem prover. *Ai Communications* 15, 2-3 (2002), 111–126.

Robert Sedgewick. 1978. Implementing quicksort programs. *Commun. ACM* 21, 10 (1978), 847–857.

Robert Sedgewick and Philippe Flajolet. 2013. *An Introduction to the Analysis of Algorithms* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.

Michael Sedlmair, Miriah Meyer, and Tamara Munzner. 2012. Design study methodology: Reflections from the trenches and the stacks. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2431–2440.

Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. 2003. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering*. IEEE Computer Society, 74–83.

Herbert A. Simon. 1969. *The Sciences of the Artificial*. MIT Press.

Dean Keith Simonton. 2022. Serendipity and creativity in the arts and sciences: A combinatorial analysis. In *The Art of Serendipity*. Springer, 293–320.

Mel Slater, Angus Antley, Adam Davison, David Swapp, Christoph Guger, Chris Barker, Nancy Pistrang, and Maria V. Sanchez-Vives. 2006. A virtual reprise of the Stanley Milgram obedience experiments. *PloS One* 1, 1 (2006), e39.

Christopher Small, Narendra Ghosh, Hany Saleeb, Margo Seltzer, and Keith Smith. 1997. Does systems research measure up. *Computer Science Department Technical Report, Harvard University, TR-16-97* 128 (1997).

Kenneth Sörensen. 2015. Metaheuristics–the metaphor exposed. *International Transactions in Operational Research* 22, 1 (2015), 3–18.

Sarah Spiekermann. 2021. What to expect from IEEE 7000: The first standard for building ethical systems. *IEEE Technology and Society Magazine* 40, 3 (2021), 99–100.

Sarah Spiekermann, Hanna Krasnova, Oliver Hinz, Annika Baumann, Alexander Benlian, Henner Gimpel, Irina Heimbach, Antonia Köster, Alexander Maedche, Björn Niehaves, et al. 2022. Values and ethics in information systems: A state-of-the-art analysis and avenues for future research. *Business & Information Systems Engineering* 64, 2 (2022), 247–264.

Mark Staples. 2014. Critical rationalism and engineering: Ontology. *Synthese* 191, 10 (2014), 2255–2279.

Mark Staples. 2015. Critical rationalism and engineering: Methodology. *Synthese* 192, 1 (2015), 337–362.

Victoria Stodden, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P. A. Ioannidis, and Michela Taufer. 2016. Enhancing reproducibility for computational methods. *Science* 354, 6317 (2016), 1240–1241.

Navid Teymourian, Habib Izadkhah, and Ayaz Isazadeh. 2020. A fast clustering algorithm for modularization of large-scale software systems. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1451–1462.

P. Thagard. 2012. Creative combination of representations: Scientific discovery and technological invention. In *Psychology of Science: Implicit and Explicit Processes*, R. W. Proctor and E. J. Capaldi (Eds.). Oxford University Press, 389–404. https://doi.org/10.1093/acprof:oso/9780199753628.003.0016

Walter F. Tichy. 1998. Should computer scientists experiment more? *Computer* 31, 5 (1998), 32–40.

Nils Toft, Erik Jørgensen, and Søren Højsgaard. 2005. Diagnosing diagnostic tests: Evaluating the assumptions underlying the estimation of sensitivity and specificity in the absence of a gold standard. *Prev. Veterinary Med.* 68, 1 (2005), 19–33.

Tetsuo Tomiyama, Hideaki Takeda, Masaharu Yoshioka, and Yoshiki Shimomura. 2003. Abduction for creative design. In *Int. Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Vol. 37017. 543–552.

Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)* 4, 2 (2008), 1–56.

Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. In *Proceedings of VLDB*, Vol. 13. 1582.

Jeffrey D. Ullman. 2015. Experiments as research validation: Have we gone too far? *Commun. ACM* 58, 9 (2015), 37–39.

Wil Van der Aalst, Ton Weijters, and Laura Maruster. 2004. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering* 16, 9 (2004), 1128–1142.

Wil M. P. Van der Aalst. 2011. Process discovery: An introduction. *Process Mining: Discovery, Conformance and Enhancement of Business Processes* (2011), 125–156.

Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking flaws in systems security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 310–325.

Boudewijn F. van Dongen, Barbara Weber, Diogo R. Ferreira, and Jochen De Weerdt. 2013. Report: Business process intelligence challenge 2013. In *Business Process Management Workshops (LNBIP)*, Vol. 171. Springer, 79–87.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *31st Conference on Neural Information Processing Systems (NIPS 2017)*, Vol. 30, Long Beach, CA, USA.

Daniel J. Velleman. 2019. *How to Prove It: A Structured Approach* (Second Edition ed.). Cambridge University Press.

Viswanath Venkatesh and Fred D. Davis. 2000. A theoretical extension of the technology acceptance model: Four longitudinal field studies. *Management Science* 46, 2 (2000), 186–204.

Walter G. Vincenti. 1990. *What Engineers Know and How They Know it.* Vol. 141. Baltimore: Johns Hopkins University Press.

Eric von Hippel and Georg von Krogh. 2016. Crossroads–Identifying viable "need–solution pairs": Problem solving without problem formulation. *Organization Science* 27, 1 (2016), 207–221.

Yair Wand and Ron Weber. 1990. An ontological model of an information system. *IEEE Transactions on Software Engineering* 16, 11 (1990), 1282–1292.

Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proceedings of VLDB* 14, 9 (2021), 1640–1654.

Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. 2002. Cognitive measurements of graph aesthetics. *Information Visualization* 1, 2 (2002), 103–110.

Lijie Wen, Wil M. P. Van Der Aalst, Jianmin Wang, and Jiaguang Sun. 2007. Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery* 15, 2 (2007), 145–180.

Paul John Werbos. 1994. *The roots of backpropagation: From Ordered Derivatives to Neural networks and Political Forecasting*. Vol. 1. John Wiley & Sons.

Roel J. Wieringa. 2014. *Design Science Methodology for Information Systems and Software Engineering*. Springer.

Roel J. Wieringa and Maya Daneva. 2015. Six strategies for generalizing software engineering theories. *Sci. Comput. Program.* 101, April (2015), 136–152.

Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, et al. 2016. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data* 3, 1 (2016), 1–9.

Russell S. Winer. 1999. Experimentation in the 21st century: The importance of external validity. *Journal of the Academy of Marketing Science* 27, 3 (June 1999), 349–358. DOI: https://doi.org/10.1177/0092070399273005

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.

Claes Wohlin, Darja Šmite, and Nils Brede Moe. 2015. A general theory of software engineering: Balancing human, social and organizational capitals. *Journal of Systems and Software* 109, November (2015), 229–242.

Michael Joseph Wolfe. 1996. *High Performance Compilers for Parallel Computing*. Vol. 102. Addison-Wesley Reading.

Jiang Wu, Dongyu Liu, Ziyang Guo, and Yingcai Wu. 2022. Rasipam: Interactive pattern mining of multivariate event sequences in racket sports. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2022), 940–950.

H. Xu, N. Zhang, and L. Zhou. 2020. Validity concerns in research using organic data. *Journal of Management* 46, 7 (2020), 1257–1274.

Anton Yeshchenko, Claudio Di Ciccio, Jan Mendling, and Artem Polyvyanyy. 2021. Visual drift detection for event sequence data of business processes. *IEEE Transactions on Visualization and Computer Graphics* 28, 8 (2021), 3050–3068.

Oliver Zendel, Markus Murschitz, Martin Humenberger, and Wolfgang Herzner. 2017. How good is my test data? Introducing safety analysis for computer vision. *International Journal of Computer Vision* 125, 1 (2017), 95–109.

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of AAAI*.

Received 30 October 2023; revised 6 August 2025; accepted 15 September 2025